# Integration of Open Data into the JValue Hub

BACHELOR THESIS

**Dirk Engelhard**

Submitted on 2 December 2024


Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

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

# Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

---

Erlangen, 2 December 2024

# License



---

Erlangen, 2 December 2024

# Abstract

When working with open data, it is essential to have easy access to the necessary sources. Open data portals provide a vast amount of data, but finding the right platform and data for a specific project can be challenging. This thesis aims to integrate open data into the JValue Hub, a web-based collaboration platform for working on data pipelines built with Jayvee, a domain-specific language for building data pipelines.

We design and implement a multistep solution that imports open data from various sources and processes it into a format suitable for the JValue Hub. This will lay the foundation for features that enable users to search for and access open data directly from the JValue Hub without needing to switch between different platforms.

# Contents

# List of Figures

# List of Tables

# Listings

# Acronyms

**API** Application Programming Interface

**CKAN** Comprehensive Knowledge Archive Network

**CSV** Comma-separated values

**DCAT** Data Catalog Vocabulary

**JSON** JavaScript Object Notation

**RDF** Resource Description Framework

**RAM** Random Access Memory

**REST** Representational State Transfer

**SPARQL** SPARQL Protocol And RDF Query Language

**UML** Unified Modeling Language

**URI** Unique Resource Identifier

**XML** Extensible Markup Language

# 1 Introduction

‘Make open data safe, easy, and reliable to use.‘ — This is the headline of the official website from the JValue project (JValue Project, n.d.). JValue is a project with the goal of enabling people to use open data more easily and efficiently. Besides Jayvee, a domain-specific language for building data pipelines, there is also the JValue Hub. The Hub is a web-based management software that allows users to create and collaborate on Jayvee projects.

Open data has become increasingly important in various fields, including research, government, and business. It provides valuable insights and fosters innovation by making data freely available to the public. However, accessing and integrating open data can be challenging due to the diverse formats and sources from which it is available.

Currently, users of the JValue Hub must manually search for datasets on the internet and integrate them into their pipelines. This process involves switching between the JValue Hub and third-party sources, creating a disconnect between the actual work on the data pipelines and the source material. This workflow interruption can be inefficient and cumbersome for users.

What if we could create the possibility for users to access open data from various sources from right inside the JValue Hub? Having the data right at hand would provide great additional value to the project. The goal of this thesis is to evaluate possibilities to find open data sources directly on the JValue Hub instead of relying on external services, and implementing the mechanisms that are necessary for such a feature to work.

In order to implement such a feature, we examine where open data is offered and how meta-data from those locations can be integrated into the JValue Hub. Further, we look for inspiration in existing data providers for enabling users to find open data on the JV Hub. Based on the findings, we design the feature for the JV Hub and, finally, implement it.

In this thesis, we evaluate and implement the following mechanisms:

- A crawler that is able to connect to various open data portals by access-

ing the Resource Description Framework (RDF) metadata catalog of the Comprehensive Knowledge Archive Network (CKAN).

- A component that converts the data representation on the data portals to data sources, the JValue Hub (and Jayvee) internal format.
- An Application Programming Interface (API) providing access to the data sources to make a search on the data sources possible.

The resulting data source search in the JValue Hub allows users to browse open data offerings from various sources based on their search options. With the provided information, they will be able to decide which data sources they want to use in their project. This removes the workflow step of switching to and between open data portals manually and improves the user-friendliness of the JValue Hub. The result also provides the foundations for more advanced functionalities in the future, like linking data sources to Jayvee projects and vice versa, creating projects based on a data source, or suggesting fitting data sources for projects.

# 2 Foundations

## 2.1 Data in Open Data Portals

First, we will examine where significant amounts of data used in Jayvee Projects might be found: Open Data Portals. We will examine how these portals handle providing and displaying datasets and explore how we can build on their approach.

When researching open data portals, one software that immediately stands out is CKAN. It is one of the most widely used platforms for managing data portals. In terms of adoption rate, there are only a few noteworthy alternatives. Socrata is one of them, but the majority of renowned and international data portals use CKAN as their base system.

Therefore, when considering the integration of datasets into the JValue Hub, it is beneficial to start by examining how data is structured and managed within CKAN-based data portals.

### 2.1.1 GovData

GovData is the open data portal of the German government. For this thesis, it will be one of the most important reference points, and ensuring compatibility with it will be a top priority.

When we open the data set overview (Figure 2.1), we see many entries with the headline ‘dataset‘. Each dataset is displayed with a title and a description. Below, there is a section that lists various file types contained within the dataset. To find out more about a dataset, we can click on the title to open it.

This leads us to another screen. The title and description of the data set is still displayed, but additional information is now visible. We see a side panel with information such as keywords, categories etc., and below, we have a list of files. Each file is displayed with its corresponding file type, and when we extend the view by clicking on the arrow next to it, we see a description of the file, a ‘last modified‘-date and some license information.

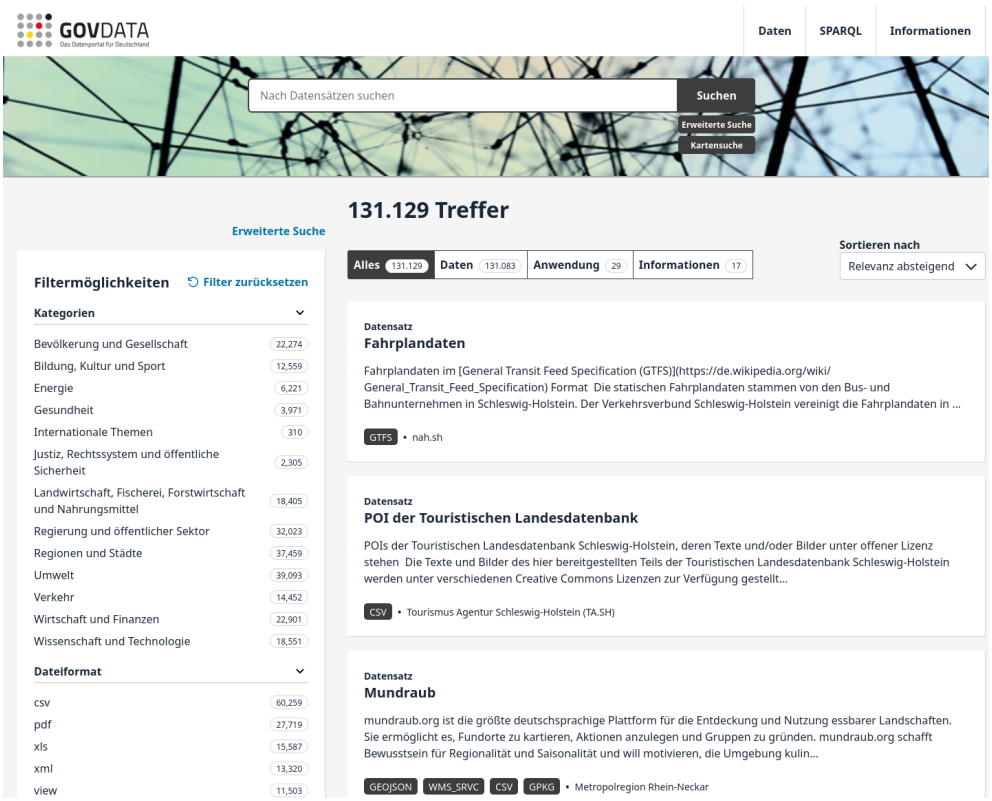


**Figure 2.1:** The data set search page of govdata.

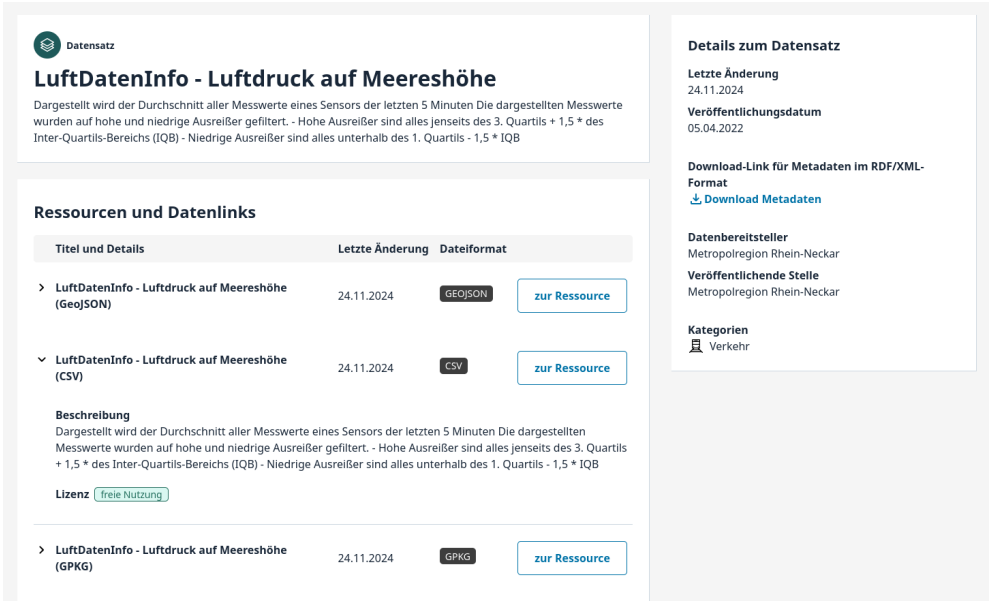


**Figure 2.2:** The data set detail page of govdata.

### 2.1.2 Data.gov

Data.gov is the open data portal of the United States government. It is also CKAN based and might be one of the initial data sources to be used in the JValue Hub. Similar to the overview of GovData, the data sets are listed with their respective title, description, and file type. If we open a data set, we see the same structure as in GovData. The data set is displayed with a title and a description, and below, there is a list of files. The files itself are displayed with an according file type, and we can also view more detailed information about the file by clicking on it.

## 2.2 Conclusion

Understanding the structure and management of data in open data portals is crucial for integrating open data into the JValue Hub. By examining platforms like CKAN and specific portals such as GovData, we can identify best practices and potential challenges. This knowledge will drive the design and implementation of features that enable seamless access to open data within the JValue Hub.

# 3 Requirements

The requirements for the data set functionality in the JV Hub were evaluated in a workshop that was held with some JValue Team members. The attendees besides myself were Georg Schwarz, Johannes Jablonski, Phillip Heltweg and Leonie Färber. The workshop aimed to provide a clear concept of how the members would expect such a feature to work. Some suggestions were defined prior and then evaluated within the group. Also, new ideas and features were brainstormed and added into the list of possible features. The group also discussed the respective importance of the features, and, in some cases, ideas about possible implementation possibilities. In the following section, the outcome of said workshop will be evaluated and portrayed as a collection of requirements. A protocol of the workshop can be found in appendix section A.

The requirements are categorized based on their respective priority. This approach allows us to be more flexible regarding the implementation and efficient use of time. In order to get a solid foundation or in case of unexpected changes, resources can be transferred from less important categories without endangering the value of the work.

## 3.1 Priority Group 1

This priority group describes the goals that are obligatory for functionality of the feature. In the context of this work, they describe a minimal viable thesis.

- **R1.1**: Provide a crawler functionality that allows the import of data set metadata from CKAN-based open data portals like GovData or Mobilithek
- **R1.2**: Create a scheduled update feature for iporting the changes made to the dataset catalogues
- **R1.3**: Persistent logging functionality of the crawler
- **R1.4**: Enable the user to inspect the metadata of data sets
- **R1.5**: Possibility to create projects that are linked to data sets, or link

existing projects to data sets

- **R1.6**: Create an intuitive way to search data sets in the hub
- **R1.7-A**: Additional requirement, added later within the process: Represent data sets as data sources (as they are referenced and used like in Jayvee) and provide a way to convert between the formats.
- **R1.8-A**: Additional requirement, added later within the process: Provide an entry point for future work where solutions for improving data source metadata with AI can be implemented.

## 3.2 Priority Group 2

This priority group describes features or functionality that is not mandatory, but would provide significant additional value. If sufficient time is available in the scope of this thesis, they should be implemented. They may be considered stretch goals.

- **R2.1**: Possibility for hub users to manually add data sets
- **R2.2**: Possibility for hub users to manually update data sets
- **R2.3**: Suggestions of fitting data sets during project creation
- **R2.4**: Ability to browse linked projects for a data set
- **R2.5**: Basic pre-initialization of projects based on the used data sets
- **R2.6**: More advanced search on data sets (search conditions yet to be evaluated, e.g. date, source platform, category, location)

## 3.3 Priority Group 3

This priority group describes features that could provide additional value, but are not necessary.

- **R3.1**: Rank the data sets (based on a to-be-defined metric)

# 4 Architecture and Design

This chapter describes how the components will be designed and integrated into the existing project architecture.

Currently, the JValue Hub project contains the following applications:

- **Hub-Backend**: The general backend application, mostly relevant for database interaction.
- **Hub-Web**: The frontend for the user accessible via a web browser.
- **File-Service**: Used to write and read files from the file system.
- **Pipeline-Service**: Used for handling data pipelines built with Jayvee.

The apps use the NX[1] build system and are containerized with Docker[2].

## 4.1 Overview

Before the actual implementation it should be determined where it is reasonable to place each functionality in the infrastructure of the hub.

When we isolate the main tasks of our code, the following abstract features can be identified:

- API crawling
- Data processing
- Data persisting (Database handling)

Now let's determine where the new added functionalities will find their place inside the code.

[1] https://nx.dev/
[2] https://www.docker.com/

| Data Portal | RDF Catalog | CKAN V3 API | SPARQL |
|---|---|---|---|
| Austria | Y | Y | N |
| Canada | Y | Y | N |
| EU Data Portal | Y | N | Y |
| France | Y | Y | N |
| Italy | Y | Y | Y |
| GovData (Germany) | Y | Y | Y |
| Japan | Y | Y | N |
| Mobilithek | Y | N | N |
| UK | Y | Y | N |

**Table 4.1:** Comparison of supported APIs on various open data portals. The data portals were checked in August 2024.

## 4.2 API crawling

This part deals with the task of retrieving the data from the sources by connecting to their API and then making the data available for further processing.

### 4.2.1 API Selection

When designing the crawler, compatibility with open data portals should be kept in mind. The different portals differ from each other in terms of API access. To find the most suitable API for now, the number of supported data portals should be as high as possible. When focusing on European or American data, the official data portals mostly use an open source software called CKAN. CKAN provides some APIs right out of the box. The most advertised API when reading through the CKAN documentation (CKAN Project, 2023) is the CKAN v3 API. It provides access to the datasets via HTTP requests and the data is being returned in JSON format.

Another option is the use of the SPARQL Protocol And RDF Query Language (SPARQL). The CKAN extension 'SPARQL Interface for CKAN' offers access to the data over simple SPARQL queries.

However, these are not the most-used APIs at the moment. Some data providers, including the German mobility data portal Mobilithek, don't offer support for these APIs. Instead, a very popular choice is to provide access via a RDF (World Wide Web Consortium, 2014) metadata catalog, provided by the CKAN extension `CKAN + DCAT` (CKAN Project, n.d.). Govdata promotes the RDF catalog as a primary way to access the data (Govdata Team, n.d.).

Table 4.1 shows the support of a selected set of open data portals for the different APIs.

These examples show that the RDF Metadata catalog, at this time, is the most popular option for the data portals we want to access. By choosing this variant, we maximize the amount of data portals that are compatible with the crawler.

### 4.2.2 Integration into Existing Infrastructure

Since the world of open data is always moving and changing, importing the catalog from a data portal is quite a time- and resource consuming task. The most reasonable place for this would be the Hub-Backend, but there are a number of reasons against this approach.

- The Hub-Backend has a more service-like functionality. A (possibly on-demand) data import doesn't fit that nature very well.
- If there is an issue with a data import, other components of the backend should not be affected.

Since, out of the already-existing options, the Hub-Backend would have been the most fitting one, we should consider the option of creating a new JValue Hub-app from scratch. This approach allows us to containerize the functionality on its own, which in turn makes it easy to schedule the runs on an infrastructure level.

In terms of functionality, the Crawler will query the data portal APIs and save the obtained data into the database of the hub.

A major difference between the Crawler and the existing hub components is, that it won't act in the background and take requests like a service, but instead runs on-demand in a more procedural manner. This means the crawler is somewhat special in the sense that it introduces a new type of component into the project. The crawler is therefore given the descriptive name **Crawler-Job**.

### 4.2.3 Configuration

The configuration of the Crawler-Job will be handled analogue to the rest of the apps via an `.env` file (and a separate one for the containerized deployment). A new instance of the Crawler-Job will be created and executed for each data portal. This means, the `.env` file will need a configuration parameter where the URL of the CKAN metadata catalog can be passed. For development/convenience purposes there will be another optional parameter allowing the developer to limit the total amount of catalog pages that will be crawled during one cycle.

### 4.2.4 Structure of the data

Before we get started, we need to get a small overview about what we're working with. The open data metadata catalog is provided by a CKAN extension called 'CKAN + DCAT'. It exposes the metadata of the CKAN catalog formatted as RDF/XML, Turtle or JSON-LD, defaulting to RDF/XML, according to the developer documentation (CKAN Project, n.d.). The RDF/XML serialization uses the Data Catalog Vocabulary (DCAT) vocabulary, which is a W3C recommendation for describing data catalogs and is described in its official recommendation (World Wide Web Consortium, 2024). It is a set of classes and properties that are used to describe data catalogs and datasets in a structured way. The DCAT vocabulary is also used in the DCAT-AP adaptation, which is a recommendation for the description of data catalogs in the European Union (AG für GovData, 2022). The DCAT-AP specification is, for example, used by the German government's open data portal GovData. Figure 4.1 shows the essential part of the UML diagram describing the DCAT vocabulary.

As illustrated, the open data representation consists of the main components labeled with `dcat:Dataset` and `dcat:Distribution` keywords. From now on, they will be referenced as 'dataset' and 'distribution'. The distribution is something many people might intuitively consider as a data set (so a resource, e.g. metadata of a Comma-separated values (CSV) file containing specific information about some topic). It also contains license information, a title for the file and many more information. The object referenced as a 'dataset' by DCAT is more like a meta-object that contains multiple Distributions. It also provides an additional title, keywords and a description. In the frontend, many CKAN-based data portals handle Datasets more like 'folders' that contain the actual data (in form of distributions).

## 4.3 Datasource Middleware

This part deals with the task of processing the crawled data and transform it into a representation more suitable for the use inside the JValue hub.

### 4.3.1 Integration into Existing Infrastructure

We now know that the crawled data can flow in at any time a crawler run is scheduled. Further, the time needed to process a new data set / distribution may be longer than the rate in which they come in. Although this will be not a major focus of this thesis, we want to consider that in the future there might be added more sophisticated solutions for processing the incoming data. Faulty metadata could be repaired, the data itself could be inspected and metadata could be extrapolated from it. In short, we should accept that the processing of

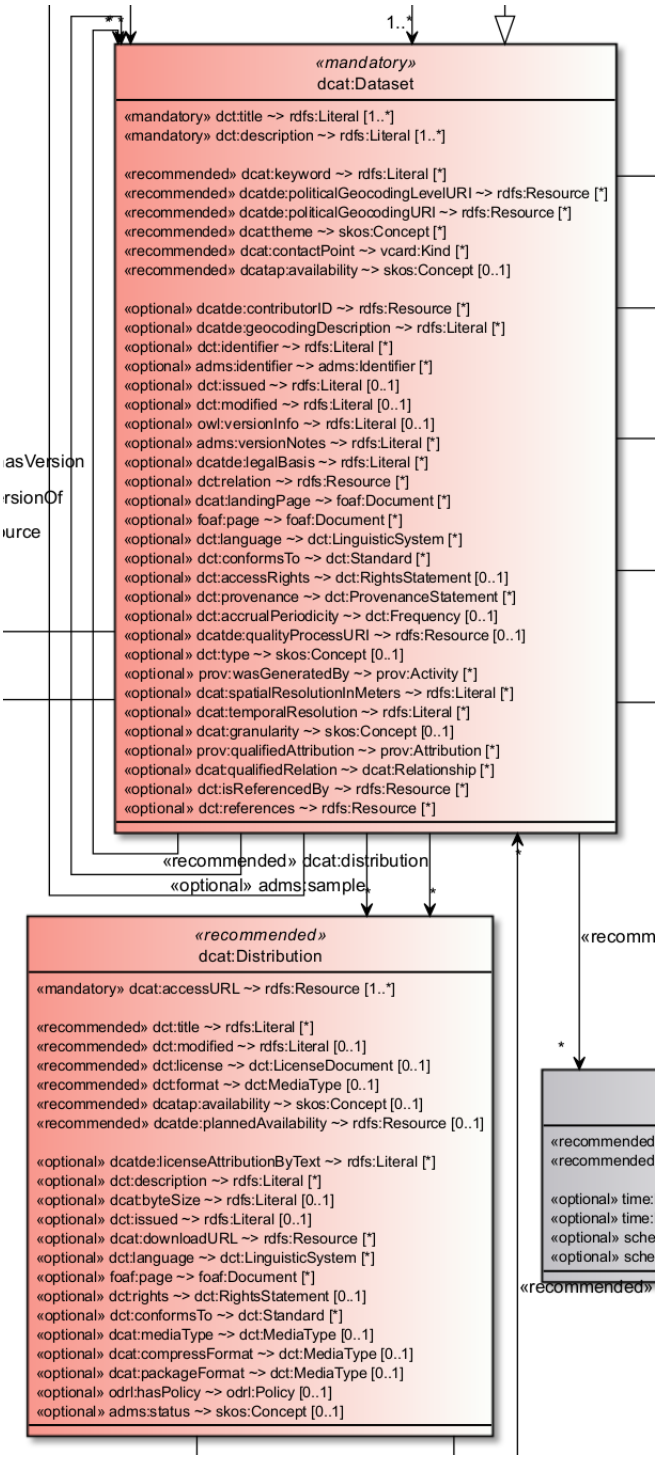


**Figure 4.1:** DCAT RDF catalog specification (AG für GovData, 2022)

the data will likely have to be done in an asynchronous way to allow us to be more flexible in scaling the application in the future.

This also means that the Hub-Backend won't be the best place for this task, either. We don't want the import and data processing to interfere with the basic functionalities of the backend, and the functionality of the Hub, in general. So this feature also will be separated into its own app.

In conclusion, we will create another NestJS-App for processing open data metadata that will operate more like a service in the background. This is because it will wait for Crawler-Job-instances to spawn and then process the resulting data. From now on, this app will be referenced as the 'Datasource Generator'.

### 4.3.2 Configuration

For the configuration, the Datasource Generator needs configuration for access to both the Distribution-Queue and the Data Source-Queue. The configuration will be handled via the `.env` file, as usual, and the app will get passed the queue names as parameters.

## 4.4 Data persisting

This part deals with the task of saving the (now ready-to-use) data to the database and providing the necessary API to the frontend, so it can be used.

### 4.4.1 Integration into Existing Infrastructure

This time, the Hub-Backend comes to mind as the natural choice. It already provides the Database handling functionality for the majority of the currently existing features. It also provides access to the saved data by exposing Representational State Transfer (REST)-API endpoints to the frontend. Creating an additional app here would create unnecessary redundancies, so we will integrate this task into the Hub-Backend app.

### 4.4.2 Configuration

The Hub-Backend already has an .env file for configuration. Most of the configurations also apply for the yet-to-be implemented features, but there will be some additional parameters for being able to communicate with the Crawler-Job and Datasource-Generator.

## 4.5 Summary

In short and as shown in Figure 4.2, we will add two additional apps to the hub: `crawler-job`, running on-demand and being a new category in the hub, and `datasource-generator`, that runs in the background like most of the other apps.

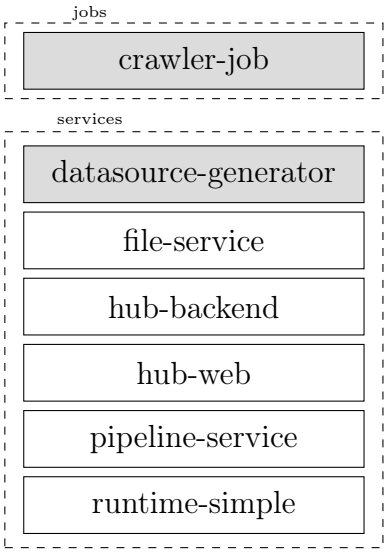


**Figure 4.2:** Diagram showing the extension of the JValue-Hub app system.

# 5 Implementation

This chapter details the implementation of the data crawler and its integration into the JValue Hub. The implementation covers various aspects, including data representation, parsing, crawling algorithms, database interactions, and containerization.

## 5.1 Crawler

The crawler is the first component that has to be implemented. It is responsible for downloading the metadata catalogs from the open data portals, parsing the data, and saving it into the database. The crawler is a standalone app that runs independently of the JValue Hub. It is implemented as a NestJS app, which is a Node.js framework for building server-side applications.

### 5.1.1 Internal Data Representation

Since the crawler itself won't be responsible for generating the data source representation of the open data, the internal representation will closely resemble the source data it works with.

Essentially, we pick the most important information documented in the DCAT-AP-DE specification, add some more useful information for working with the objects and get these interfaces as a result:

From here on, objects that comply to these interfaces, are referenced as datasets and distributions. The main differences (besides the removed parameters) are:

- `id`: Since we want our own versioning system, we cannot solely rely on the IDs that come with the source data. This parameter is the hub-internal UUID that will also be the primary key to be referenced in the database later. Regarding the interfaces, only the Dataset needs the ID field at this stage. More clarifications on this behalf follow in section 5.1.4.
- `crawledDate`: In addition to the already-provided creation and last mod-

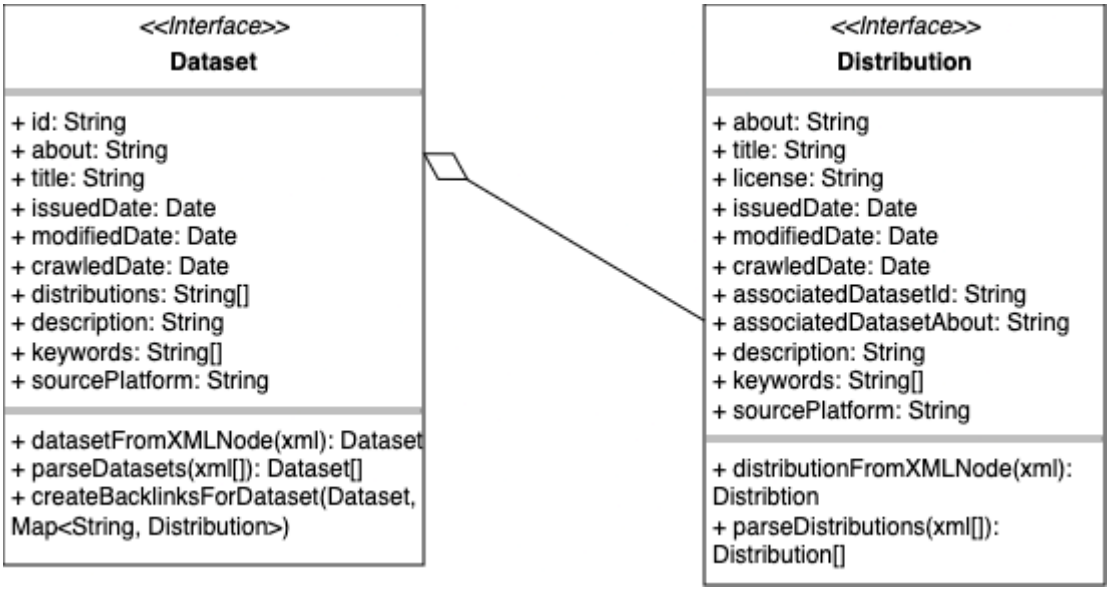


**Figure 5.1:** XML diagram of Dataset and Distribution interfaces

ified date fields, we add a `crawledDate` field to indicate when this entry was found on the source platform. This field is important for the crawler to determine if the entry has changed since the last crawl.

Additionally, we create some functions that provide the creating and parsing functionality:

- `datasetFromXMLNode()`: Takes a raw dataset from the Extensible Markup Language (XML) data, turns it into a Dataset object and returns it.
- `parseDatasets()`: Takes multiple raw dataset representations and turns them all into Dataset objects.
- `createBacklinksForDataset()`: For the purpose of finding the corresponding Dataset for a Distribution more easily, this function sets the `associatedDatasetAbout` field for every distribution connected to it.
- `distributionFromXMLNode()`: Takes a raw distribution from the XML data, turns it into a Distribution object and returns it.
- `parseDistributions()`: Takes multiple raw distribution representations and converts them all into Dataset objects by calling `distributionFromXMLNode()`. The results are being returned as an array.

### 5.1.2 RDF-XML parsing

As discussed in chapter 4, the RDF and RDF-XML serialization specs predetermine the way we have to interpret the provided metadata catalog. If we use the DCAT vocabulary for filtering the correct nodes, we can use a simple XML parser to extract the desired information. Listing 5.1 shows a simplified outline of how the data is represented in the GovData metadata catalog.

```
<rdf:RDF>
    <dcat:Dataset rdf:about={DATASET-IDENTIFIER}>
        <dct:title>{DATASET-TITLE}</dct:title>
        <dct:description>{DATASET-DESCRIPTION}</dct:description>
        <dcat:keyword>{KEYWORD 1}</dcat:keyword>
        <dcat:keyword>{KEYWORD 2}</dcat:keyword>
        <dct:issued rdf:datatype="date">{CREATION-DATE}</dct:issued
            >
        <dct:modified rdf:datatype="date">{MODIFIED-DATE}</
            dct:modified>
        <dcat:distribution rdf:resource={DISTRIBUTION-IDENTIFIER}/>
    </dcat:Dataset>

    <dcat:Distribution rdf:about={DISTRIBUTION-IDENTIFIER}>
        <dct:title>{DATASET-TITLE}</dct:title>
        <dct:description>{DISTRIBUTION-DESCRIPTION}<dct:description>
        <dct:license rdf:resource={DISTRIBUTION-LICENSE}/>
        <dcat:accessURL rdf:resource={DISTRIBUTION-URL}/>
        <dcat:downloadURL rdf:resource={DISTRIBUTION-DOWNLOAD-URL}/>
        <dct:format rdf:resource={DISTRIBUTION-FORMAT}/>
        <dct:issued rdf:datatype="date">{CREATION-DATE}<dct:issued>
        <dct:modified rdf:datatype="date">{MODIFIED-DATE}<
            dct:modified>
    </dcat:Distribution>
</rdf:RDF>
```

**Listing 5.1:** A simplified example of the XML-formatted source data

Essentially, we can link the distributions and datasets to each other by using the 'rdf:about' field specified in the RDF standard since it contains the Unique Resource Identifier (URI) information for the referenced nodes. The URI of the distribution (`DISTRIBUTION-IDENTIFIER` in the example) is accessible in the `rdf:resource` parameter inside the `distribution` XML node.

To handle the catalog crawling, we also need a way to iterate over the different catalog pages. Listing 5.2 shows which XML nodes the DCAT RDF/XML catalog uses to express this information about itself by using the `Hydra` vocabulary.

This information will also be parsed and put into an object implementing a newly created `catalogMetadata` interface:

```
<hydra:PagedCollection rdf:about={LINK-THIS-PAGE}>
    <hydra:nextPage>{LINK-NEXT-PAGE}</hydra:nextPage>
```

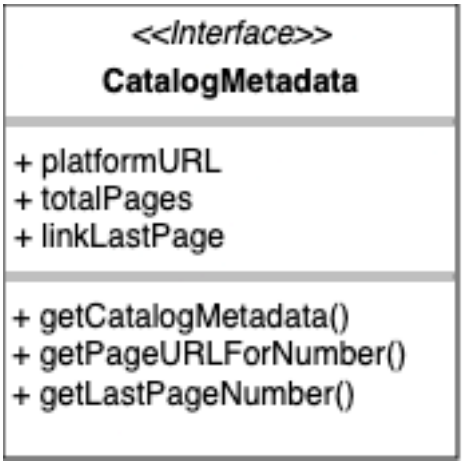


**Figure 5.2:** An UML diagram for the CatalogMetadata interface and its corresponding methods

```
    <hydra:firstPage>{LINK-FIRST-PAGE}</hydra:firstPage>
    <hydra:lastPage>{LINK-LAST-PAGE}</hydra:lastPage>
    <hydra:totalItems rdf:datatype={INT}>{NUMBER-OF-ITEMS}</...>
    <hydra:itemsPerPage rdf:datatype={INT}>{ITEMS-PER-PAGE}</...>
</hydra:PagedCollection>
```

**Listing 5.2:** Example of a DCAT RDF/XML catalog metadata

- `parseXMLStr(body: string)`: Uses the package `xml-js` [1] to parse the downloaded XML file into a Typescript object representation.
- `tryGetRDFBodyFromURL(url, logger, maxRetries: number){}`: Wraps `getRDFBodyFromURL()` to handle failures on `fetch` and retry a pre-defined number of times.
- `findNodesByName()`: Finds all XML nodes that have the provided name, e.g. `<dcat:distribution>`.
- `getXMLContent(xmlElement: Element): string | undefined {}`: Takes an XML element and extracts the content. If the specified element is not found in the object, then it tries to get it from typical XML attributes used in DCAT.
- `getDateFromXMLElement(xmlElement: Element): Date | undefined {}`: Uses `getXMLContent()` to get a date object contained in a DCAT RDF/XML date node.

[1] https://www.npmjs.com/package/xml-js


```
export enum XMLStrings {
    Distribution = 'dcat:Distribution',
    distribution = 'dcat:distribution',
    Dataset = 'dcat:Dataset',
    downloadURL = 'dcat:downloadURL',
    accessURL = 'dcat:accessURL',
    resource = 'rdf:resource',
    description = 'dct:description',
    title = 'dct:title',
    about = 'rdf:about',
    format = 'dct:format',
    license = 'dct:license',
    hydra = 'hydra:PagedCollection',
    hydraLast = 'hydra:lastPage',
    modified = 'dct:modified',
    issued = 'dct:issued',
    datatype = 'rdf:datatype',
    keyword = 'dcat:keyword',
}
```

**Listing 5.3:** Implementation for DCAT-DE-AP

**Figure 5.3:** XMLStrings interface definition and Typescript implementation

However, there is a caveat to consider: the DCAT-Vocabularies may vary from one Data Portal to another. To handle that possibility, the vocabulary is being inserted by using an enumeration that can be easily replaced with another one that implements another DCAT variant. For a case like that, typescript provides a string-enum feature. Figure 5.3 shows the general enum layout and the implementation for the DCAT-DE-AP vocabulary.

### 5.1.3 Crawling algorithm

This section provides a detailed overview of how the crawling algorithm is implemented. This is a list of the functions used to implement the algorithm.

- `crawl(catalogMetadata: CatalogMetadata){}`: higher-level function that crawls all pages of the provided RDF/XML catalog metadata. Saves the results in the internal state of `crawler-job`.
- `async function getRDFBodyFromURL(url: string, logger: Logger): Promise<string | undefined> {}`: uses `node-fetch` to get the `RDF/XML` file from the provided URL.

- `saveDistributions(){}`: saves the crawled distributions to the database.
- `saveDatasets(){}`: saves the crawled datasets into the database.
- `appendDistributions(){}`: adds a parsed distribution to the `distributions`.
- `appendDatasets(){}`: adds a parsed distribution to the `datasets`.
- `sendDistributionOverQueue(){}`: sends a distribution object over the queue to the `datasource-generator`.

On instantiation of a new `crawler-job`, the app reads the URL of the desired metadata catalog from an environment variable. It uses `node-fetch` to download the file from the webserver and creates a new `CatalogMetadata` object. This object is used to slowly iterate over the catalog and download every remaining page. On every iteration, the catalog page is being parsed into a typescript object by using the `parseXMLStr()` function. The `findNodesByName()` function is then used to get all the `Dataset` and `Distribution` nodes from the XML file. Afterwards, the `parseDatasets()` and `parseDistributions()` functions are then used to turn the raw data into Dataset and Distribution objects. The objects are then being saved to the internal state of the `crawler-job` by using the `appendDatasets()` and `appendDistributions()` functions. When the full catalog is processed, the `saveDatasets()` and `saveDistributions()` functions are being called to persist the data into the database. If this step was successful, the `sendDistributionOverQueue()` function is being called to send the distribution objects to the `datasource-generator`.

### 5.1.4 Database

Although it is not planned to show the raw crawled data directly to the user, it will be persisted in the database nonetheless. The Crawler, just like Hub-Backend, uses TypeORM[2] for communication with the underlying Postgres[3] database. It's crucial that other apps, especially the backend, are also able to access the same tables, so the database service class is created within the shared library folder and then being used by importing the NestJS module. The service class mainly consists of two methods:

- `saveDataset()`: Saves a dataset object to the postgres database and returns the entity on success. On error, undefined is returned.
- `saveDistribution()`saveDistribution(): Saves a distribution object to the postgres database and returns the entity on success. On error, undefined is returned.

[2]https://typeorm.io/
[3]https://www.postgresql.org/

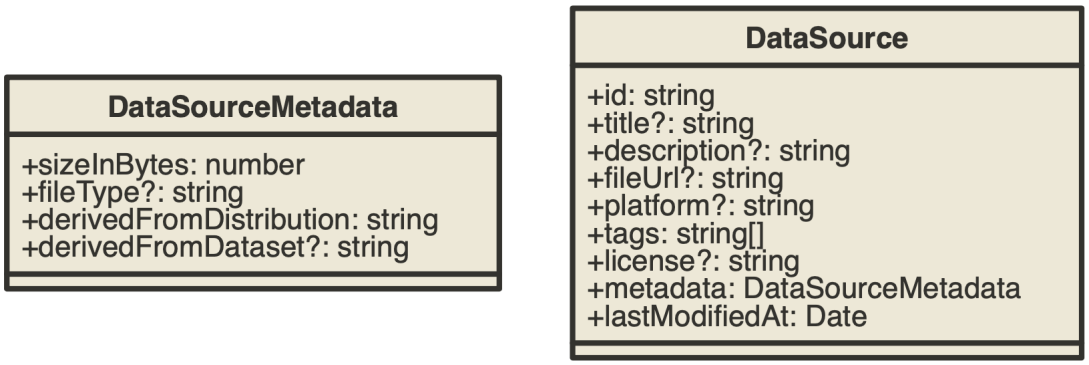


**Figure 5.4:** A XML diagram of DataSource and DataSourceMetadata classes

Both save functions check, if the object already exists in the database. 'Already exists' in this case means that both of those conditions apply:

- An object with the same `about` field already exists.
- The object also has the same value inside the `modifiedDate` field.

The conditions ensure that only new versions of datasets and distributions will be inserted into the database. A Postgres SQL constraint verifies the conditions are met.

There are two tables, one for the datasets and one for the distributions. There is a foreign key relation that maps multiple distributions to one dataset. The primary keys are generated at insertion time, because the 'about' fields, despite being URIs, are not unique due to the possibility of having duplicates when entries are changed or updated.

## 5.2 Datasource Generator

The `datasource-generator` is the second component that has to be implemented. It is responsible for generating the data source objects from the Distributions provided by the Crawler. The `datasource-generator` is also implemented as a NestJS app.

### 5.2.1 Mapping of the Crawled Data to Data Sources

Before we can talk about creating data sources, we need to clarify how data sources are supposed to look like. Figure 5.4 portrays how the data sources are defined.

We quickly recognize that the fields of the data source class are mainly a mix

of the dataset and distribution classes. A few details were put outside the class in another class called `DataSourceMetadata`. Information about the underlying file is stored there as well as the Distribution and Dataset the data source was created from.

Now we will create some functions that help us creating DataSource and DataSourceMetadata objects from the Distribution objects provided by the CrawlerJob:

- `createDataSourceFromDistribution()`: This function, as the title suggests, takes a Distribution object and returns a promise for a DataSource object. The `lastModifiedAt`-timestamp is being set to the current date. The following functions are called from within this function in order to set the leftover fields accordingly.
- `generateDataSourceTitle()`: This function gets the source distribution object and returns a fitting title for the new data source.
- `generateDataSourceDescription()`: like the previous function, but for creating the description.
- `generateDataSourceMetadata()`: Generates a metadata object for the data source. It sets the `fileFormat`, `derivedFromDistribution` and `derivedFromDataset` fields, and sets `sizeInBytes` by calling the following subroutine:
- `getFileSizeInBytes()`: Uses the provided fileUrl to get the file size for the current data source. An HTTP GET request is being sent to the web server, and the returned 'content-length'-header is being used to retrieve the size of the actual file. This way, we can speed up the process and get the information without downloading every data file, at the cost of possible cases where the server does not provide the information.
- `getCatalogMetadata(xml: Element[])`: Takes a parsed XML file, and creates a `CatalogMetadata` object from it.

### 5.2.2 DistributionQueue and DataSourceQueue

#### Setup

We already established that the `datasource-generator` won't have access to the underlying database and will communicate with the other apps asynchronously by using queues. For this functionality, we need the following components: First, we need two NestJS queue services, each handling its own queue. They will be called `DataSourceQueueService` and `DistributionQueueService`. The `DataSourceQueueService` will be used to send the generated data sources to the

Hub-Backend, while the `DistributionQueueService` will be used to receive the distributions from the Crawler-Job.

The app listens passively on the `DistributionQueueService`. This is being achieved by registering the service class in the NestJS `AppModule` as a provider. RabbitMQ[4] now listens for incoming objects in the registered Queue.

When a Distribution object is being received, it shall be passed from the `DistributionQueueService` to the `DataSourceQueueService`. This can be achieved by creating a function `handleNewDistribution()` which calls `createDataSourceFromDistribution()` on the incoming object and sends the return value by using the `sendMessage()` function OF THE `DataSourceQueueService`.

Now we can use the `onApplicationBootstrap()` function included in NestJS controllers and pass a routine called `onNewDistribution()`, which takes `handleNewDistribution` as a callback, parses the incoming queue object and then invokes `handleNewDistribution` on the parsed object.

#### Object validation

For passing the distributions and data sources through their respective queues, we need classes to instantiate them as objects and parse/validate them. Since the handed objects might slightly vary from the entity objects that will be used to save them into the database, we create the classes `NewDistributionEvent` and `DataSourceEvent`. For RabbitMQ, to parse the objects correctly, we need to add decorators from the `class-transformer` and `class-validator` libraries to make sure every passed object was correctly sent. Every attribute undergoes its corresponding type checker and optional attributes are marked as such. More complex attributes like `DataSourceMetadata` have to undergo nested validation.

## 5.3 Hub-Backend

The main role of the backend in this setup is to serve as receiver for data source objects from the generator, saving them to the database and exposing them over the REST API, so they can be accessed from within the frontend.

### 5.3.1 Database Handling

#### Data model

The current database setup in the backend is TypeORM based. This approach can be easily applied to data sources. A new relation is added to the database.

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

Listing 5.4 is the table scheme generated by TypeORM.

```
CREATE TABLE public.data_source_entity (
    id uuid DEFAULT public.uuid_generate_v4() NOT NULL,
    title character varying NOT NULL,
    description character varying NOT NULL,
    "fileUrl" character varying,
    platform character varying,
    tags text[] DEFAULT '{}'::text[] NOT NULL,
    license character varying,
    metadata text NOT NULL,
    "lastModifiedAt" timestamp without time zone NOT NULL
);
```

**Listing 5.4:** SQL schema of the data source table

The metadata column is of type text because TypeORM uses a JavaScript Object Notation (JSON)-parsing mechanism to handle the object conversion internally. We have to provide a so-called transformer class to handle the conversion. It consists of two methods:

- `to(value: DataSourceMetadata): string`

  Parses the `DataSourceMetadata` object to a JSON string.

- `from(value: string): DataSourceMetadata`

  Reverse operation of `to()`. Takes a JSON representation and parses it to a `DataSourceMetadata` object.

### Database service

TypeORM also needs a service class that provides the insertion/retrieving operations. It contains the following methods:

- `async saveDataSourceFromEvent(dataSource: DataSourceEvent)`

  Takes a data source that is provided by the crawler over the queue, creates an DataSourceEntity instance and saves it into the database.

- `async saveDataSource(dataSource: DataSource)`

  Takes a data source that comes from another source (most likely the frontend) and saves it into the database. Data sources provided this way may have a slightly different structure than the generated ones, hence this separate method.

- `async findById(id: string): Promise<DataSourceEntity[]>`

  Returns the data source entity with the given ID, if it exists.

- `async findAll(paginationOptions: PaginationOptions):`

  `Promise<PageDto<string>>`

  Returns all data sources wrapped for use in a paginated overview.

### 5.3.2 REST API

The now established service classes are exposed over a REST API implemented in the data source controller.

- `GET` and `POST` requests on `/projects/data-source` are being used to save and delete connections between projects and data sources.
- `GET` on `data-sources` returns all data sources in a paginated way.
- `GET` on `data-sources/{ID}` returns a single data source by its ID.

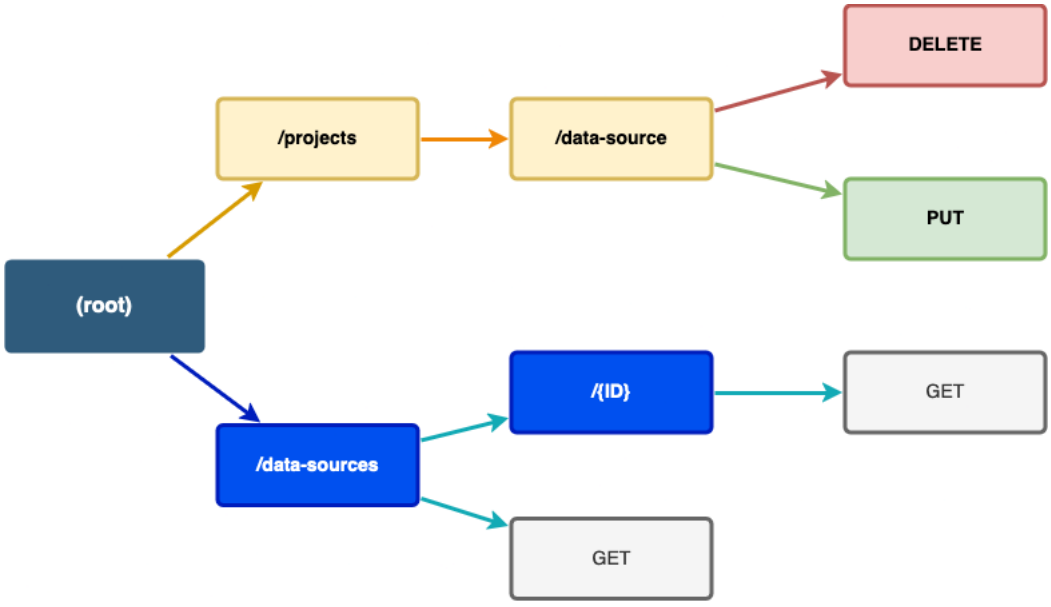


**Figure 5.5:** Diagram showing the data source related REST API endpoints.

## 5.4 Shared Libraries

However, despite the added app logic being mostly independent of the rest of the hub, other components might have to access some if its interfaces, e.g. the Dataset and Distribution interfaces or database entities. This means that a place outside the apps themselves is needed, where we can define these structures and reuse them on multiple places of the hub. Luckily, there already exists such a

| App Name | Logging identifier |
|---|---|
| Crawler | `crawler-job` |
| Datasource Generator | `datasource-generator` |

**Table 5.1:** Logging identifiers for the new apps.

place. We can create a NestJS-module, and put it in the NestJS-shared-library of the hub. Then, other apps can access the resources by importing the module.

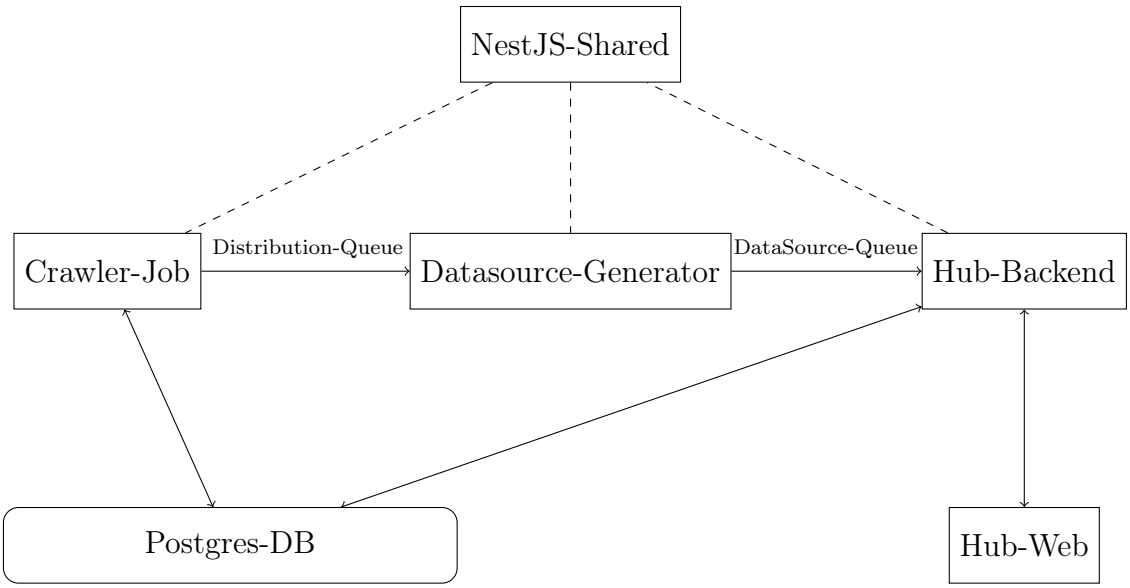


**Figure 5.6:** A graph visualizing the interaction between components.

## 5.5 Logging

For all the newly introduced modules, the already-established logger class of the JValue Hub is used. It is implemented by extending the `LoggerService` interface of NestJS. When a logger is created, a string has to be provided that specifies the category to be displayed within the log message. Each log message includes the name of the originating app to ease tracing back issues to their source.

We will differentiate between two types of logging: Progress logging and error logging. The progress logging is used to show the current status of the app, while the error logging is used to show any errors that occur during the runtime of the app.

### 5.5.1 Progress Logging

When running Crawler-Job, the process may need some time to finish. Especially in cases of data portals with lots of datasets, we want to have some progress logging in order to visualize the current status.

When designing the logging, we have to consider ways of making the logs as informative as possible without flooding the log files. Since we work with potentially big amounts of data, we have to make sure that the logs are not too verbose. The logs are most important to the people maintaining the system, so we have to make sure that they can easily see the current status of the app and only get the most important information.

This is achieved by logging the following events:

- A new catalog page has been downloaded successfully from the portal: `Page {$i} downloaded.`
- A new Distribution/Dataset is being parsed/processed:
  `processing Dataset/Distribution ${i} of ${total}`.
  Since there might be large amounts of data being parsed, the logger only sends this message when hitting a configurable interval, e.g. every 1000 datasets. This makes sure the crawler sends its heartbeats as frequently as needed without flooding the log files in the process.
- The crawler job run has ended:

  ```
  Summary: ${this.stats.crawledDatasets} datasets were
  crawled, ${this.stats.newOrUpdatedDatasets} new or updated ones
  were saved.
  Summary: ${this.stats.crawledDistributions} distributions were
  crawled, ${this.stats.newOrUpdatedDistributions} new or updated
  ones were saved.
  ```

  This overview is crucial for the maintainer to see how many datasets and distributions were processed and saved, and also gives indications if something might be wrong or misconfigured. The summary log also contains some error related information, which are explained in the following section.

### 5.5.2 Error Logging

There might be some errors when processing the metadata catalogs. Those cases also have to be covered by the logging functionality.

- Problems when downloading catalog pages:

The downloads are wrapped in the function `tryGetRDFBodyFromURL()`. On an unknown error, the message is logged, and the download is retried combined with a growing cooldown time until a predefined number of retries is reached. It returns a `fetchResult` object which contains the downloaded page, or undefined on faults and the amount of retries needed at the attempt. In a crawler run, the total download retries are counted and used in the final statistics log.

- The page URL for the next catalog page could not be found: `Could not find next catalog page`.
- The crawler job run has ended:

  ```
  ${this.stats.retries} download retries were needed,
  and ${this.stats.pagesSkipped} catalog pages had to be
  skipped.
  ```

  This message is included in the summary log at the end of each run.

## 5.6 Containerization

Now that the new apps are implemented, we need to ensure the deployment of them is easy, reliable and compliant with the current project standards.

For deployment, the JValue Hub uses `docker compose` to build the apps as containers. Ultimately, they will be deployed in a Kubernetes environment, but for this thesis, it is only necessary to create Docker containers that are ready to be deployed.

### 5.6.1 datasource-generator

`datasource-generator` will get an entry in the already existing compose file `docker-compose.yml`. The build depends on the successful build of `runtime-queue`, since it cannot operate without the queue container running. The docker image is named `datasource-generator` and the build will be based on a new Dockerfile called `datasource-generator.Dockerfile`.

To optimize the image size, we will implement a two-step build process. One step for building the app, and one step for copying the transpiled code into a minimal runner image. The Dockerfile uses the `node:lts` docker image as a base. It will serve as the build image. The `package-lock.json` is copied inside the container as well as the `src` folder. The dependencies are installed, and the project is built using `nx`. After a successful build, the `dist` folder is copied

into a minimal `node:lts-alpine` image alongside with necessary dependencies. Finally, the JavaScript main file can be started.

### 5.6.2 crawler-job

When typing `docker compose {build/up}` into the terminal in order to build/run the JValue Hub apps, we don't want a `crawler-job` job to build/run every time. For separation, we create a second compose file called `jobs.docker-compose.yml`. It contains a single entry. The container will be called `crawler-job`, it depends on `datasource-generator` and `postgres-db`. If a developer wants to build or run the container, he has to type in the command

`docker compose -f docker-compose.yml -f jobs.docker-compose.yml up crawler-job`.

The Dockerfile will be called `crawler-job.Dockerfile`. The build process, for the most part, is quite similar to the one of `datasource-generator`. The main difference is that the crawler needs a manual assignment of higher Random Access Memory (RAM) limits by using the `--max-old-space-size` option. The node process in the container has a standard limit, which, in deployment, was agnostic of the actual RAM size that was allocated. Since we process the open data portals in-memory, we give a generous limit of 10GB, which is more than sufficient when crawling the GovData portal.

# 6 Evaluation

In this chapter, we will evaluate whether and how the requirements and goals defined in chapter 3 were fulfilled. Since at the creation time of these requirements, the data source mapping was not yet established, the term "data sets" now refers to data sources.

For a better overview, table 6.1 categorizes the requirements. We will differentiate between the requirements that have been met (green), those that have been partially met (yellow) and those that were not met (red). The following sections will elaborate each one in more detail.

## 6.1 Additional Requirements

First, we will look at important requirements that will be relevant at multiple points of the evaluation. They were introduced later in the process, taking away some focus from the original requirements.

- **R1.7-A: Represent Data Sets as Data Sources** was fulfilled by implementing the `datasource-generator`.
- **R1.8-A: Entry Point for Metadata generation** was fulfilled by designing `datasource-generator` as a separate app that works asynchronously

| Priority Group 1 | Priority Group 2 | Priority Group 3 |
|---|---|---|
| R1.1 | R2.1 | R3.1 |
| R1.2 | R2.2 | |
| R1.3 | R2.3 | |
| R1.4 | R2.4 | |
| R1.5 | R2.5 | |
| R1.6 | R2.5 | |
| R1.7-A | | |
| R1.8-A | | |

**Table 6.1:** Overview of the requirements and their status

and independent of `crawler-job`. This way, `datasource-generator` can act as an instance that can generate enhanced metadata for data sources that were crawled by `crawler-job` using more sophisticated, such as machine learning-based techniques.

Throughout the project, other developers also advanced the project. Among other features, a design system was created for a more streamlined way of creating and maintaining UI components. Since the initial requirements for this work also targeted work on UI components, we had to weigh adding new components with the conflicts it might cause arising through the introduction of the design system. In the end, we decided to shift focus on backend features in favor of producing merge conflicts on the UI part. Thus, we agreed on an interface in the backend that future UI features and components can be programmed against.

## 6.2 Priority Group 1

- **R1.1: Crawler Functionality** was fulfilled. The `crawler-job` app crawls and imports data set metadata from open data portals based on CKAN by accessing the commonly used RDF-XML metadata catalog.
- **R1.2: Scheduled Update Feature** was fulfilled. The fundamental concept of the `crawler-job` app allows simple scheduling on an infrastructure level (by scheduling runs of the container). On multiple subsequent runs, the crawler handles new or changed data sets accordingly, and therefore provide a full data update functionality.
- **R1.3: Persisted Logging** was fulfilled. The implemented features use the JValue Hub internal loggers and create meaningful messages in order to provide a good insight.
- **R1.4: Metadata Inspection** was partially fulfilled. On implementation of the API that provides the data source metadata to the frontend, it was ensured that the data complies to the frontend concept that was created by the frontend developers. The actual integration of the components fell out of the scope of this thesis, as explained in section 6.1.
- **R1.5: Dataset / Project Linking** was fulfilled. The many-to-many relation between `projectEntity` and `dataSourceEntity` suffices to track connections between the two tables. The API endpoints and database service functions provide simple access to the relationships and allow easy linking and unlinking of data sources and Projects. The data source info card in the frontend will also display the linked projects. **Figure 6.1** demonstrates a way of displaying the reference count of a data source in the JValue Hub.
- **R1.6: Data Set Search** was partially fulfilled. The search in the JValue


**Figure 6.1:** A data source info card in the JValue Hub displaying some crawled data. The frontend was not implemented as a part of this thesis, but uses the API endpoints created in this work as a foundation.

Hub got a big redo by the JValue Hub Team, so the search functionality already exists. The API endpoints created in this work aim to comply to the new search in the hub.

## 6.3 Priority Group 2 and 3

- **R2.1: Manual Adding of Data Sets** and **R2.2: Manual uUdating of Data Sets** were fulfilled partially. Besides the automated import, `hub-backend` also provides a service function that saves potential manually created/edited data sources. Since building the frontend fell out of the scope of this thesis, the action cannot be done by the end user at the current time.
- **R2.3: Suggestions of Fitting Data Sets** and **R2.5: Basic Pre-Initialization of Projects**, as they are frontend-focused features, they also fell out of the scope of this thesis as described in section 7.1.
- **R2.4: Ability to Browse Linked Projects for a Data Set** was partially fulfilled. The implementation of **R1.5** also provides the option to easily access the corresponding projects for a data set and vice versa. For example, when retrieving a `projectEntity` from the database, the resulting object contains a `dataSources`-Array that contains all referenced data sources.

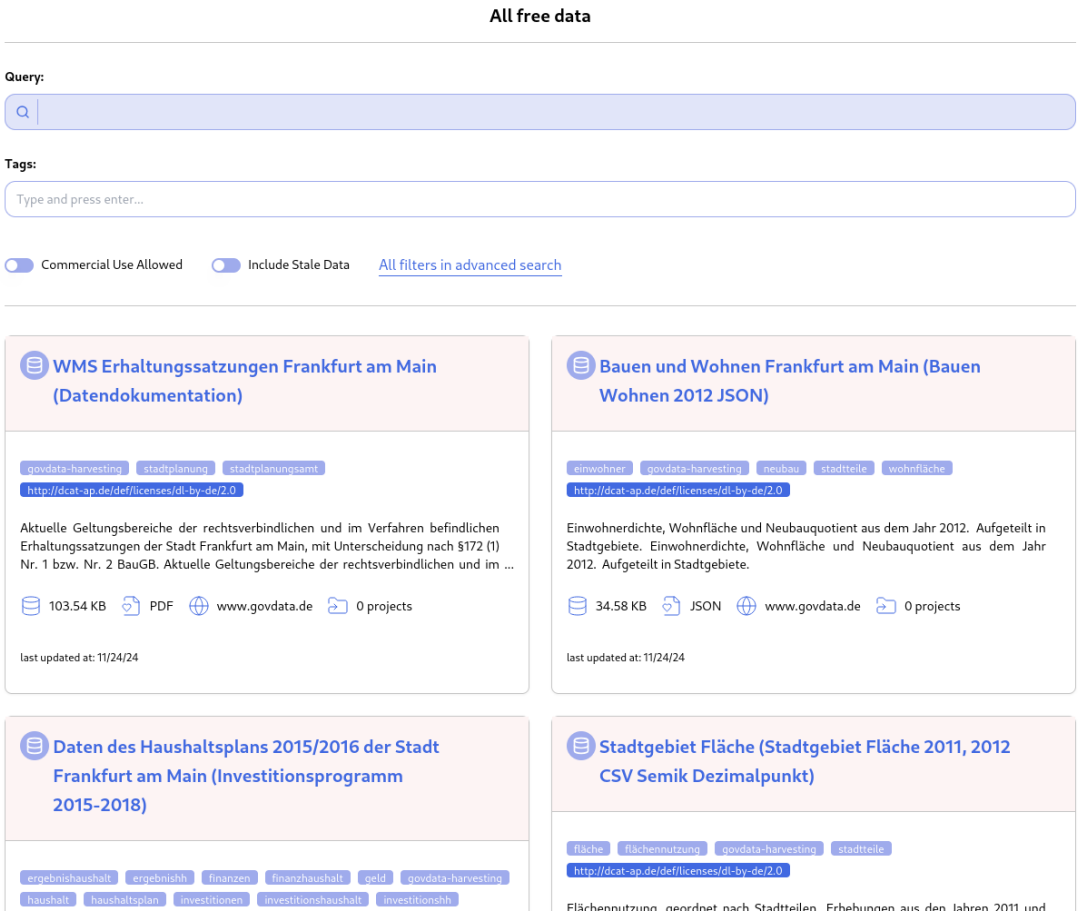


**Figure 6.2:** The new topic search in the JValue Hub displaying some crawled data. The frontend was not implemented as a part of this thesis, but uses the API endpoints created in this work as a foundation.

- **R2.6: More Advanced Search on Data Sets** also was partially fulfilled in the same sense as
- R1.6. The new topic search provides the advanced search functionality that is aimed for. The Backend provides an Interface that will make this search usable with the data sources.
- **R3.1: Data Set Ranking** was not fulfilled. Since this primarily was a brainstorming idea, the already mentioned priority shift meant that this idea could not be implemented at this point in time.

# 7 Future Work

This section elaborates on how we could improve the feature based on the foundation that was created within this work.

## 7.1 Remaining Requirements

First, we will look at the requirements that haven't been implemented yet:

- **R2.3: Dataset Suggestions for Projects** - This could be implemented by using a keyword search. By searching for the project name components inside the data source table, relevant data sources could be retrieved and displayed in a list.
- **R3.1: Data Source Ranking** - This could be realized by counting the number of references to Jayvee projects for every data source. This way, we can rank them by popularity. Additionally, a like/dislike system could be added to enable users to influence the rating.

## 7.2 Additional Ideas

There are many potential enhancements for the JValue Hub data source functionality. Here are some ideas that could be implemented:

- **Generating enhanced data source metadata** - Open data often comes with the caveat that it may be auto-generated or poorly maintained. To provide the best possible experience to the user, we could attempt to enhance the metadata in the data source generation step. The `datasource-generator` could be extended to use machine learning algorithms to enhance the metadata of the data sources. This could be achieved by analyzing the metadata content, comparing data sources to one another, and extracting additional information such as data quality, relevance to a specific topic, or popularity.

- **Data source linking to Jayvee projects** - The data source linking to Jayvee projects could be improved by adding a feature that allows users to link a data source to a Jayvee project directly from the data source info card. This would enable users to create a project based on a data source or to link a data source to an existing project with just a few clicks.
- **Data source suggestions for projects** - The data source suggestions for projects could be enhanced by adding a feature that suggests relevant data sources for a project based on the project's name or description. This could be achieved by analyzing the project's content and comparing it to the data sources in the JValue Hub. The user could then choose from a list of suggested data sources and link them to the project.
- **Support for additional APIs** - The JValue Hub could be extended to support additional APIs for importing data sources from other open data portals. This would enable users to import data sources from a wider range of sources and to access more data for their projects.

# 8 Conclusion

The main purpose of this work was to conceptualize and implement a method for using open data inside the JValue Hub. We examined the most efficient way to ensure compatibility with as many open data portals as possible and created a method for importing the catalogs into the Hub. Then, we transformed the data into a format that is suitable for further use.

We designd and implemented an appropriate asynchronous two-step data importing and processing architecture to provide maximum flexibility and possibilities for future enhancements. In this regard, we also provided entry points for metadata enhancements in the open data conversion step implemented in the `datasource-generator`.

Although implementing the according frontend was beyond the scope of this thesis, we provided all necessary features and APIs for retrieval of the crawled and processed data.

In general, the implementation of the data set, or retrospectively, data source integration into the JValue Hub can be considered a success. We now have a technical foundation that provides all necessary connection points to build upon. Once the new components are integrated into the UI, end users will benefit from access to various data sources, including their license information and much more. This will enable users of the JValue Hub to build data pipelines with ease and manage their resources more effectively.

However, the work on the data source feature is not yet complete. There are many possibilities for improvement in the future. For example, we could generate enhanced data source metadata, link data sources to Jayvee projects, suggest fitting data sources for projects, or support additional APIs for importing data sources from other open data portals. Possibilities are endless, and the JValue Hub team will continue to work on improving the platform and providing the best possible experience to users.

# Appendices

# A Requirements workshop notes

- Scheduled update
  - create new one
  - deal with removed ones
  - update existing ones (metadata / data)
  - expect edge cases: removed data (but not metadata)
- Link projects with data sets
  - 2 ways:
    - (1) Create project from data set ⇒ automatic link
    - (2) Create project from scratch ⇒ suggest links to user / let them choose
- Which portals are crawled
  - Focus on CKAN
  - make configurable
  - Johannes: manual import and extendable to completely different sources
  - Philip: don't do manual import, focus on automation (Johannes agrees)
- How to deal with data sets that Jayvee cannot process?
  - Concerns about data size
  - Philip: no worries about the data size
  - Philip: show everything in hub but make note when not possible ⇒ link to feature request in Jayvee (Georg agrees)
- Reporting statistics which data sets are linked most / created most / shown most in searches
  - helps with Jayvee development
- Crawler should only deal with metadata (not the actual data)
- How to trigger the import? Security concerns

  - Johannes: not include into hub
  - Georg: docker container that we can deploy, e.g., as k8s jobs (Philip and Johannes agree)
  - Philip: no manual CLI tool
- Voting / Stars?
  - Dirk: vs. number of uses
  - Johannes: current UI design has stars feature
  - Philip: makes sense in the future, but not in thesis; focus on stable imports
- Configuration
  - same as plans: in JValue-shared?
- Search for data sets
  - filter for tags and keywords?
  - worries about keywords if originate from import
  - Philip: do a prototype and then decide
- UX design: columns and rows not in metadata of CKAN
  - Georg: don't show and leave to future work (Philip agrees)
  - Same data set from different portals?
    * portal currently decoded in logo
    * display source / portal link
    * Georg: dataset is per portal ⇒ there can be data sets with same name but different portals
    * Philip: is there a federated field in metadata?
    * Philip: checksum as indicator
- Naming of concepts: data set is in CKAN a collection of distributions
  - Philip: stick to Govdata terminology
  - Johannes: same data in different format should be a data set
  - Georg: different data within a Govdata data set should be separate JValue datasets

  - Philip: maybe other naming but same structure: data collection (Govdata: sets) and data sets (Govdata: distribution) (Georg agrees)
  - Wikipedia: data set is a collection
  - Conclusion: research terms, choose some better names available, but mirror structure
  - Philip: maybe data source instead of data distribution to map to the terminology of Jayvee
- Search on data set or source?
- Search vs. what to show
  - Philip: Potentially search over more than to display in the end
  - Johannes: focus on one (everyone agrees)
- Focus on quality over quantity of features (everyone agrees)
  - e.g., help debug what went wrong when we were banned from a platform etc. ⇒ persistent logs
- Some statement about resources / costs / time consumption would be nice as evaluation
- Configuration of import
  - Not via GUI
  - Separate application
  - Interface: database or API?

# References

AG für GovData. (2022). DCAT-AP.de Spezifikation 2.0 [Accessed: 2024-11-27]. https://www.dcat-ap.de/def/dcatde/2.0/spec/

CKAN Project. (n.d.). CKAN DCAT Extension [Accessed: 2024-11-27]. https://extensions.ckan.org/extension/dcat/

CKAN Project. (2023). CKAN API Documentation. https://docs.ckan.org/en/latest/api/

Govdata Team. (n.d.). Govdata Metadatenkatalog [Accessed: 2024-11-27]. https://www.govdata.de/suche/daten/govdata-metadatenkatalog

JValue Project. (n.d.). Vision and Mission [Accessed: 2024-11-27]. https://www.jvalue.com/

World Wide Web Consortium. (2014). RDF Schema. https://www.w3.org/TR/rdf-schema/

World Wide Web Consortium. (2024). Data Catalog Vocabulary (DCAT) - Version 3 [Accessed: 2024-11-27]. https://www.w3.org/TR/vocab-dcat-3/