

Prototype for a Scalable Web-based Research Environment

Master's Thesis

Mathis Neumann, B.Sc.

August 2, 2017

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
Nelson Tavares de Sousa, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 2. August 2017

Abstract

The work of scientists involves finding and aggregating research data to answer their scientific questions. This thesis designs a prototypical web application to help them find, access, and evaluate data from multiple sources at once. Additionally, the concept is implemented as a prototypical web application. We achieve this by designing a distributed architecture to integrate existing research data repositories. Metadata harvesters retrieve information from these sources and integrate it into a search engine. The search functionality is accessible through a user interface which is designed to be extensible and maintainable. Its functionality allows researchers to not only search but also explore and collect relevant data sets. They can then preview them within their web browser, share them, and download multiple sets at once.

We verify the functionality of the system through a set of testing scenarios, showing its feasibility. This includes the integration of an external data repository as a proof-of-concept. The scalability of the system is then analyzed by simulating a high number of users simultaneously accessing the search functionality. Contrary to the design goals, the measurements indicate a decrease in performance when replicating the search engine. In addition to the validation of technical aspects, we present the application to researchers and interviewed them. This first user study indicates that it can provide a benefit and support in their daily work.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Document Structure	2
2	Goals	3
2.1	G1: Conception of a Scalable Software Architecture for a Web-based Research Environment Prototype	3
2.2	G2: Implementation of a Web-based Research Environment Prototype	3
2.2.1	G2.1: Web Browser User Interface	3
2.2.2	G2.2: Implementation of the Search Engine	4
2.3	G3: Evaluation	4
3	Foundations and Technologies	5
3.1	Research Data Management and Life Cycle	5
3.2	Web Platform Fundamentals	5
3.2.1	Locating Resources on the Web	5
3.2.2	The HTTP Protocol	6
3.2.3	Web Browsers and Their Restrictions	7
3.3	Anatomy of Scalable Web Service Architectures	7
3.4	Elasticsearch and Search Engine Terminology	9
3.5	Managing Docker Containers	12
3.5.1	Rancher	13
3.6	The Vue.js Framework for Component-Based Frontend Applications	15
4	Approach to a Web-based Research Environment	19
4.1	Design Concepts and Architecture Overview	19
4.2	Providing a Distributed Search Engine	21
4.2.1	Selection of a Suitable Search Platform	23
4.2.2	Designing a Scalable Search Architecture	25
4.3	Harvesting Metadata for the Research Environment	26
4.3.1	Metadata Ingest API	28
4.4	Designing the Web Frontend	29
4.4.1	Component-based Frontend Using JavaScript	30
4.4.2	Coding Styles and Conventions for a Maintainable and Extensible Frontend Application	30
4.4.3	Requirement Identification and Component Extraction	32

Contents

4.4.4	Providing a Research Environment	33
4.4.5	Providing a Common Data Access Interface	34
4.4.6	Providing Downloads of Bundled Files	36
5	Backend Services Implementation	39
5.1	Restrictive Service Management	39
5.2	Design Conventions for a Dynamic Service Deployment	40
5.3	Search Engine Mapping Using the DataCite Schema	41
5.4	Query and Ingest Proxy	42
5.4.1	Accessing Metadata with the Query API	42
5.4.2	Ingest API to Publish New Metadata Documents	43
5.5	Data Access Services	45
5.5.1	File Proxy for a Uniform Data Access Interface	45
5.5.2	File Merger to Combine Multiple Files	46
6	Implementing an Extensible and Maintainable Frontend Application	49
6.1	Code Structure and its Conventions	49
6.2	General Layout of the User Interface	53
6.3	Providing a Search User Interface	55
6.3.1	From Search Input to the Results	55
6.3.2	Displaying Search Results	60
6.4	Document Detail Page	62
6.4.1	Retrieving the Document	63
6.4.2	Component Structure of the Detail Page	63
6.4.3	Metadata Preview Components	64
6.4.4	Finding Similar Documents	66
6.5	Research Page	69
6.5.1	Accessing and Handling Files	70
6.5.2	File Preview Components	71
6.5.3	Managing the Data Collection	73
7	Evaluation	75
7.1	General Test Setup	75
7.2	Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories	76
7.2.1	Question: Can external repositories be integrated?	77
7.2.2	Question: Does the File Provider allow file downloads from external services?	80
7.2.3	Question: Does the File Provider restrict the throughput of the file download?	82
7.2.4	Question: Does the File Merger service bundle downloads from external sources?	83

Contents

7.2.5	Question: Do the file services stream data to the client?	84
7.2.6	Question: Is the architecture scalable to support high load scenarios?	86
7.2.7	Threats to Validity	94
7.3	G2.1: Web Browser User Interface	96
7.3.1	Question: Does the use of facets narrow down search results?	96
7.3.2	Question: Is it possible to preview CSV data in the browser?	98
7.3.3	Question: To what extent does the application provide value for the research process?	100
7.3.4	Threats to Validity	103
8	Related Work	105
9	Conclusion & Future Work	107
	Bibliography	109
	Appendices	113
A1	Query API	113
A2	Ingest API	113
A3	File Provider	113
A4	File Merger	113
A5	Frontend Application	113
A6	Elasticsearch Mapping	115
A7	Evaluation Scripts	115
A8	DataCite to Elasticsearch Transformer	115
A9	Interview Transcript	115

Introduction

1.1 Motivation

The general research process involves collecting data, the analyzing of the data and the publishing of results in scientific literature. In the case that multiple research teams rely on the same data, but do not cooperate, the data has to be gathered twice. To mitigate this, the principal of re-usability and public access is recommended by the European Commission to enable that research data is "accessible, usable and re-usable through digital e-infrastructures" [European Commission 2012]. Following this recommendation, the project *Generic Research Data Infrastructure (GeRDI)*¹ aims to provide and improve the means to allow open and interdisciplinary access to the corresponding data [Grunzke et al. 2017].

Existing data repository projects like *Zenodo*² or *Pangaea*³ are centralized and not suitable for every scientific community, because requirements differ based on the scientific fields. Discipline specific research data repositories provide storage capabilities for the specific needs of their field, e.g. *Pangaea* for earth and environmental sciences. However, the exploration and access of their data is hindered for other scientific communities, as field specific knowledge is required. The prototype proposed in this thesis explores an architecture that can practicably span and combine a large set of research data repositories for scientists. The prototype aims to explore an architecture and its feasibility for projects like *GeRDI* that combine such a broad landscape of data repositories into a set of tools for researchers. The prototype implements a search engine to help scientists find relevant data sets, allows for exploration based on results, and introduces a data collection. This collection provides a way for scientists to review multiple data sets, inspect their files, and share them with colleagues. Furthermore, the access to the data will be provided via a uniform interface across multiple repositories. This aims to be the foundation for further research of additional functionality that improve the workflow of scientists with research data. These tools are combined into a web application that provides a virtual research environment.

¹<http://www.gerdi-project.de>

²<http://zenodo.org>

³<http://pangaea.de>

1. Introduction

1.2 Document Structure

We begin this thesis by introducing and describing the goals in Chapter 2. The foundations and technologies necessary to understand this thesis are described in Chapter 3. We then identify the requirements from the goals and design a suitable architecture for services and the application frontend in Chapter 4.

In Chapter 5 we describe how we implement the server-side architecture through a set of services. The implementation of the user interface is described in Chapter 6. At last, we evaluate our implementation in Chapter 7 based on the previously defined goals.

Goals

2.1 G1: Conception of a Scalable Software Architecture for a Web-based Research Environment Prototype

This thesis introduces a web application with a focus on research data exploration and therefore must be able to provide access to a diverse set of research data repositories. The application provides functionality to search for data sets by their metadata and functionality for data retrieval, including displaying capabilities in the web browser. In order to support the diverse data repository landscape, it must be adaptable to allow search for multiple metadata representations.

The architecture for the application must be focused on handling a large amount of simultaneous users through a distributed infrastructure.

2.2 G2: Implementation of a Web-based Research Environment Prototype

In addition to the architectural concept described with the first goal G1, this thesis also implements the architecture as a prototype. The first part of this section defines the scope of the user interface, followed by goals for the services.

2.2.1 G2.1: Web Browser User Interface

The application provides a web-based user interface (UI) which consists of multiple components. For this prototype it is limited to a search component, download functionality, and data inspection interface. The search results page must provide additional means to apply further filters on the list, e.g. limiting the results to a source repository or subjects where applicable. This is also referred to as *Faceting*. A detail page displays all the information that is stored in the search index and provide download links to the actual research data, if possible.

In addition to handling metadata, the user interface has to provide a way to download data sets from the original research data repositories. This must not only include single file downloads, but functionality to download multiple data files at once. For this prototype the

2. Goals

download functionality is constrained to only support downloads from data repositories which allow public HTTP download access to the files.

Furthermore the UI implements means to display *comma separated value* (CSV) files within the web page to allow user to inspect or preview the data as a proof of concept.

Both the search and data access features are implemented within an extensible user interface which provides a framework for further functionality and maintenance.

2.2.2 G2.2: Implementation of the Search Engine

The search engine has support for a multitude of data types, e.g. keywords or geospatial information. The metadata schema definition is limited for prototype and testing purposes and is not designed for advanced scenarios. In order to make the metadata from research repositories searchable, the metadata has to be retrieved and processed. Supporting a diverse set of metadata sources, requires the possibility to implement custom harvesting strategies for repositories when standards are not available.

2.3 G3: Evaluation

In order to verify that the implementation meets the goals provided above, the application must implement all listed features and needs to be tested accordingly. This includes verification of the search functionality. The adaptability of the architecture to integrate new data repositories is crucial to evaluate the feasibility of the prototype. The implementation of the download functionality must also be verified. It should always allow the download of data if the data is accessible for the application on the source repository. This requirement is also applicable to the displaying of data.

Since scalability is also an important aspect of this thesis, the architecture must be able to apply horizontal scaling at run-time to distribute the network traffic across multiple servers. Additionally, it must be evaluated whether the functionality of the user interface provides benefits to researchers.

Foundations and Technologies

In order to design and implement the goals from Chapter 2, we need terminology and foundational knowledge about technologies used in this thesis. We begin by introducing terminology about research data in Section 3.1. In Section 3.2 we describe basic web infrastructure and protocols. Section 3.3 defines terminology when describing software architectures in the context of web development. The search technology used in this thesis is described in Section 3.4. After that, we describe the used service management tool in Section 3.5.1. Lastly, we describe the application framework used and its conventions in Section 3.6.

3.1 Research Data Management and Life Cycle

The handling of research data is often described with a life cycle. An example is the life cycle of the UK Data Archive¹ which includes data collection, processing, analyzing, storing, publishing, and reusing. This thesis focuses on the aspects of reusing research data by improving access to it. Data is described by metadata like creator, title, location and other information which will be collected and stored by the search engine described in this thesis. The process of collecting the metadata is also referred to as *harvesting*. The data itself is stored at *research data repositories*. In the context of this thesis, we define a *repository* as a web service that a scientific community uses to store research data and make it publicly available. This service is hosted by research groups themselves or institutions who provide long-term storage. The usage of the term repository is equivalent to *Research Data Repositories* (RDR) defined by re3data.org [Pampel et al. 2013].

3.2 Web Platform Fundamentals

3.2.1 Locating Resources on the Web

Web resources like documents or images, are identified and locatable by a *Uniform Resource Locator (URL)* which element of a *Uniform Resource Identifiers (URIs)* subset. They not only identify the resource but describe the method of accessing the data [Berners-Lee et al. 2005].

¹<http://www.data-archive.ac.uk/create-manage/life-cycle>

3. Foundations and Technologies

The syntax of a URL consists of a *scheme* which is an protocol identifier followed by `://` used to access the resource, e.g. `http`. After that comes the *authority*, which contains the *host* and *port*. The host is either an IP address or a registered *domain* name using the *Domain Name System (DNS)*² server and an optional socket port, e.g. `www.example.com:80`. The `www` part of the example domain is also called *subdomain*. Following the server info may be an optional *path* hierarchy, which is separated by forward slash `/`, describing where the resource is located on the server. A URL can also end with a *query* component, initiated by a question mark character. The query contains key-value pair strings further describing what elements of the resource are wanted. We refer to them as *query parameters*

3.2.2 The HTTP Protocol

The technology enabling the world wide web is the *Hypertext Transfer Protocol (HTTP)*. It is the framework for all applications and services described in this thesis. The protocol is located in the application layer of the *Open Systems Interconnection (OSI) model* [Fielding et al. 1999] [Zimmermann 1980]. Communication is achieved through interactions of a *client* which sends a *request* to a *server*. The service answers with a *response* message. A message in the protocol is stateless and contains two parts. These are the *headers* and *body* of the message. A header is a meta information string about the message and is a key-value pair. It includes information like the `content-type` of the transferred message body. While some keys and their allowed values are defined by the specification, additional headers are possible but are often prefixed with `x-` to avoid collision with standardized headers. Headers can be applied to both requests and responses. For requests, the header section also contains a method (e.g. `GET` or `POST`), the path the resource is located at, the host name (ip or domain), and information what data types the requester accepts. The response contains a 3-digit integer status code which can be categorized by its most significant digit, i.e. codes starting with `2xx` describe successful actions.

The protocol *HTTP over TLS (HTTPS)* is the regular protocol over a *Transport Layer Security (TLS)* connection and therefore enhances HTTP with an additional security layer [Rescorla 2000].

The body of HTTP requests can be of any type. In addition to documents that use the *Hypertext Markup Language (HTML)* to display human-readable information, there are also several dominant markups for machine-readable data. Relevant for this thesis are the *Extensible Markup Language (XML)* and the *JavaScript Object Notation (JSON)*. XML is similar to HTML in its markup and can be validated through *XML Schema (XSD)*³. JSON provides a more compact syntax consisting of objects and arrays. Objects in JSON are key-value pairs wrapped in curly brackets, with a string key and the values being literals, nested arrays, or objects. Literals can be strings wrapped in double quotation marks, double precision floating point numbers, `null` for missing values, and boolean values [Ecma International

²https://en.wikipedia.org/wiki/Domain_Name_System

³<https://www.w3.org/XML/Schema>

3.3. Anatomy of Scalable Web Service Architectures

2013]. Arrays in JSON are lists of any of the supported data types. The values in objects and arrays are not limited to only one data type, but can be freely mixed. Multiple entries are separated by commas in both objects and arrays.

JSON is a syntax subset of the programming language *JavaScript*, making it the native choice when sending or receiving structured data with a JavaScript applications. While JSON is a first-class citizen of JavaScript, it is supported in most other programming languages because it is similar to structs or dictionaries in other languages.

3.2.3 Web Browsers and Their Restrictions

Web browsers, such as *Google Chrome*, *Firefox* or *Microsoft Edge* send requests to servers and visualize or handle their responses. For security reasons, they follow additional policies and restrictions. One of these is the *Same-Origin Policy*. The policy restricts the usage of resources that are not located at the same *origin* [Zalewski 2008]. Origin is defined as the combination of scheme (protocol), DNS host name, and TCP socket port of a URL [Barth 2011]. If the origin of the web page loads another resource dynamically via a script or style sheet, the browser will not perform the request by default. In situations where an external data source can be trusted but is not the same origin, e.g. in distributed systems, web browsers follow Cross-Origin Resource Sharing (CORS) mechanisms [van Kesteren 2014]. These define headers for both requests and responses which define which origins and other type of requests are allowed to access the resource of the server. The browser compares the current origin of the page with the allowed origin defined by the service and then blocks or allows further requests to that resource. For request methods that are not guaranteed to be without side-effects (e.g. POST), the browser will send an OPTIONS request to check the headers first. These *preflight* requests have to be accepted and handled by the server. These requirements have to be considered when for each service which is accessed by browser applications.

In addition to rendering static *HTML* documents, images and other elements, browsers also provide an execution environment for the JavaScript programming language. JavaScript applications can interact with the *Document-Object-Model (DOM)* which is the data structure and API of the HTML markup after it was parsed by the browser. Scripts can also send HTTP requests or interact with parts of the users system. However, this environment is running inside a virtual machine, similar to the JVM for the Java language. It implements sandboxing and restricts operating system interactions, like accessing the file system [Barth et al. 2008].

3.3 Anatomy of Scalable Web Service Architectures

Web applications like the one in this thesis can be split up into two basic components. The first component is the rendered web page inside a web browser which might hold additional logic through the use of the JavaScript programming language, referred to as

3. Foundations and Technologies

frontend. The logic which calculates the information for the browser will be referred to as *backend* and groups all the software on servers.

Since web application performance is dependent on the backend, it must be able to handle any incoming requests. When the usage of a single or fixed set of servers is not feasible due to the number of requests, the backend must be able to be distributed onto multiple servers. In these cases, the software needs to be able to handle *replication*. This describes the duplication of a software to multiple servers in order to distribute the work load. This "ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement" is called *scalability* [Bondi 2000, p. 1]. Scalability is achieved by changing the number of servers that execute the software system, also called *horizontal scaling* [Vaquero et al. 2011]. A common industry approach to achieve a high scalability is the implementation of the *microservice architecture pattern*. Microservice refers to a small independently deployable software which is focused on providing an autonomous service functionality [Lewis and Fowler 2014]. The architectural style comes with some best practices, e.g. limiting the communication of microservices to network communication and avoiding central databases. This style allows to monitor each microservice individually and apply scaling only to the specific parts of the system that require it [Newman 2015][Lewis and Fowler 2014].

Load balancers are used which implement strategies to distribute the network traffic between all *instances* of the same microservice and servers [Vaquero et al. 2011]. There are multiple load balancing strategies that can be used for such a purpose. Relevant for this thesis are DNS based load balancing using round robin methods and the use of load balancing *reverse proxies*. *Round Robin DNS* cycles through multiple IP addresses for each DNS resolution request [Schemers 1995] [Brisco 1995]. The effectiveness of this kind of load balancing can be decreased by client-side caching of the resolved IP address which will be reused for multiple requests. The use of DNS does not allow for advanced strategies like balancing load based on information from HTTP requests as these are not available when resolving DNS data. Reverse proxies such as *HAProxy*⁴ or *NGINX*⁵ can implement such features. These proxies are servers which accept and manage incoming HTTP connections and then choose a server which has to provide the requested resource or perform the requested action. The response may be further processed and then passed back to the client. Advanced functionality like answering with a cached response in cases where no backend server can process the request can also be implemented. Both presented load balancing techniques can also be combined, e.g. by using Round Robin DNS to select the reverse proxy which will handle the request and balance the load between all application servers.

⁴<https://www.haproxy.org/>

⁵<https://nginx.org/>

3.4 Elasticsearch and Search Engine Terminology

The prototype proposed in this thesis will contain a search engine for metadata of research data. In general, a search engine is a service that provides the ability to search through a set of documents. Depending on the software, documents might consist of unstructured, e.g. text, or structured data like XML documents or JSON. To improve the efficiency and especially response time when searching for data, documents are preprocessed through a set of algorithms in an *indexing* operation. The resulting data structure of these algorithms is called *search index* which will be used to answer queries instead of parsing the original documents every time. An example for an open source search engine is *Elasticsearch*⁶. Elasticsearch aims to be distributable across a network of servers (a *cluster*) while providing a JSON centered HTTP API. A cluster of Elasticsearch servers (*nodes*) is replicating its data between nodes that store the same search index. The replication allows for search requests to be answered by any server that contains the same search index, allowing for load balancing. When an index is too large to be stored on one machine, the index can be split up into multiple partitions which are called *shards*. The process of partitioning is called *sharding*. In Elasticsearch the number of shards and the replication factors can be configured. The combination of both forces all shards to be replicated N times to at least N nodes, with N being the replication factor. It is possible to have more shards than nodes and each node can then hold multiple shards. One of the shard replications will be selected to be the *primary shard* which is considered the *source of truth* in data conflict situations. All other replications will synchronize with the primary shard. The coordination of servers is provided by a *master node* which decides which shards will be allocated to which node and other management tasks. The master node is selected by an election algorithm through *discovery* which is repeated in case of failures or outages [Elastic 2017b]. There are also the *data node* types. Data nodes store search index shards, answer the search engine requests and are responsible for applying the search indexing algorithm. By default, a node provides the full feature set of Elasticsearch but can be configured to only handle a subset. Each node holds information about other nodes in the cluster through an internal service discovery and the master node as a broadcaster of state changes [Elastic 2017b].

The automatic replication in the cluster does not guarantee consistency between all servers at every point in time, but instead can have inconsistencies which are resolved asynchronously over time, also called *eventual consistency* [Fehling et al. 2014]. While storing and synchronizing the search index data, Elasticsearch also stores the source document which allows it to also be used as a database. Elasticsearch does not follow the *ACID* (*atomicity, consistency, isolation, and durability*) of transactional relational databases, due to the eventual consistency, missing relationship constraints, and other properties [Haerder and Reuter 1983]. These types of databases are often referred to as *NoSQL* databases as they (partially) drop these properties including the use of the *Structured Query Language (SQL)* in favor of other properties, like performance or scalability [Sadalage and Fowler

⁶<https://www.elastic.co/de/products/elasticsearch>

3. Foundations and Technologies

```
1 {
2   "greetings": [
3     {
4       "greeting": "hello",
5       "greeted": "world"
6     },
7     {
8       "greeting": "bonjour",
9       "greeted": "Bob"
10    }
11  ]
12 }
```

Listing 3.1. Example JSON document before Elasticsearch flattening.

```
1 {
2   "greetings.greeting": ["hello", "bonjour"],
3   "greetings.greeted": ["world", "Bob"]
4 }
```

Listing 3.2. Example JSON document from Listing 3.1 after Elasticsearch flattening.

2012].

The indexing of data from documents is implemented with the *Apache Lucene*⁷ library which implements search capabilities for text and other properties. In a simplified way, the Lucene index data structure is similar to a collection of key value pairs. The keys are *tokens* found in the documents with the values being the documents that contain these token. Tokens refer to strings that can be words, phrases, or parts of each to also provide search results for partial input or typing errors by users. In order to be indexable by Lucene, Elasticsearch JSON data is normalized by Elasticsearch, e.g. because Lucene does not support lists of objects. In order to support object arrays allowed by the JSON specification, the lists are flattened [Elastic 2017a]. Listing 3.1 shows an example of a JSON document before any preprocessing and Listing 3.2 shows the object as it will be processed by Lucene. By default, this will not allow queries that rely on the grouping of elements, e.g. by searching for the document in Listing 3.1 where the `greeted` person was Bob and the greeting `bonjour` but not `hello`. This distinctive information is lost with default indexing. To mitigate this, Elasticsearch provides a *nested* data type for such fields which generates a hidden document for each array entry that is indexed separately. While this allows for more context aware queries, it creates additional overhead and requires specific handling in some queries, like highlighting terms that were found.

In order to index documents, Elasticsearch uses a *mapping* description of the expected data which is similar to schemas in relational databases. A mapping for an index or

⁷<https://lucene.apache.org/>

3.4. Elasticsearch and Search Engine Terminology

```
1 {
2   "properties": {
3     "doi": { "type": "keyword" },
4     "description": { "type": "text" },
5     "authorName": {
6       "type": "text",
7       "fields": {
8         "exact": {
9           "type": "keyword"
10        }
11      }
12    },
13    "meta": {
14      "properties": {
15        "creationDate": { "type": "date" },
16        "source": { "type": "keyword" }
17      }
18    }
19  }
20 }
```

Listing 3.3. Example mapping for documents with a DOI, author name, creation date, and source string. Author name can be searched for by name parts and exact match.

```
1 {
2   "doi": "10.17487/RFC2818",
3   "description": "HTTP Over TLS",
4   "authorName": "Eric Rescorla",
5   "meta": {
6     "creationDate": "2000-05-01",
7     "source": "RFC"
8   }
9 }
```

Listing 3.4. Example document matching the mapping provided by Listing 3.3.

document type within an index consists of data type⁸ definitions for all expected and searchable properties within a JSON document. Listing 3.3 shows an example for documents such as Listing 3.4. It declares of which type the value of a property is text, integer, date, keyword, etc. Additional settings can also be provided, such as processing steps, e.g. converting text to lowercase, or customizing the tokenization of text. Due to the tokenization of text, searching for a specific string expecting a full match, e.g. by searching for a DOI such as 10.17487/RFC2818 might be stored in the index as multiple tokens, like 10, 17487, RFC2818. A search for the same DOI might resolve other documents which include all these tokens, instead of resolving the expected single document. To mitigate this, the keyword data type exists which stores the string unaltered as a token. Using a text type also allows to store

⁸Data type overview: <https://www.elastic.co/guide/en/elasticsearch/reference/5.4/mapping-types.html>

3. Foundations and Technologies

multiple versions of the same string, e.g. one with the default tokenizer and another storing the string as a keyword. This can be relevant for names as these might contain multiple words, e.g. Tavares de Sousa, and should be searchable by using only parts of the name (Tavares) and the complete name. Listing 3.3 implements this for the `authorName` property in the `can` is later searchable by `name` for token based search and `name.exact` for matching the complete name. Other important data types for this thesis are the geological location data types `geo_point` and `geo_shape`. The first being a tuple of longitude and latitude which describes a single location on the globe. The `geo_shape` type offers compatibility with the *GeoJSON* specification⁹. This allows for more complex shapes such as polygons and advanced location based search queries.

Search queries for Elasticsearch are simple text queries or a JSON string that uses multiple combined queries to get the expected results. Basic queries are achieved by using the `query string` query¹⁰. The string which is provided with such a query will be parsed by Elasticsearch and follows a specific syntax. By default, any words in the string will be matched against all fields that are defined to be included via the schema. Restricting the search to a more specific set of data can be achieved by providing the field name from the schema that should be used in the search followed by a colon and the search term afterwards. More than one search term are possible by wrapping them in quotation marks. Searching for the example from Listing 3.4 could be achieved by searching with the string `doi:10.17487/RFC2818`. Conjunction with the keyword `AND`, Disjunctions via `OR`¹¹, and complex field queries, e.g. a date range, can also be achieved in this syntax. JSON based queries allow for fine-grained control over the request. This syntax allows for multiple queries to be combined, additional filters over the result set and additional features like *aggregation* queries. Aggregations¹² are calculations which are applied to all results, such as counting the objects that have the same values at a field, finding the maximum value of a field and other calculations.

3.5 Managing Docker Containers

Docker is an open source platform for Linux *containers* allowing the isolation and encapsulation of software with all its dependencies [*What is Docker*]. Docker containers have some shared properties with virtual machines like isolation and portability, but are reusing the underlying kernel of the system and its shared resources and infrastructure [Docker 2016]. The reuse of these resources leads to an "equal or better" performance in almost all cases when compared to virtual machines [Felter et al. 2015, p. 1]. They can also have a smaller storage footprint than traditional virtual machines, as they do not require a full operating system to be included. Process isolation is achieved through kernel namespaces

⁹<https://tools.ietf.org/html/rfc7946>

¹⁰<https://www.elastic.co/guide/en/elasticsearch/reference/5.4/query-dsl-query-string-query.html>

¹¹When just searching for multiple terms, a disjunction is used by default

¹²<https://www.elastic.co/guide/en/elasticsearch/reference/5.4/search-aggregations.html>

3.5. Managing Docker Containers

for independent processes and resource usage is limited and isolated by using Linux control groups [Felter et al. 2015]. Communication between containers is still possible but implemented as network based interfaces, even if the communicating applications are running on the same physical machine.

Docker *images* are portable versioned bundles of all files that form the basis for the container once started from. Images are built using a configuration file (Dockerfile) which lists all necessary steps for creating an image. Once the images are built they can be distributed and deployed to other machines, while always staying unchanged and versioned. In order to share images with other parties in a standardized way, Docker also provides a *registry* as a repository software for images. The platform *Docker Hub*¹³ provides a free and public registry of images. The underlying registry software can also be hosted privately. While isolation is crucial for security, it provides a challenge when wanting to store any information permanently. Using the default setup, all files from inside containers are deleted completely when they are upgraded (by restarting with a newer version of the image) or removed. To mitigate this in cases when file system access should be explicitly permitted on the *host machine*, the user has to allow access to certain files or directories. External files can be bound to a path inside the container which for example can be used to persist some database state or loading an application configuration file from the host machine. The linked files and directories of the host machine are called (*data*) *volume*. Such an explicit permission to access base resources also applies to network ports which can be mapped from the host machine (*external port*) to any port within the container (*internal port*). Docker enables communication between containers by implementing its own configurable named networks. All containers connected to a network are able to communicate with each other and receive their own internal IP address. Network traffic that is not specifically allowed is by default blocked using a firewall. A DNS service provided by Docker allows containers of the same network to resolve each others IP address by the name of their container and access even their internal ports.

3.5.1 Rancher

This thesis is implemented with the open source software *Rancher*¹⁴ to manage containers on multiple host machines. Rancher is managing and interacting with an underlying container *orchestration* software while adding authentication, a user interface, SSL certificate management, and other functionality around the orchestration. It exposes all its functionality through a web-based *application programming interface (API)*. While Rancher can support multiple other orchestration frameworks, but also provides a native tool, called *Cattle*¹⁵ which we use. As most of the relevant features of Rancher are provided through Cattle and the distinction is not relevant for the understanding of this thesis, we will treat them as synonymous..

¹³<https://hub.docker.com/>

¹⁴<http://rancher.com/rancher/>

¹⁵<https://github.com/rancher/cattle>

3. Foundations and Technologies

Rancher/Cattle connects to multiple host machines and can deploy, monitor and manipulate their containers. Cattle uses the concept of *services* which are instances of the same image. A service can group multiple running or stopped containers which can be distributed onto multiple host machines, they are also referred to as *service instances* in this thesis. Services are grouped into *stacks* which can be configured to only allow communication between services within the same stack. The communication between containers within the same Cattle environment is achieved by implementing a *Virtual Private Network (VPN)*, *IPsec* by default [RancherLabs 2017]. The use of a VPN allows secure communication between multiple host machines. Cattle can also manage geographically distributed host machines spanning multiple computing facilities as long as the Cattle server can connect to all hosts. The DNS functionality of Cattle allows service instances to resolve the IP addresses of other services through the name of the service. If permitted via the Rancher configuration, services can also resolve IP addresses of service instances in other stacks by appending the stack name to the domain, e.g. `myservice.stackname`. Services of one stacks can also be linked into another stack which emulates that the service is within the other stack. As an example we could set up a stack `db` for a database which might have a database service and a service for an administration user interface (`dbadmin`). Another stack (`frontend`) would consist of two services which provide web sites that display data from the database. The database would then be linked into the `frontend` stack and is therefore accessible for its services. The `dbadmin` service would not be accessible if assuming inter-stack communication is disabled.

The deployment of services is scheduled by configurable rules which can constrain the deployment of services to specific machines, e.g. by limiting deployment to host machines which run specific other service containers. The service deployment and upgrade process can be achieved using configurable strategies. This includes an option to start new servers in batches of several services at once or to start new instances first and stop existing afterwards. By upgrading instances consecutively, a *rolling upgrade* can be achieved which mitigates time in which the application is not accessible. This provides a tool for high availability of services [Roush 2001]. A deployment of a container is considered successful once a configurable number of *health checks* passed. Health checks are configurable automatic requests sent the application to check if the application responds. This can include checks whether a socket accepts connections on a specific port or if an HTTP GET request returns a status code of 2xx or 3xx indicating that the HTTP server is active and ready. Health checks are one of the metrics that are collected from running containers. Other metrics are memory, storage, and CPU usage as well as throughput of incoming and outgoing network communication. These features allow for basic monitoring support and can be accessed in other tools through Ranchers API for more advanced features.

Rancher also provides an integrated load balancing service which can be added to any stack. The load balancer service is a wrapper around HAProxy with configurable rules which HTTP domain or path is handled by which service. By default, the proxy uses a round robin algorithm to balance the load between all instances of the configured service.

3.6. The Vue.js Framework for Component-Based Frontend Applications

Service instances which fail health checks or are disabled are automatically removed from the rotation or added when they are considered healthy. This is the foundation for dynamic scaling capabilities of services. With additional configuration, the load balancer can also implement additional functionality, like *SSL Termination*. This termination allows for a secure connection to a load balancer which handles private keys and certificates necessary for HTTPS, but then sends regular HTTP requests to the services. This normalizes HTTP traffic for services and avoids that each service has to have access to the private key and certificates, while traffic is still isolated because of the encrypted VPN between services [RancherLabs 2017].

3.6 The Vue.js Framework for Component-Based Frontend Applications

This thesis uses the *Vue.js*¹⁶ framework for implementing interactive JavaScript user-interfaces. Vue mainly provides an API to implement *components*, which are self-contained dynamic parts of the user interface which ideally provide a single functionality. A component in Vue describes a customizable element in the DOM of the page and consists of a template, a component script and possibly stylesheet declarations. Templates are based around the declarative syntax of HTML and enriches them with additional functionality. Other components can also be used in templates and are syntactically identical to HTML tags. Attributes on these tags, also called *properties (props)*, are passed to the component similar to arguments of a function. Properties can be bound to any value, by prefixing the property with a colon. Handling user interactions, e.g. mouse interactions, are implemented by binding a function to an event. Computed properties are value getter functions on the components which are automatically cached and only recomputed when the used properties are updated [*Computed Properties and Watchers*].

The functionality and interactivity of components is defined by their component script, which is a class that defines possible properties, methods, and computed properties. The rendering of components is done automatically, because components subscribe to changes to all properties. When rendering, the components template is compiled into virtual DOM elements. Other components used in the template are then evaluated and rendered if their properties changed. The virtual DOM is an data structure that matches the real DOM but is not rendered by the browser. After rendering the whole component tree, the resulting virtual DOM is compared to the current state of the web browsers DOM. This allows to apply only the changes to the DOM that are necessary without completely replacing all elements, minimizing the impact of the web browsers layout algorithms.

Components can contain their own state, however in large-scale applications this might lead to unclear state management. The use of the plugin *Vuex* introduces a centralized approach for managing the whole state of the application. The state consists of a JSON

¹⁶<https://vuejs.org/>

3. Foundations and Technologies

object which contains all information which is necessary to render the application at any point in time. This allows for a single unidirectional data flow from the top-most level which subscribe to changes of the state and pass the necessary part of the data to nested components. Most components are then similar to side-effect free function that receive properties and render the DOM structure based on their properties. Avoiding side-effects allows to reason about the behavior of a component with the goal of an improved maintainability. The manipulation of state is possible through *mutations* and *actions*. Mutations are synchronous functions that can alter the state JSON. Actions on the other hand can perform asynchronous operations and then commit new data to the state through mutations. For example, a `getDocument` action can be dispatched with a document id as payload, the action then performs an HTTP request and calls the `setDocument` mutation. After mutations are applied, the application rerenders all components that depend on the changed data. This forms a cycle which is shown in Figure 3.1 in which components can dispatch actions with the intention to change the state while actions allow asynchronous behavior. Actions then call the mutations which updates the internal state, automatically triggering an automatic rendering of the updated state.

3.6. The Vue.js Framework for Component-Based Frontend Applications

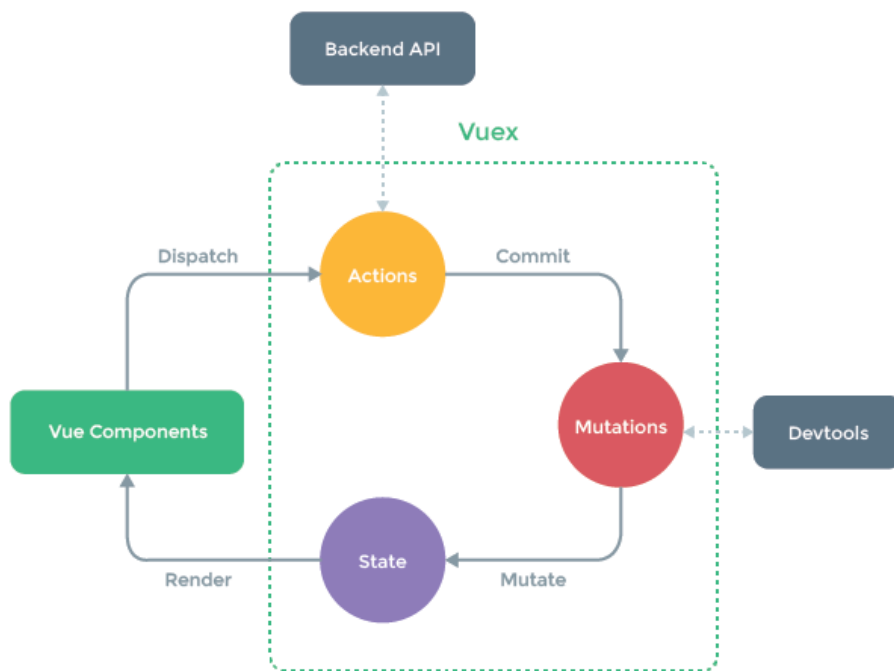


Figure 3.1. Basic data flow implemented by Vuex [Introduction to Vuex]. Components render the current state of the application, they can dispatch actions which can asynchronously update the state through synchronous mutations. Components are automatically rerendered when the state changes.

Approach to a Web-based Research Environment

This thesis designs and implements a software suite to provide a basis for a web-based research environment. In the following sections, we identify requirements and challenges in order to implement the goals from Chapter 2 which we then try to resolve. We begin with a general overview in Section 4.1 of what functionality is necessary, what components are required to provide this, and how they are connected. In Section 4.2 we design and describe the search infrastructure. The flow of harvested metadata to imported into the search infrastructure is described in Section 4.3. Section 4.4 describes the architecture of the user interface application and how we try to achieve this in an extensible and maintainable fashion.

4.1 Design Concepts and Architecture Overview

The overall architecture of the software needs to be distributed to multiple servers to support high network traffic situations, as demanded in goal Section 2.1. This thesis tries to prepare for these situations by implementing an architecture following the microservice architecture pattern. Each function is its own microservice which allows for individual scaling of services, e.g. the search engine, data access APIs, frontend etc. Services will only communicate through APIs reachable over network protocols, e.g. HTTP, while having an isolated state or avoiding state completely. Following this pattern allows the use of load balancing techniques to distribute the requests to all instances (replications) of a microservice. Additionally, it allows for multiple people to work mostly independent from each other by splitting responsibility for services between the people involved. The concept for the application does not only focus on the performance and scalability, but also takes great care to have a maintainable foundation for future additions.

As central starting point for the interaction and exploration of research data, we will utilize a search engine, to help researchers find the data they require for their work. Especially in situations when the person only has a general idea of what data is required or wants to browse a broad set of data objects, a search engine is destined to support such cases. The search not only provides a central role in the web application user interface (frontend) for the user, but also takes a key role in the architecture. The metadata, which is

4. Approach to a Web-based Research Environment

stored and indexed in the search engine, can be used to provide the functionality defined in the goals from Chapter 2. Next to the apparent search functionality to search for and explore research data through its metadata, the frontend displays the results with the corresponding metadata. Additionally, the download and inspection of actual research data sets (not its metadata) is based on the metadata objects from which they are extracted. Following these assumptions, the main purpose of the architecture is to handle research metadata and directly provide it to requesting user or use it to provide other functionality, like the download of data sets, as described in Section 4.4.5.

In order to separate the architecture into a set of microservices, we need to split the required features into separate services. The first feature is the search engine, which can be considered its own microservice, but requires for some additional functionality which can also be split up into multiple smaller services that interact with the central search engine. Even though such central search engine handles quite a set of features by itself, a single service allows for the use of existing software solutions, namely Elasticsearch. Further details about the search engine and the set of additional services will be described in Section 4.2. Another feature is the research data access interface which provides the ability to download datasets from an external repository through a uniform interface. This is required to implement the download and inspection functionality of the user interface for Goal 2.1 to avoid web browser restrictions, which we further describe in Section 4.4.5. This can be split up into two features and therefore microservices: the *File Provider* service that provides a download and a compression service, called *File Merger* service, which combines multiple files into a single compressed download.

In order to add new metadata to the search index, the data must be imported from external research repositories. Each repository might potentially use a proprietary format for metadata and any type of interface to provide them. Such heterogeneous requirements are not suitable to be handled by a single application or service, because decisions in the design phase of such an abstract service might not allow for future functionality. In order to avoid this restriction, we instead implement metadata importers as one or more microservices in a decentralized manner which allows to be applicable to any kind of automatic data retrieval. Abstract interfaces, e.g. to handle a specific metadata standard like the DataCite schema¹ [Brase 2009], can still be implemented by allowing inter-service communication in order to make them composable. In some cases it might be more manageable to convert from one schema into another, which is already integrated with the architecture, than to transform the original schema into one that is compatible with the proposed system of this thesis. The handling of harvested data is further described in Section 4.3.

Functionality like the data access service relies upon the extraction of data URLs from the metadata. To improve the fault-tolerance of the service when the metadata store (search engine) is not accessible, the service should have its own data store that is asynchronously updated to contain the state of the main store. Under the assumption that

¹<https://schema.datacite.org/>

4.2. Providing a Distributed Search Engine

future functionality will also depend on metadata documents, the infrastructure should provide a generic approach to implement redundant data store, instead of a solution just for this particular service. This can be achieved by using a *publish-subscribe pattern* in which every newly indexed document is broadcast and all microservices, that need to extract data from the document, subscribe to this update stream. A message broker service, e.g. *Apache Kafka*² can provide such functionality which can also be used for future messaging functionality, i.e. implementing queues or preprocessing messages before sending them to the subscribers. Such a message broker is a message oriented middleware as defined by Fehling et al. [2014, pp. 136-140].

With these requirements for the general architecture, we can extract main *service groups* which can further consist of multiple microservices. These additional services are described in more detail in the following sections, but work in conjunction to fulfill a more general service.

4.2 Providing a Distributed Search Engine

The architecture is designed around the handling of metadata which provides the basis for any functionality in this thesis. The search for metadata specifically provides the entry point for users to explore and work with research data. The conception of the scalable search engine as described in Goal 2.2 with the scalability considerations from Goal 1 requires distribution of the search engine and its index to multiple servers.

Figure 4.1 shows the *Unified Index Infrastructure (UII)* as described by the GeRDI search architecture fact sheet and provides the base concept for the search engine in this thesis [GeRDI Project 2017]. In that approach the search engine (blue elements) is a logical group of servers which contain the search index (here: *Index Node*). Each server receives its own replication of the search index, which can split up into multiple shards which work in conjunction, as explained in Section 3.4. A *Public Search Mask* provides the interface for users and applications to access the index to search and retrieve documents. Metadata is harvested from multiple external research data repositories (*repo*) and written into the index.

We can use this concept as a general foundation and identify separation potential to further split up the functionality in order to use load distribution more effectively. Figure 4.1 shows two possible generic use cases that the search engine must provide: the import of documents with its preprocessing/indexing and the actual search for documents. Assuming high network traffic situations for both processes, this separation helps to avoid a large number of incoming documents negatively affecting the performance of search requests and vice-versa. This requires that the preparation and preprocessing of documents can be performed on one server while the replicas receive the preprocessed results. One of the reasons for the separation is that we want to be able to route the traffic of new documents to

²<https://kafka.apache.org/>

4. Approach to a Web-based Research Environment

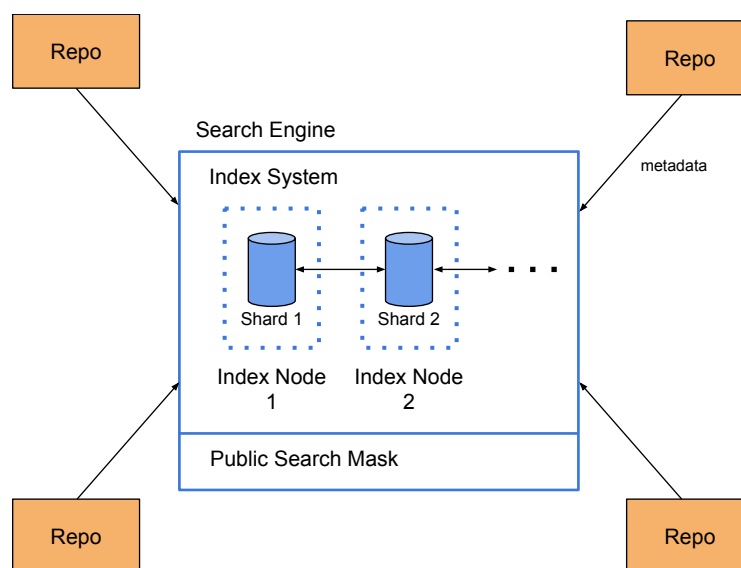


Figure 4.1. Envisioned distributed search index architecture (simplified), by GeRDI [GeRDI Project 2017]. The search engine consists of index shards distributed to multiple index nodes and a search interface (blue elements), the research data repositories (orange) are governed by external organizations

a specific set of servers. This separation can be achieved by having different subdomains for read and write operations. While HTTP path based routing would be possible, having it as different subdomains is more flexible for further use, e.g. the DNS for the subdomain could resolve to other load balancers or even data centers for write operations than for searches. Another argument for such a separation is that we can introduce microservices which act as proxies before reaching the elasticsearch. For the import of metadata documents this microservice can preprocess and normalize incoming data. One such normalization is the automatic generation of a unique (across the whole search engine) identifier for each new document based on a designed schema or adding the date of import to each document. This Ingest API is also the microservice where broadcasting new documents to a message broker is implemented. Furthermore, a proxy allows future enhancements such as automatic migrations of documents between different versions of the metadata schema. A feature which would most likely be required in future developments is the implementation of authentication in order to verify that only allowed providers can write data to the search engine. Such security feature is necessary for a production version of the proposed system, while the prototypical system in this thesis does not implement it. The same reasoning applies to the introduction of a microservice which receives all search

4.2. Providing a Distributed Search Engine

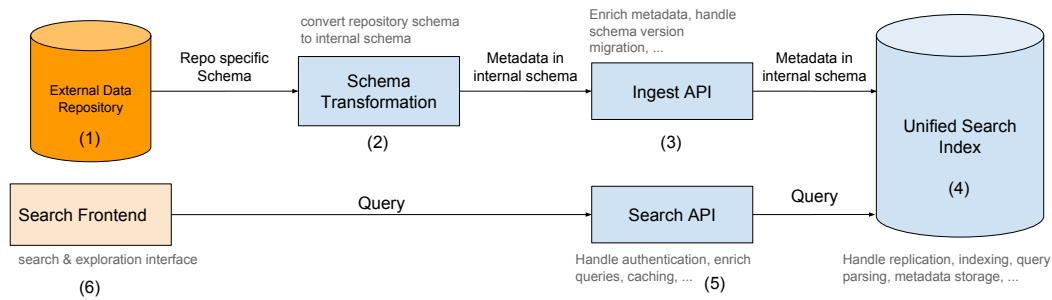


Figure 4.2. Data flow sketch for queries (6 to 4) and metadata import from external repositories (1 to 4) through a repository-specific import microservice

requests and relays them to the search engine (if allowed). This allows for the search engine software (Elasticsearch) to not be publicly exposed, especially since Elasticsearch only provides security restrictions and authorization for commercial customers through *X-Pack*³. The proxies only expose the import and search functionality for the allowed indexes while blocking other requests, e.g. index deletion requests.

Figure 4.2 shows the approach of how metadata can be imported from research repositories and be queried by users. The metadata is harvested from the external research repository (1) which holds the documents that will be added to the search index. As the metadata can use any metadata schema standard or use even a proprietary schema or retrieval protocol, a microservice (2) is responsible for retrieving metadata from that specific repository or standard. This service outputs the data into the schema for the search engine. The figure is a simplification of this process, as the service might use other services in the retrieval or transformation process. For example, this can occur when the metadata standard is already generically implemented by another service but with a different method of retrieval. Such a service may also hold its own metadata cache to ignore documents that are already imported. After the repository import service transformed the metadata into a compatible format, it is then uploaded to the Ingest API (3) which is located between the transformations and the search index (4). This is where new documents get a new identifier and other preprocessing steps occur as described above. Search queries from the user interface (6) pass through a search API (5) to restrict the publicly exposed functionality of the search engine.

4.2.1 Selection of a Suitable Search Platform

There are multiple open source search engine technologies available for free use. We narrowed it down to the two most popular engines, according to the ranking of DB-

³<https://www.elastic.co/products/x-pack>

4. Approach to a Web-based Research Environment

Engines.com [2017], which are Elasticsearch and *Apache Solr*⁴. Both are based on the full-text search library *Apache Lucene* and therefore use similar search result scoring approaches. In terms of functionality, both search engines provide the necessary requirements in order to fulfill Goal 2.2. Both enable *faceting* of search results, which allows for the user interface to show how many results contain similar terms, e.g. how many documents were found in a certain category, to narrow down on the search. Elasticsearch uses aggregations to achieve this, as stated in Section 3.4, while Solr has facets as its own feature. The indexing and search of geographic/geospatial data is also possible with both engines, they natively supports the GeoJSON specification which provides automatic compatibility with some location libraries, like *Leaflet*⁵ for map visualization in the browser. Solr and Elasticsearch both allow the use of JSON as the data format. However Solr's default data format and its focus is on XML. Elasticsearch on the other hand only supports JSON and YAML⁶, which can be considered a more human-readable form of JSON. This focus on JSON especially helps with the interaction from the JavaScript programming language as it is natively supported and most of the application in this thesis will be implemented in JavaScript. Mainly the frontend can directly interact with the Elasticsearch interface without any additional client libraries. The implementation of JSON in Solr however is more of a compatibility layer for XML, e.g. one of the ways to update multiple documents at once is by using the key add multiple times in one JSON object, which is not disallowed by the specification, but discouraged in the JSON RFC [Bray 2014]. It is not possible in JavaScript to create such a document with the native JSON serialization. However, Elasticsearch has similar issues regarding HTTP because it uses request bodies in GET request which is discouraged in the HTTP specification [Fielding et al. 1999, Section 4.3]. It should be noted that the JSON issue in Solr and the GET body issue with Elasticsearch can both be avoided by using different API endpoints. An argument for Solr is the support for XML which is an advantage when working with XML metadata standards like DataCite or *DublinCore*⁷.

Both systems are build for scalability, while Elasticsearch was designed to be distributable from the beginning of the implementation, Solr was enhanced to include *SolrCloud*⁸ which introduced features like sharding and replication in a distributed environment. SolrCloud includes *Cross Data Center Replication*⁹ to distribute the search engine across multiple data centers for additional redundancy and load balancing. However, Elasticsearch introduced *Cross Cluster Search* with version 5.4¹⁰ allowing a similar functionality. In terms of functionality, both search engines are reasonable candidates for the basis of all services this thesis designs and implements. During the work on this thesis the switch was made to Elasticsearch due to positive prior experience, the improved interoperability with JavaScript,

⁴<https://solr.apache.org>

⁵<http://leafletjs.com/>

⁶<http://yaml.org/>

⁷<http://dublincore.org/>

⁸<https://lucene.apache.org/solr/guide/6.6/solrcloud.html>

⁹<https://lucene.apache.org/solr/guide/6.6/cross-data-center-replication-cdcr.html#cross-data-center-replication-cdcr>

¹⁰<https://www.elastic.co/guide/en/elasticsearch/reference/5.4/modules-cross-cluster-search.html>

4.2. Providing a Distributed Search Engine

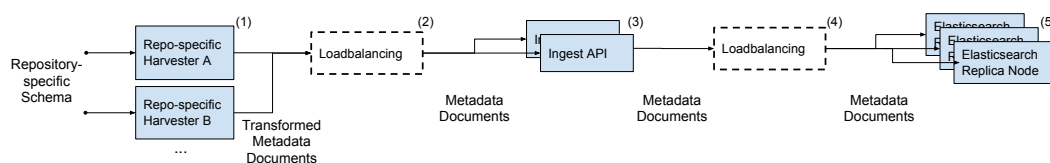


Figure 4.3. Addition of load balancing (2)/(4) to allow for horizontal scaling of import components. Shows the flow of data from external repositories (1) which is transformed to a generalized schema and written into the search index (5) through an Ingest API service (4)

and its JSON-centric API design.

4.2.2 Designing a Scalable Search Architecture

The scalability of the search architecture is defined by the scalability of the underlying search technology. Elasticsearch provides built-in functionality to allow the distribution of parts of the index (shards) across a dynamically definable set of replicas, which are eventually consistent. The query and Ingest API are scalable, because they do not contain any state, but depend on the search engine. Due to their lack of state, they can be scaled horizontally as long as the underlying search software can handle the requests. The use of load balancing techniques in between each layer allows for requests to be redirected to any of the service or search index replicas. Figure 4.3 provides an overview of the granularity at which horizontal scaling of components can be applied by showing the simplified data flow for the import of documents. The repository specific services (1) retrieve a metadata object in a proprietary format which they transform into a compatible document. The resulting documents are then pushed via a load balancer (2) to any instance of the Ingest API microservice (3) which further normalizes the data. It is then passed on via another load balancer (4) to any of the search engine nodes (5) which can handle the ingest of documents. This node can then initiate the indexing process. Figure 4.4 shows a similar approach for the search queries. Queries originate from the frontend application or any other client that has knowledge of the search interface (1). A load balancer (2) will then pass on the request to an instance of the public search API (3) which sends it to any Elasticsearch node to handle the request. Both figures only show a unidirectional data flow for simplification, i.e. no responses are included in the flow. Additionally, the request handling of Elasticsearch internally is not included. Each node in Elasticsearch can scatter queries to all shards which might include the same query results or which is defined to store a new document.

The load balancing layers from these figures are not necessarily reverse proxies, but can also be achieved through Round Robin DNS, as described in Section 3.3.

4. Approach to a Web-based Research Environment

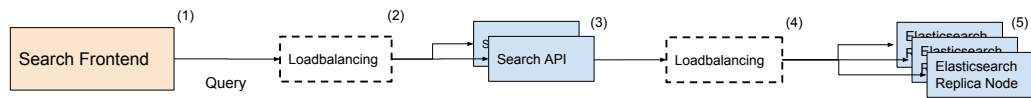


Figure 4.4. Addition of load balancing to allow for horizontal scaling of search components. Shows the unidirectional flow of search queries from the frontend (1) to a search engine node (5) through the search API (3)

4.3 Harvesting Metadata for the Research Environment

In order to provide its functionality, the research prototype needs to have access to metadata from external research data repositories. These repositories are not required to implement a certain metadata exchange standard or protocol, as having a large set of requirements and constraints for the metadata or its retrieval might make it impossible to include repositories with a proprietary schema or protocol. To mitigate this, we try to set as few requirements and constraints as possible by treating almost every information as optional. For the metadata, only a title and a string which acts as a unique identifier for a metadata document must be provided. Additionally, we store information where the metadata document was retrieved from, including the repository name and a link to the original metadata. Every other metadata information can be provided if the repository supports this information, e.g. geographic locations or full text descriptions. These limited restrictions should allow for the integration of repositories such as *Sea Around Us*¹¹ which only provide raw statistical data and therefore might be less focused on providing metadata such as descriptions. However, the transformation from the proprietary schema into the metadata schema used by the search engine should generally include as much semantic information as possible. Every piece of information that is included in the schema can be used to find and retrieve the document. Providing only the required fields with the document will create an almost invisible document, as this document does not provide any keywords or terms which a user might search for.

The integration of a repository can require domain-specific knowledge about the interfaces of the repository software and also about the metadata and data provided by the repository. This also includes the method of metadata retrieval which we cannot implement generically to include all possible standards. For the retrieval we use the same approach as we did for the requirements for metadata documents: by having as few requirements as possible and by treating the data retrieval from repositories as black box implementations. These are managed independently, with the only requirement being that it provides metadata data documents in a schema that is compatible with the search engine. This is achieved by implementing repository importers as microservices, following

¹¹<http://www.seaaroundus.org/>

4.3. Harvesting Metadata for the Research Environment

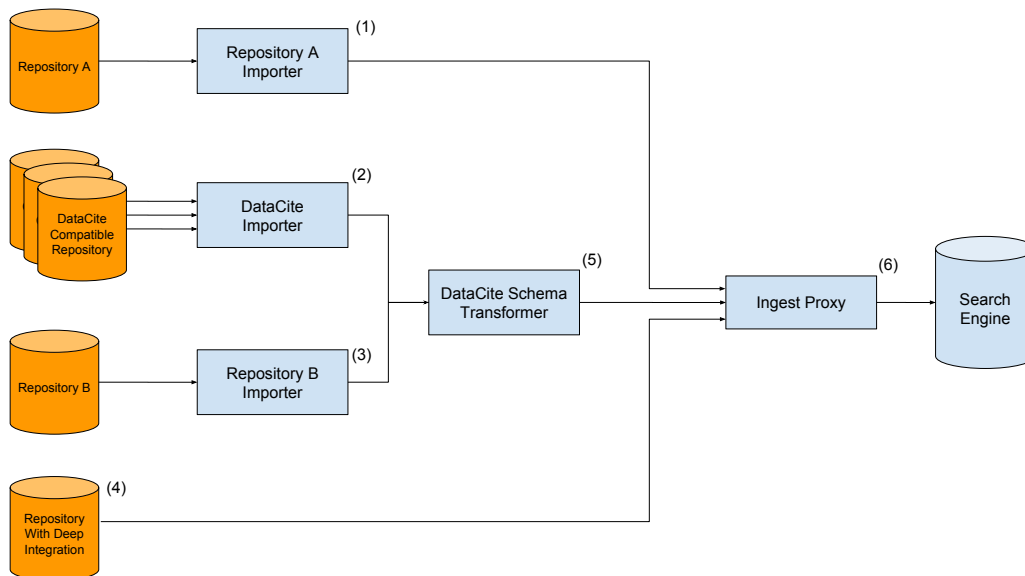


Figure 4.5. Examples for supported metadata data flows from repositories on the left to the Ingest API and search engine on the right.

the general architectural style of the backend. In order to minimize traffic and load on the infrastructure, the service should only upload documents that require updates. The detection of document changes should be self-contained within the service. If the data retrieval interface of the repository does not provide means to only retrieve documents that were updated since the last import, the service can keep its own database to check against for differences. The service should then transform the metadata into the generic metadata schema as described in Section 5.3 and upload it to the Ingest API service. The upload can also be implemented indirectly through another service to avoid duplication of functionality. The service could then retrieve the data from the repository, upload it to another service that transforms the DataCite schema into the schema compatible with the search engine and passes it on to the Ingest API. This allows for implementing arbitrary service compositions, avoiding duplication of functionality and separation of concerns.

Figure 4.5 shows import scenarios the document import infrastructure should support. The basic case is a service (1) that is designed to receive data from a single repository and transform the metadata documents directly into the metadata schema supported by the search engine. The transformed document is then uploaded to the *Ingest Proxy* (6) which we implement in this thesis and which is responsible for document validation and its publishing to the application. Its implementation is described in Section 5.4.2. A single service might also be responsible for the metadata retrieval from several repositories if they used the same metadata standard. This is shown with service 2 in Figure 4.5. In this

4. Approach to a Web-based Research Environment

example, the transformation between the DataCite schema and the schema of the search engine is extracted into its own service (5) which is reusable. Service 4 is an example for a service that can use proprietary data retrieval methods to extract DataCite documents (in this example) and pass them on to the existing DataCite schema transformer service (5). This demonstrates an example for service composition. The last example (4) shows a repository which natively supports an automatic export of documents to the application. This can be achieved by allowing the Ingest Proxy to be publicly accessible, which is possible with this structure, but should only be implemented if any means for authentication exist.

4.3.1 Metadata Ingest API

For importing metadata documents, we provide a service which is located between the uploading service, e.g. a repository import service and the search engine (Elasticsearch). First and foremost the service acts as a firewall to the Elasticsearch API, limiting the exposed functionality to only allow the upload of documents to specific indexes and as specific document types in Elasticsearch. This allows for the API of Elasticsearch to not be publicly accessible. In the prototype, this includes the validation that the mandatory `identifier` field contains a value in the uploaded document and that a source exists, specified by an identifier and a URL. Similarly to sources, provided file references are validated in order to ensure that they provide a URL to the file and an identifier, if they are provided within the document.

Documents may also require preprocessing: in our prototype we use the provided identifier to generate an internal id that is URL-safe and can be used within the system as well as in the frontend. This generated identifier must not only contain characters which are safe to use in the path segment of a URL, it must also deterministically generate the same id if the same document is indexed again using a hashing algorithm. This avoids the duplication of documents in case they are updated or indexed twice, as Elasticsearch will find and replace the existing document with the same id. After validation and preprocessing, documents will be pushed to the search engine which applies its indexing algorithms and stores the results and source documents. The Ingest API only includes validations that are not already provided by Elasticsearch, this requires that errors in the indexing step have to be accounted for. This can occur if the documents are not compatible with the internal data types of the mapping in Elasticsearch, e.g. providing string values on a field that is configured to only accept integers.

Documents that are indexed successfully by the search engine are then published to the message broker. This allows for other microservices to update their internal state, if they are dependent on documents. The order might change if the Elasticsearch is no longer the central metadata storage.

For this, it should provide a message stream for documents which are newly created and documents which were updated. This allows for services to only subscribe to the necessary streams, minimizing the documents to process. In this thesis, we use these message streams for the download service which extracts file URLs from metadata documents and

4.4. Designing the Web Frontend

stores them in its own independent storage system, see Section 4.4.5. The Ingest API is not responsible for the interactions of other services with the documents, it only broadcasts new documents through the message broker without awaiting responses from other services.

The API of this service provides a method to upload multiple documents at once, similar to the Bulk API from Elasticsearch¹². It accepts a list of documents which are to be updated or created. Having a distinction between updating and creating documents is not necessary in our approach because Elasticsearch can return this information automatically. This allows for simpler implementation of repository import services, as they do not need to make this distinction via any queries to a database. Additionally, this makes the function effectively idempotent and therefore repeatable in error situations. This allows to be only restricted to *at-least-once* message delivery which guarantees that the [Fehling et al. 2014, pp. 14]. Repeating a function is not strictly idempotent, as the response of the functionality might change when a partial set of documents was already processed but the resulting state in the search index will be the same. However, sending the same document (including the same id) multiple times to an Elasticsearch instance will just overwrite the document with the same information. This will lead to the same final state of the search index, but sent document as updated to the message.

Even though the API receives a list of documents, each document is to be treated individually. This means that documents which do not pass the validation, either by our own implementation or by the search engine, will not lead to a failure of the complete list of documents. However, this requires the Ingest API to return errors in such a way that repository import services can identify faulty documents, apply fixes and resend them until they are imported successfully. The approach to handling errors is such that the order of the incoming documents determines the order of responses, e.g. extracting the response for the second document is possible by checking the second response for a success message or error description.

4.4 Designing the Web Frontend

The frontend application is the user-facing part of the infrastructure and combines multiple other parts of the system into a single web-based interface. The prototypical application implemented by this thesis is the foundation for a research environment that helps scientist find, explore and retrieve research data. This can be achieved by providing them with a set of tools which they can use in their research process. Such tools can require an interactive visual user interface to make functionality of the backend service accessible to the user, e.g. the data download service requires a file selection and button. Otherwise the user interface can be the tool itself, like the visualization of metadata.

This thesis implements the foundation for an extensible and highly modular frontend application. The modularity of the frontend is the user interface counterpart to the microservice pattern of the backend. The features of the proposed system provide a proof-of-concept

¹²<https://www.elastic.co/guide/en/elasticsearch/reference/5.4/docs-bulk.html>

4. Approach to a Web-based Research Environment

for different use cases a user might have in regards to an interactive research environment, but cannot provide a feature-complete implementation. The provided features provide the first implementation for several classes of functionality and can be used as a guide for further enhancements in a maintainable way.

4.4.1 Component-based Frontend Using JavaScript

In a traditional statically rendered website the complete HTML document will be loaded on each request and each interaction requests a new version of the page. Thus creating a lot of network overhead, requiring the use of servers which aggregate data from all microservices that are necessary to answer a request. Additionally, the response can only be send to the browser if all the data from every necessary microservice was successfully received and rendered to HTML on the server. Until the server has completely prepared the full HTML document, the web browser cannot display any HTML which adds latencies to the rendering. Using a JavaScript application can shift the computing from a server to the web browser on the users machine. The fetching of data can be implemented asynchronously while the application can still be fully interactive and usable. Using a JavaScript based application also allows to let the browser directly interact with several backend services without the need for a frontend server other than a fast static file server.

4.4.2 Coding Styles and Conventions for a Maintainable and Extensible Frontend Application

When implementing the application with JavaScript, the strong separation of concerns from the microservice architecture pattern, can be introduced in parts by using a component-based application structure. A component is responsible for one visual part of the application, handles its visual rendering and provides the interactivity. Each component should provide a single purpose, but can itself use other components internally, e.g. a search input component can provide an extended text input and send the search request, but is not responsible for handling the results. This allows for new functionality to be developed independently and defines strict APIs for interactions by passing parameters to a component and handling some callback events. In conjunction with the microservice architecture of the backend, the frontend application can implement a component which provides the user interface for each microservice feature. These components can then also be maintained independently from one another and are reusable.

In addition to using components, we define a set of implementation rules, in order to further improve maintainable and component independence. Firstly we follow the rule that components can be considered functions that basically receive data as input parameter and render it to an HTML structure. Components should not introduce side-effects other than passing events to the parent component when there is any interaction with the part of the webpage which it is responsible for. The use of a centralized state management implemented with Vuex allows for components to also be rendered synchronously as they

4.4. Designing the Web Frontend

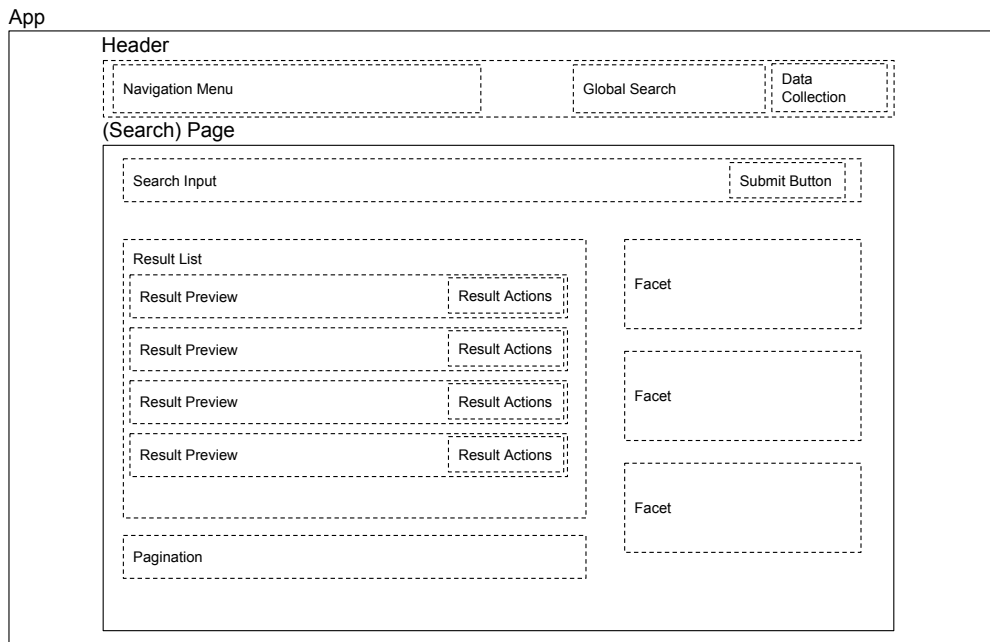


Figure 4.6. Component overview example for the search page. Dashed boxed are side-effect free components that only render the data passed to them. The boxes with solid borders are subscribed to the central state of the application and can mutate it through actions.

always reflect the current version of the central state, see Section 3.6 for more information about Vuex. Components are not responsible for updating the state, only displaying it. In order to enforce a clear data flow from parents to nested components, only the highest levels of the component tree must be connected to the state. These components are the pages which extract the data from the state management, handle updates and pass the data down to nested components. Figure 4.6 shows the search page as an example. Higher level components with solid borders are connected to the state management while dashed boxes represent components that only render the data passed to them from a connected parent. These pages should also provide their own route/path that is displayed in the browser and shareable, e.g. `/search` for the search page. In order to achieve shareability, it should be possible to get deep links to the pages and their state. These deep links should not only contain the page, but also the information necessary to recreate the current state of the page on another machine, basically a form of state serialization. For example, the search page includes the current search query and which page the user is on as query parameters in the URL, e.g. `/search?q=germany&page=2`. The focus of shareability might be important if multiple researchers are working on the same project.

4. Approach to a Web-based Research Environment

4.4.3 Requirement Identification and Component Extraction

In order to list the necessary components, we need to define the implementation requirements for the frontend application which are based on Goal 2.1. The frontend in this thesis serves two main functionalities: providing access to the search engine and the inspection and visualization of its results. We use the search as the main entry point for the research process. The search page is shown in the *(Search) Page* box within Figure 4.6. A search functionality requires a *Search Input* for entering a query. After using the *Submit Button* for sending the query to the search engine backend, the results have to be displayed in a *Result List* and may be browsed by the user through multiple pages using a *Pagination* component. This identifies components for displaying a list of nested result components (*Result Preview* in Figure 4.6), each displaying some preview information from the metadata document, such as title and parts of the description which contain terms from the search. The list of results may be narrowed down by the user through the use of facets, e.g. limiting the results to metadata that originate from the same source. There may be multiple facets applicable, which are located in a sidebar next to the results, using a *Facet* component. Each facet relates to one property within the result set and lists possible additional filter terms that preview, how many documents can be found through them, e.g. it can show that 90 documents of 200 results contain a specific keyword. This allows users to decide which additional filter provides the most specific subset of relevant results. Clicking on the facet value then changes the search query to also match for the term from the facet which will limit the number of results returned by the search engine.

While searching for metadata, the user should be able to select documents that are relevant to her/him and collect them for further inspection, shown as *Result Actions* in Figure 4.6. Each search result should also provide a detail page that lists all available information about the document, including but not limited to: title, description, files, links, and geospatial data. This detail page is reachable by clicking on the result preview. Within the detail page, each visualization of metadata information is provided by its own component and visualization. The description, for example, can be displayed as regular text, while keywords from the metadata document require a list visualization. In addition to showing the available metadata, the detail page is meant for implementing further exploration features for users. This means that viewing a document is another starting point for finding other relevant documents.

After finding relevant documents, a research page allows users to view the collected metadata and further study multiple documents at once. The collection of documents is always visible in the header of the application, while the main content of the page shows the data or metadata visualization the user wants to inspect. This is referenced as *Data Collection* in Figure 4.6. For each document in the collection, a button for the metadata and the list of files is shown, the latter of which provide a preview on click if supported for this kind of file. This preview functionality of data and files can reuse components from the detail page. This research page also provides the functionality to download all files in the collection as a single compressed bundle through the File Merger microservice,

4.4. Designing the Web Frontend

see Section 4.4.6. Furthermore, the page allows users to inspect files of a document and preview them, if they are in a supported format, such as CSV data, which is rendered as a table. Each preview is its own component which receives the loaded file content and has to implement a visual representation of the data.

Global components wrap all pages and provide the general layout and design of the frontend. The root of the component tree is the application component (*App*) with the general layout consisting of the global *Header* and the page content component which contains all the pages. The (*Search*) *Page* is an example within Figure 4.6. The navigation bar contains components for the *Navigation Menu*, a *Global Search* input and the *Data Collection* preview. Data collection behaves similar to a shopping cart on e-commerce websites and allows for quick access to the documents that the user wants to inspect in the research environment and also allows the download feature.

4.4.4 Providing a Research Environment

The research page provides a way for scientists to inspect, preview and gain knowledge from the data without downloading it first. Additionally, it may be used to compare and perform simple visualization driven evaluations if the data can help the researcher with their scientific question. The page is meant for quick interactions with the data and if they are helpful, the data can be retrieved all at once. If the data is not relevant, it can be removed from the collection, but the researchers did not have to download the data first, design evaluations and create visualizations, thus reducing the number of steps necessary to work with research data. In the prototype as proposed in this thesis, we will only include a basic set of data driven visualizations. These include the preview of CSV data as tables and a JSON document explorer to quickly browse structured data. The potentially large number of entries/rows in a CSV file have to be considered as large tables can have noticeable effects on the site's performance in web browsers. Mitigation techniques are described in its implementation, see Section 6.5.2. Next to the displaying of data extracted from files, the research page can also show information extracted from the metadata, like geographic locations, descriptions, keywords, and citable author information. The approach to handling data provides an extensible foundation for further visualizations that works with the actual research data, but also its metadata. The metadata may also provide relevant information for evaluating the data for further research, e.g. the geographic location or area the data was collected from or is relevant to. For that metadata, we provide more advanced visualization which is a map showing these points or polygon areas. The research page provides an overview over this relevant data and metadata. Additionally, the research page provides means to download all files referenced in the metadata object, either by downloading only the relevant files separately or by downloading all files as a combined archive.

The shareability of this research environment is achieved by providing a button to generate a shareable URL which can then be send to other people. Visiting this link allows users to adapt their selection of documents from other people. The share URL only contains

4. Approach to a Web-based Research Environment

the list of document identifiers in the collection, but does not provide a shared session between multiple users. In the future, such shared session would allow for collaborative work on a single research environment, but also requires additional backend services to store and manage these sessions. While not being part of this thesis, an extension can easily be developed due to the microservice architecture pattern and the highly modular frontend.

In addition to the serialization through the generation of sharing links, the document collection is saved in the client's browser. The research environment will restore the collection automatically if the user reopens the frontend application with the same browser. Since the prototype does not provide any login or session capabilities, this can be achieved by using local storage mechanisms in the user's web browser without requiring any backend service.

4.4.5 Providing a Common Data Access Interface

Web browser restrictions, as mentioned in Section 3.2.3, block access to arbitrary web servers from other web applications. In order to avoid these restrictions, the file server has to either be on the same domain as the frontend application or it has to provide HTTP headers that allow certain origins to access the resources. Since the files are located at the original data repositories, it would be impractical to try to let the repositories handle this. This would create a lot of dependencies and would hinder further integrations of additional repositories. We avoid this problem by introducing a *File Provider* microservice. This service acts as a proxy by basically downloading the file onto the server and then providing the download itself. This opens the possibility to implement all necessary technical requirements in a central service. The first approach at the beginning of working on this thesis involved the use of HTTP redirection. A service would respond to download requests, answer with a redirection response (HTTP status code 302) which includes the URL of the file, but also the necessary headers to avoid the cross-domain restrictions. This would not require servers to download the data first, minimizing resource usage. However, this caused faulty behavior in some older versions of the Firefox browser. This behavior is therefore following the official recommendation by the *World Wide Web Consortium* which disallows the usage of redirection proxies, as the CORS implementation also requires a check on the destination location, see Section 3.2.3 [van Kesteren 2014].

The approach to downloading the files indirectly via a server middleware introduces some challenges and caveats we need to address and mitigate. The approach to handling these downloads might introduce additional delays when a user requests a file and the download begins. Loading the files completely and providing them as a download afterwards introduces such a delay in which the user has no information about the download progress. Additionally, this requires the server to provide significant storage resources for large file downloads. We can mitigate both issues by passing each chunk of data that the server received from the original server to the requesting user. In web browsers, this leads to a visual indication and progress animation that the download started

4.4. Designing the Web Frontend

and how much data has been loaded already.

In addition to avoiding cross-browser restrictions, the download service also unifies the data access, providing a single, robust interface which can implement the vast landscape of research repositories. The API acts as a normalization step between the users who want to access files manually or programmatically, i.e. it always provides the name for the file and a compressed transport, even if the original file server does not. A uniform file download service can be achieved by including an identifier of the metadata document in the request and since a document might reference multiple files, it must also include a file identifier. We use a file identifier that is not dependent on the position in the list of the files within the metadata document or the URL, as documents might be updated in the future, changing the file order or URL. However, the implementation of the file identifier is done by the person responsible for integrating an external repository. The download URLs should be as stable as possible, even after such document updates. An example interaction is a request to an URL with the following convention: `GET https://download.example.com/<metadataId>/<fileIdentifier>/`. The placeholder `<metadataId>` identifies the specific metadata document in the search index. The service middleware then loads the metadata from a store, extracts the URL of the file with the help of the `<fileIdentifier>` and starts the download. This approach is loosely inspired by the Digital Object Identifier (DOI) System which is also a data tuple that identifies one object from an external publisher with a unified URL schema [Paskin 2010]. Since the metadata object holds the information about which repository contains the data, a repository specific download procedure can be implemented in the future. For example, this might be necessary if the repository requires a specific download flow, like preparing a temporary access token. Another aspect is that this centralized interface can be used in order to change the access methods, e.g. in the case that files are accessible from other URLs. It also provides an entry point to collect download metrics or implement file caching/mirroring mechanisms for redundant storage in the future, even though the latter might be hindered by legal challenges.

Following the principles of the microservice architecture pattern, the download service should have a decentralized data storage which does not rely on the central search index [Lewis and Fowler 2014]. This would create coupling which we avoid, as the download functionality does not require the functionality of the search, but requires the search as a mere database. We can achieve decoupling by having a separate storage system, solely for the download functionality. This introduces redundant storage and the download of files could be possible even if the Elasticsearch cluster is not available or is having high traffic situations. This store only holds the information from the document that is relevant for the file retrieval, i.e. the URLs to the files based on its identifier. The store of the microservice is updated asynchronously by receiving newly indexed or updated documents from the ingest proxy via the message broker. For this, the download service subscribes to these documents, extracts the relevant information and updates the store accordingly. In case a file is requested from a document that is not yet in the service-local store, the search index is still used as a fallback.

4. Approach to a Web-based Research Environment

4.4.6 Providing Downloads of Bundled Files

The ability to download multiple files at once was a suggestion by researchers in conversations within the GeRDI project and is included in the requirements for the user interface as stated in Goal 2.1. A simple approach to providing such functionality is to let the frontend application handle all downloads automatically. However, web browsers do not allow web applications to directly store files onto the hard disk of the user through APIs. Downloads can only be triggered, by providing a URL to download the files from, but there is no standardized API to trigger the download of multiple files at once. A possible workaround would be to open multiple download links in new browser windows or embedded sites, this could start downloads but provides a suboptimal user experience. For example, the Firefox browser by default opens a dialog for each file download and asks the user how to handle the download. Depending on the amount of files to download, this can be a lot of dialogs to confirm. In the Chrome web browser this is not the case, but instead a dialog appears, if the web application should be allowed to download multiple files at once. Another caveat of this approach is that the context of the file is lost, more specifically which metadata document is the source of the data. A user could for example download one `data.csv` file from marine biology data set and another `data.csv` from a geological source. Both of these issues can be avoided by downloading all files first and then combining them into a compressed *ZIP* archive which is then downloaded.

In theory, the JavaScript frontend application could download all files, hold them in memory, and create the archive completely on the client side. This could minimize resource usage on the backend, but is restricted on the client side. The resulting file can then be encoded as a data URL, which is the bit data encoded as a string in base 64 and can be provided as a download. However, this is limited to a maximum of around 4 gigabytes of data and the processing is restricted by an upper limit for the memory usage of web applications which the web browser enforces. Streaming the data as a download is not possible, due to the lack of such APIs in web browsers. Additionally, this is largely dependent on the users machine and could introduce a large burden on its computing resources.

Avoiding this caveat is possible by using a backend service which implements the compression and provides the bundled file. This service itself uses the common download interface introduced by the File Provider. A completely decoupled microservice with its own data storage would duplicate the main functionality of the File Provider, creating a lot of overhead and decreasing maintainability. The requirements for the File Provider, stated in Section 4.4.5, also hold true for providing such a compression service. This includes the focus on minimizing download latencies and resource usage on the server but should additionally allow created bundle files to not only include files from one metadata document, but potentially include files from arbitrary other documents.

A data stream based approach, similar to the file download service approach, can also be used for the implementation of the zip stream. However, avoiding temporary buffering of files on the server to minimize memory usage can increase the overall download time

4.4. Designing the Web Frontend

compared to a browser. That is because downloading multiple files in the browser can be achieved concurrently, if the bandwidth allows for it. Using the ZIP compression requires the sequential compression of files, because the format stores files in sequence. That means that implementing a concurrent download on the server requires the buffering of files which were only enqueued for the compression. However, the download of the compressed zip file stream has the maximum download speed when the next file is already buffered on the server. The file that is getting compressed can be streamed into the compression module and the compressed data chunk can be passed directly to the client. Similar to downloads from the File Provider, this allows web browsers to inform the user about the progress of the download (how many bytes were received). A percentage is not possible, because the size of the finished compressed .zip file is not available before the compression finished completely.

The download of multiple files should not be terminated if one or more files are not accessible or the download is interrupted. This would require users to restart the download. This requires the graceful handling of errors which we can achieve by adding an error log to the bundle file. The log includes instructions for the users how to access the file manually. In cases when the file download has an error which could not get detected, e.g. when a download stream closes correctly, but not all the data was transferred, we provide another log with all the links to the original repositories. The log maps the path and name of the file to the original download link for users to use manually in such situations. In order to keep the source context of the files, we group files into folders named after the title of the document the file originated from.

Backend Services Implementation

In this chapter, we describe the implementation challenges and final design of the services used in the infrastructure, following the approach from Chapter 4. Firstly, Section 5.1 defines what conventions and restrictions are used for handling a growing infrastructure with multiple metadata harvesters. We then describe the implementation details of the specified services, beginning with the implementation of the search engine's data schema in Section 5.3. After that we describe the implementation of the our *Node.js*¹ services. In Section 5.4, we describes how the query and Ingest API provide gateway functionality to the search engine. After that, the common data interface is described with the File Provider service in Section 5.5.1 and the download of multiple files through the File Merger in Section 5.5.2.

5.1 Restrictive Service Management

Services like the search engine, Ingest API, or frontend application have to be deployable on multiple servers. The deployment is done by building Docker images for each individual application in the system and running them as containers on servers. The management of these container instances is achieved by using *Rancher* which can deploy and monitor services in addition to other functionality, as described in Section 3.5.1. Providing the proposed research environment as web application requires that parts of the application are publicly available on the Internet. We have to implement measures to restrict access to the parts of the application that can only provide their functionality if they are accessible via the Internet. Docker containers are, by default, not accessible from outside of the same environment, unless TCP/UDP ports are explicitly exposed. However, services in the same Rancher environment can access all other services by default, but only through a virtual private network, providing secure inter-service communication without other public exposure. However, this setting is currently not active as it needs further evaluation. As described in Section 4.3, services responsible to retrieve metadata from external repositories might require domain-specific knowledge to be implemented. The implementation might be done by third-party developers who are more involved with the metadata access for a repository. Through the use of existing software solutions, we might introduce security risks as well. Therefore, we want to limit the possibility for such services to access all parts

¹<https://nodejs.org>

5. Backend Services Implementation

of the application, especially unrestricted access to the Elasticsearch API, which does not provide any authorization methods in the version used currently. In order to achieve this, we restrict the communication between services by default and only allow specific services to communicate.

First and foremost, we only allow access to the search engine (Elasticsearch) via the Query API and the Ingest API services. The Query API is a simple service that provides read-only access to a fixed part of the search index. It needs to be publicly accessible in order for the frontend application to use the search from the user's web browser. The Ingest API, however, should not be publicly exposed until there are authentication methods available, in order to verify that only a list of known third-parties can write new documents to the search engine. Only these two services are allowed direct access to the Elasticsearch nodes. This can be achieved by configuring the Rancher environment to only allow communication between explicitly linked services. This means that only the query and Ingest APIs are linked the Elasticsearch services, while others might only be linked to one of these APIs. The repository-specific import services also require explicit links and should only have access to the ingest proxy or another service that can handle the import, only allowing the least possible access to other services. A basic example is a service that retrieves the data from an external repository and only has one link which is to the Ingest API, disallowing access to other services within the environment. It should be noted that restrictions to limit the access of services to the rest of the Internet would require an external firewall. This also implies that services can access other services within the infrastructure if they are accessible from the Internet.

5.2 Design Conventions for a Dynamic Service Deployment

In order to support granular scaling of services, we need to implement load balancing techniques to distribute network traffic to all instances of a service. Services in the infrastructure must be scalable at runtime. Additionally, a distributed and dynamic infrastructure requires fault and partition tolerance whenever possible. The technical infrastructure to deploy new services at any point in time is provided through the use of Rancher for service orchestration. However, the implementations of services must be designed in such a way that they are not dependent on a specific instance of another service, but dynamically choose an instance for requests. Having such a dependency to other services does not allow for dynamic changes in the environment, leading to failing computations when that specific service is shutdown or fails. This means that services must not be dependent on a single long-living socket connection.

In order for Rancher to decide whether a service is still accessible, it should implement an HTTP health check resource. Some temporary faults or fluctuations might also not get detected by health checks, still requiring fault tolerant network implementations.

All services we implement with this thesis should follow the semantics of HTTP methods, in order to make assumptions about the behavior of the system state depending

5.3. Search Engine Mapping Using the DataCite Schema

on the method used. This mainly dictates that GET operations should provide a read-only operation, allowing these requests to be idempotent and therefore safely repeatable if the operation failed. Other methods may also be idempotent, but this has to be verified for each functionality. Idempotent requests are favored in the implementation if possible. Following best practices, retries can be implemented using an exponential backoff strategy to wait for the network or service to recover, assuming that the state is temporary [Amazon Web Services 2016] [Heorhiadi et al. 2016]. This approach uses an exponentially increasing delay between retry attempts so as to not overload the external service.

Having such a dynamic infrastructure also requires the connection between services to be adaptable to changing service deployments. Thanks to Rancher's DNS implementation we can rely on it when implementing any form of inter-service communication, as it respects service links and uses round-robin DNS for basic load balancing. As noted in Section 3.3, we can also use load balancing reverse proxies for more fine-grained control over the load balancing techniques, which even works when DNS responses are cached by the application. Services which are accessible via the Internet are only accessible via such load balancers as they allow routing of services based on HTTP domain or path information. The load balancer acts as a gateway to the internal service virtual private network, only exposing specific services. Additionally, having a single load balancer service responsible for SSL Termination, means that only it requires access to the private key files, minimizing the risks for security breaches.

5.3 Search Engine Mapping Using the DataCite Schema

In order to make metadata documents searchable with Elasticsearch, their structure has to be described in a mapping. The mapping configures the data types for each field, how they are searchable and what kind of preprocessing steps should be applied to the values in those fields by Elasticsearch. Supporting a wide range of incoming document formats with a single generic mapping is achieved by adapting the *DataCite Schema*², but transformed [Brase 2009]. The schema defines a set of supported information which form the metadata documents for research data. The purpose of DataCite metadata is to make the data academically citeable for other researchers. To achieve this, DataCite requires some mandatory fields, like a `title`, or `publicationYear`, `creator`, or `publisher`. Additionally, it identifies and describes other generic properties in order to support multiple external research data repositories and their data. Amongst these properties are multiple titles, geographical information, keywords (subjects), categorization, titles, and creators which is also relevant for the implementation of this thesis. The first issue is that the DataCite schema is provided as an XML based schema, so the schema has to be transformed into a JSON data structure Elasticsearch accepts. The full mapping is provided in Appendix A6. Further transformations would be necessary to support the date ranges that DataCite

²<https://schema.datacite.org>

5. Backend Services Implementation

allows as they are not natively supported by Elasticsearch. We try to workaroud this limitation by splitting date ranges into beginning and end. In this thesis, we not only focus on displaying all available metadata and making it searchable, but also provide access to the research data itself. However, the DataCite schema only includes metadata for the research data, but no suggested methods or information about how to access and retrieve the corresponding research data. The schema contains information about the file size and type but they are provided in different, independent properties. In order to support data retrieval, the adapted schema must include a separate list of files with nested properties including a URL to access the file, its type, an identifier and a label. The identifier is a document-local unique string, to identify the file when the document contains multiple files. Additionally, the adapted schema contains a list of web links with a label, the URL and a type which currently only supports the value `viewURL`. The `viewURL` type is meant for data visualization websites or a link to the source. This link is then to be displayed and used in the research environment user interface in order to visit the source website and can provide users with additional functionality. The adapted schema also contains URLs which link to the metadata source or sources. This can include links to APIs that were used to aggregate the metadata in the harvesting phase by a repository import microservice.

In order to support the transformation from DataCite documents to the internal schema, we implemented a transformation library which is provided in Appendix A8. While its general functionality is described in a set of test cases, it was not integrated into the system because of time concerns.

5.4 Query and Ingest Proxy

Restricting and controlling the access to the internal Elasticsearch instance is crucial to avoid malicious interactions from external users. Thus, one requirement is that the search engine is never requested directly. This provides the opportunity to define APIs for interacting with the search and enforce data constraints, as described in Section 4.2. We separate read and write operations into two services: the externally exposed Query API for metadata document search and the Ingest API for writing new documents.

5.4.1 Accessing Metadata with the Query API

The Query API is publicly available and is the primary data source for the frontend, as described in Chapter 6. It provides endpoints for both for search and retrieval of specific documents by their identifier. Both APIs are generally passing the information on to Elasticsearch's API. The `/metadata/search` endpoint accepts simple queries via `q` query parameter, but it also supports complex JSON-based Elasticsearch queries. These queries can be sent as the body of a POST request to this route or serialized and URL-encoded through the `?jsonQuery=<queryJSON>` parameter. Simple queries are transformed into `query_string` queries to have a uniform interface with the `elasticSearchClient` module. This module handles the

interactions with the search engine, passes the JSON queries to it and returns the results. Following the convention from Section 5.2, we implement an exponential backoff strategy if the search engine does not respond or responds with an error, in case this is a temporary problem. The search API only supports the minimal number of query parameters to provide the functionality that the frontend needs. This includes the `jsonQuery` parameter, but also `size` to limit the number of returned documents and `from` to support pagination. The `size` parameter is limited to at most 100 documents. This mitigates that external third parties can query too many documents at once, resulting in large resource usage for a single request. Loading documents via their identifier is possible by retrieving them via an HTTP GET request from `/metadata/document/<id>`. This is a simple endpoint that queries the Elasticsearch via `/<indexName>/_all/<id>`, in which `_all` allows to return the document with the identifier `<id>` of any mapping type. This is possible, because we defined our identifiers to be unique across indexes in Section 4.2.

Both the search and the document retrieval API limit the data that is accessible to a configurable set of indexes in Elasticsearch, avoiding unrestricted access to other data that might be stored in the search engine. This is currently only the index `datacite`. Additionally, the Query API service sets HTTP `Cache-Control` headers for both the search and single document retrieval. Configuring this header informs clients to store the document in cache and reuse it within a certain timespan. This will prohibit browsers from re-requesting documents or search results for a configurable amount of time, minimizing request load. For retrieving documents, we set a caching lifetime to one hour, because we assume that documents do not change often and if they change, it is not crucial that users can request the latest state of the data at any time. Search requests are set to only 5 minutes of caching time, because new documents can be added by repository harvesters at any time which can change the order of results.

While the service provides only minor additions to the API that is provided by Elasticsearch, it provides the basis for further features. For example, it could implement authentication based access restrictions for specific documents. These, however, are not in the scope of this thesis.

5.4.2 Ingest API to Publish New Metadata Documents

In order to support the import functionality as described in Section 4.3, we provide a proxy layer between the repository specific harvesters and Elasticsearch. The resulting service is the Ingest API which is inspired by the Bulk API of Elasticsearch. In our prototype we currently only support adding new documents via batch processing, meaning that only multiple objects at once through the `/import/batch` endpoint. Following the general conventions of our backend services, we use a route script file at the same file path as the URL path to accept incoming requests, i.e. `routes/import/batch.js`. The endpoint accepts POST requests with a body of either an JSON array or newline delimited JSONs³. The parser

³<https://github.com/ndjson/ndjson-spec>

5. Backend Services Implementation

is chosen based on the `Content-Type` header of the request, which is either `application/json` or `application/x-ndjson`. The route script is responsible for normalizing the incoming data type, resulting in an array of documents independent of the incoming file format. Newline delimited documents use a stream-based parser, which we buffer until all documents are parsed.

Each document is then processed in a validation step by the script `lib/document-validator.js`. The validator module returns a function that accepts an input and returns objects with a boolean `valid` property, the potential validation error, and the document. The validation is implemented with the library `joi`⁴ that provides a validation builder API, similar to a domain specific language. The validation for the prototype contains checks for the `identifier`, `sources`, and `files` fields in documents. The `identifier` is verified to exist and to have a minimal length of 10. The `files` and `sources` definitions are checked whether they provide syntactically valid URLs. Files are verified to have at least an `identifier`. The `source` definition is required to have at least a `provider` property for identification and a link to the providers website.

Invalid documents are filtered from the list and stored separately, along with their initial index. All valid documents are uploaded to the Elasticsearch instance by the `http/storeDocuments.js` module. The module transforms the valid documents into a newline delimited JSON string which is then uploaded to the Elasticsearch Bulk API. To configure the target search instance, we can use the `ELASTICSEARCH_BASE_URL` environment variable. The Bulk API returns a list of objects that contain status information whether the document was created, updated, or could not be imported due to an error. We merge this list with the previously collected validation errors and use the stored index information to get the same order as the incoming requests. The validation errors are wrapped into an object containing the property `status` with the value 400. The error message is nested under the `error` property. If a validation error occurred, we also add the `errors` flag to `true`. This roughly mimics the error response from Elasticsearch's Bulk API.

Broadcasting Documents to Kafka While the response is sent back to the requesting client, we asynchronously push the document to the message broker. The `lib/publishDocuments.js` module implements this functionality. In it, we iterate over the response list from Elasticsearch. The list contains a status indicating whether a document was created or updated in the index. We use the status to split the list of documents into newly created and updated documents. These are then pushed to a Apache Kafka message broker into two different topics, named `documents.created` and `documents.updated`. Both topic names are configurable via environment variables `KAFKA_TOPIC_DOCUMENT_CREATED` and `KAFKA_TOPIC_DOCUMENT_UPDATED` respectively.

The initial configuration for Kafka is realized via a initial connection to an Apache Zookeeper instance. The internal Kafka client library uses this to retrieve the IP addresses of Kafka nodes to connect to. Therefore, the Ingest API service must be able to access this IP. The environment variable `ZOOKEEPER_HOST` defines the Zookeeper target instance.

⁴<https://github.com/hapijs/joi>

5.5 Data Access Services

The file access services the research environment provides, consist of two microservices, a *File Provider* and a *File Merger* service. The File Provider implements a uniform HTTP interface for accessing files, resolving their name and allowing web browsers to handle those files. In order to support the browsers, we need to applying HTTP security headers which allow cross-origin request for the web browsers. Generally, it should provide a uniform API for accessing files for metadata documents while minimizing resource usage and latencies for the download, as described in Section 4.4.5. See Section 5.5.1 for the implementation of this central API.

Section 4.4.6 describes the requirements and motivation for implementing the a service to allow the download of multiple files and combines all files into a single bundled compressed archive. This should allow users to select and download all the research data they want in the user interface (frontend) and download everything with a single interaction. Files should also be grouped by the name of the source metadata object through folders. We can see a detailed description of the service implementation in Section 5.5.2.

5.5.1 File Proxy for a Uniform Data Access Interface

Security restrictions of web browsers, as described in Section 3.2.3, by default only allow same origin (domain) access. This can be mitigated by implementing a reverse proxy service that sets HTTP headers that allow access from a configurable domain, which is set to the domain, the frontend is located at. As explained in Section 4.4.5, this forces the use of a reverse proxy which downloads the data from the original file server and streams them to the requesting user. The HTTP interface to access the files uniformly is implemented as an HTTP GET request. Its path contains the unique identifier of the metadata document and the file identifier which identifies it in the list of files within the metadata document. When the proxy receives a request from a user, the document is resolved based on the metadata identifier and loaded from the data store (any Elasticsearch data node). In order to mitigate errors while retrieving the metadata document from the data store, failures are detected and connections are automatically retried for a configurable number of times. This is a consideration, necessary for fault tolerance in such a distributed setup of services which can for example be under high load from other request sources. Following best practices, retries are implemented using a simple variant of an exponential backoff strategy to wait for the network or service to recover, assuming that the state is temporary.

From the resolved metadata document, the file URL is then extracted and the proxy begins to send an HTTP GET request to the URL which loads the file from the original data repository. The headers from the initial user request are not passed to the other service, as this could contain private data, like authentication cookies, which is a privacy and security concern. Not using the step of resolving the document first, but instead allowing downloads of arbitrary URLs based on the request, would also introduce a security risk. Attackers could then for example download illegal files from any external source while

5. Backend Services Implementation

hiding their own identity.

When the download starts, the proxy server inspects the HTTP headers from the response and sends each chunk of data it receives directly to the requesting user. This avoids the use of any files or buffers to store the complete file on the proxy server, minimizing the memory requirements per download. Another advantage of this method is that web browser of the user can show the download progress to the user and does not have to wait for the proxy to finish the download first and then download it from the proxy afterwards. While the proxy server sends the file stream unaltered to the requester, the HTTP headers must not be send to the client unfiltered as this could lead to security issues. For example, the original file server could send headers which could alter CORS headers. Only a configurable list of headers are passed on to the user which include data such as the creation date of the file. The proxy also adds CORS headers and headers which force browsers to treat any data as a download, instead of trying to render the data, e.g. when loading HTML data. In that way, the service does not only provide download capability, but also access to the files for the JavaScript frontend application, which can then process and display the data. The name of the file download file is determined by the file server through its headers or based on the URL path and content type (*MIME type*)⁵ of the file, the latter for determining the file extension where applicable. If not provided by the file server, an identifier from the search index is used as the file name. At last, the server also adds headers to the response which provide the title of the source document, the original source URL, and file label from the store for later reference. These headers are `x-document-title`, `x-file-source`, and `x-file-label`. This is useful for services that require the download and file metadata, but do not require access to the data store, such as the File Merger service.

5.5.2 File Merger to Combine Multiple Files

The File Merger service provides an interface to send a list of multiple tuples of document and file identifier which will be downloaded as a single compressed Zip archive. These files will be accessed by reusing the File Provider service, as this already resolves the document, file data and the file name, which enforces separation of concerns and avoids duplication of logic. Similar to the implementation of the File Provider, the File Merger is implemented by compressing each received chunk of data as soon as possible. This allows for the memory usage to be rather constant. The size of the buffer necessary to run the compression algorithm on, is defined by the configurable compression level for the underlying `zlib` compression library⁶. The compression can only handle one file at once which requires that files are downloaded sequentially, while a web browser can handle multiple downloads simultaneously. The restriction on a single file stream per zip file would therefore lead to a overall download time based on the cumulative downloads times, while parallel downloads are at minimum as slow as the slowest single download. As a

⁵https://en.wikipedia.org/wiki/Media_type

⁶https://nodejs.org/api/zlib.html#zlib_memory_usage_tuning

trade-off between memory usage and download speed, the File Merger can download multiple files concurrently. This requires the use of buffers to download the files into, while one download stream gets compressed and appended to the zip file. Once the file is compressed, the next file buffer or stream will be selected from a queue and compressed. The queue in this implementation is a *first in, first out* queue. When the download is started, the HTTP headers are contained in the first data chunks of the download stream. Once they are retrieved, a new entry for the .zip file is then added to the compression queue which requires the name beforehand. Each file is located in a folder with the name being the title of the source document, automatically grouping all files from the same document. The source URL, document title, and file name are extracted from the headers which are written by the File Provider service, as described in Section 5.5.1. This information is then added to a list of sources which will be included in the zip archive as `_download_sources.txt`. The source URL is not a link to the common download interface (a link to the File Provider), but to the data repository where the file is actually located. This provides a way to users to directly access the files in situations when the file is corrupted or missing while also crediting the original source. Whenever the File Merger service detects that a file cannot get downloaded, the source URL and additional information is added to an optional `_download_errors.txt` file contained in the zip. An error will be detected and added to the file if the HTTP status code of the file response is not in the range of 200 or the stream is closed unexpectedly. The `_download_sources.txt` and `_download_errors.txt` are appended to the zip as the last entry of the compression queue. Both file names are prefixed with an underscore (`_`) such that they are sorted at the top of the file list in some file explorer softwares of users, like the *Windows Explorer* or *Macintosh Finder*.

Implementing an Extensible and Maintainable Frontend Application

The frontend application combines all backend services into a single user interface. As described in Section 4.4, the frontend allows users to search for and explore metadata objects. It also provides a research environment which allows researchers to not only explore metadata information of research data, but also delve into the research data itself. The following sections contain an overview of how we implemented this functionality in an extensible and maintainable way for further implementations based on this foundation, as defined by Goal 2.1. We begin in Section 6.1 by describing how conventions defined in Section 4.4.2 are applied to the general structure of the frontend and how we can deal with exceptions. In Section 6.2 we see the general layout of the frontend application, what functionality is globally accessible and how it is implemented. This is followed by the implementation details of the pages, beginning with the search engine in Section 6.3. The search results can be further inspected on the detail page, providing an overview on the information of a single metadata document. At last we describe the implementation of the research environment in Section 6.5 which can be used to preview files and information from multiple documents.

6.1 Code Structure and its Conventions

While the frontend provides separated pages like the search result, detail, and research page, it provides some global functionality and conventions with its implementation. The implementation of the frontend follows a set of conventions to increase its maintainability. In this section, we see how use the conventions defined in Section 4.4.2 for our implementation of the frontend.

Route Handling We begin by implementing pages with their own specific route in the application containing enough information to provide a shareable link to the pages and their content. Routes are paths in the URL of the application and components are registered to match some of the routes. The defined paths can also contain placeholders which are parsed similar to regular expressions, so that a single path definition is responsible for a any URL that matches this pattern. The matched parts can be accessed and are used to include

6. Implementing an Extensible and Maintainable Frontend Application

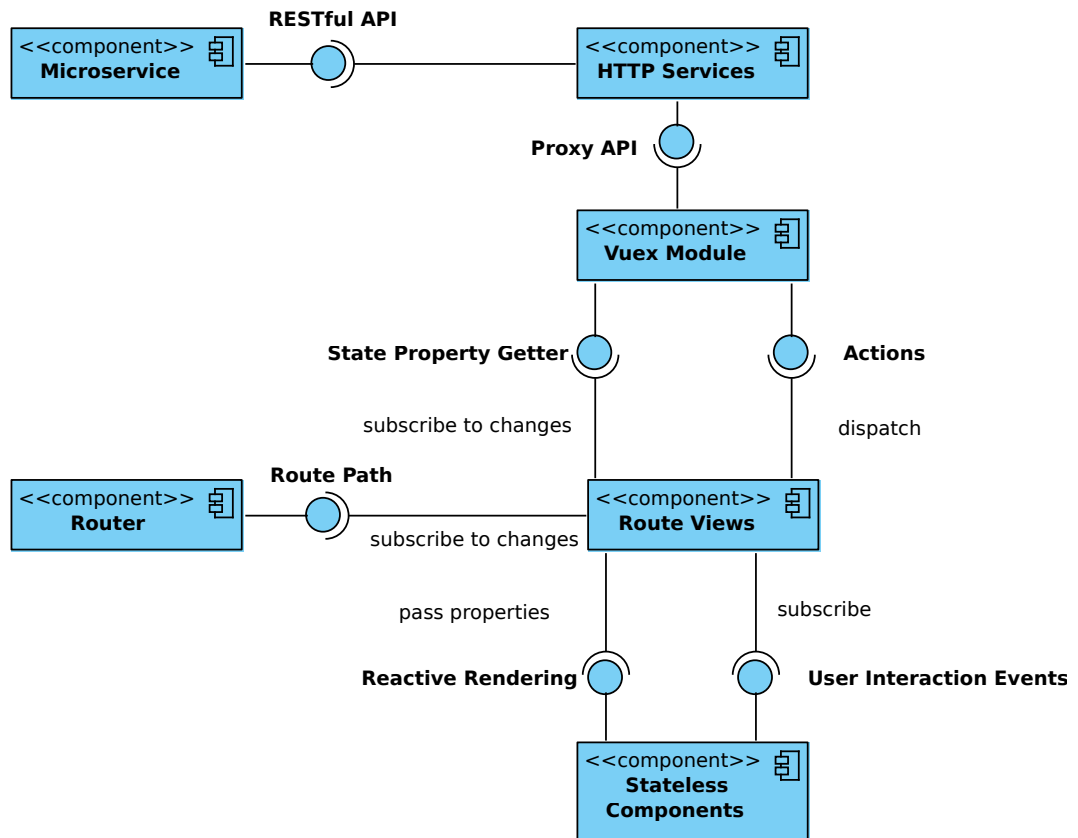


Figure 6.1. Abstract overview of the frontend components and their interaction.

identifiers on the detail page, e.g. we define the path `/details/:id/`, open the URL `/details/123/` and the id of value 123 is extracted. Route definition also contain callback functions which are applied if the location changes. We use this to allow each route to set a title displayed as the name for the tab in the web browser of the user. It can be either a string or a function which returns a title based on given information, i.e. we use this to show the users search term on the result list page.

The components responsible for handling a route are called *views* and defines the content for this page through their template. Figure 6.1 shows an abstract overview over the parts of the frontend, including the views and their relationship to the route. Views are delegating state information for all internally used components and initiate the data retrieval from external sources. This uses the information provided through the route and query parameters to begin the network communication and load the information necessary to display the page. Views are the only components which directly interact

6.1. Code Structure and its Conventions

with the central state management of Vuex. They dispatch actions and can start mutations which the Vuex store implements to change the central state of the application. While views are implemented as components and other non-view components could potentially implement the same interactivity, they are separated by convention. We enforce some separation by storing views in separate folders from the regular components. This allows future maintainers and developers to make assumptions on the behavior of the system based on the file path and might improve maintainability.

Stateless Components Regular components should be side-effect free and only render the page based on what data is explicitly passed to them, as described in Section 4.4.2. As shown in Figure 6.1, stateless components provide events for external interactions with elements of the component, mostly interactions of the user, e.g. clicking on an element. The handling of these events is done by the parent view as it could lead to state changes. While the use of events is possible, the frontend implementation passes callback functions from the view to the stateless components as it can throw exceptions when not provided correctly.

Generally, these components are not allowed to have an internal state. However, in some cases, state can not be avoided, e.g. if the component wraps the functionality of an external library which does not provide a stateless API. In these situations, the component is implemented in such a way that their behavior is consistent and predictable. Providing the same data as properties must always result in the same rendered elements for these components. Components are therefore not strictly side-effect free, but we try to emulate it in such special situations. The same also holds true to CSS style declarations used by the components. These are scoped to only be valid within exactly the component that declared them, but certain situations might require that the component provides declarations which are not scoped. This can occur if nested components are used and need stylistic adaptations to work in the context of the parent component, e.g. using a button within a table cell can require specific CSS properties to preserve the intended visual style of the nested button. However, scoped style declarations are not valid for nested components, as ensured through Vue.js and its handling of these styles. In these situations, great care is necessary to ensure that non-scoped declarations are only used in these specific situations, without changing the layout or style of other components on the page. They are also specifically documented in the source code as being unscoped declarations.

The implementation reuses several basic components in several scenarios, like loading indicators or buttons that show an info text when hovering with the mouse pointer. Reusing these generic components whenever possible, allows the application to quickly be adapted in all instances if the design or functionality needs to be altered. In our implementation we use the library *vue-strap*¹ which provides additional components like panels that are reused throughout the application.

¹<https://github.com/wffranco/vue-strap>

6. Implementing an Extensible and Maintainable Frontend Application

Using Vuex The state is managed through *Vuex modules* which define the part of the central state necessary for providing a single data based functionality. They group the data, mutations and actions that are dependent on one another. Additionally, they can implement getter functions for computed values based on the state. We choose the functionality of a module based on the data and operations surrounding this data are implemented within the same module. In our implementation all modules are forced to use namespacing which restricts access of this module to only its own data, mitigating unintended side effects with other modules. This allows to implement state-machine like behavior with all allowed operations and functions collected in a single file, allowing developers to understand every possible state change by inspecting this single file. In the frontend we use a module for the data collection (`researchEnvironment/collection`) which the research page uses to operate and two search related modules. These modules are the `search/input` module which handle the state of the search input and also contains the results and the `search/details` module stores the document that the user inspects using the detail page and its related documents.

Network Services The HTTP interactions are implemented through HTTP service scripts. These provide JavaScript-native APIs to load data from servers or upload queries to them. They add an abstraction layer over all server communication and provide asynchronous functions. These get necessary information for data retrieval passed as arguments and return the body of the HTTP response. For each of the microservices that the frontend uses, there is a corresponding HTTP service script which wraps all used functionality, following the proxy pattern. The microservices and their HTTP endpoints are configurable via environment variables and weaved into the application code at build time. All implemented network interactions require state changes, like visualizing the loading process or the responses, because of that the use of the HTTP services is limited to the actions in Vuex modules. Only actions in these modules are allowed to handle asynchronous state changes and then apply mutations to the state.

Normalizing Responses Through Models The data schema used by the search engine and therefore the schema of the stored documents might be changed by future usage of this prototype. In order to mitigate tight coupling to the data mapping of the search engine, we introduce a compatibility layer between the documents and search results from the search engine. Any component that renders documents is not allowed to access any of the JSON responses by the search engine directly. Instead we introduce model classes which can define allowed properties, therefore defining a strict access API. The classes also validate that required properties exist in their constructors. Instances of these models are wrapped around the original response JSON from the search engine, which split up into two data structures which are stored as `_data` and `_meta`. The `_data` object contains the raw information stored in the `_source` property from search engine responses. These are the fields of the metadata document which were harvested and imported into the search index. Descriptive information about the document is stored in the `_meta` property and provided or enriched

6.2. General Layout of the User Interface

through Elasticsearch. This includes the id and version of the document, the index the document is stored at, and other descriptive information. These properties exist to hide the raw information behind a clearly defined access API and some normalization. The models define property getters for accessing any of the information from their internal `_data` or `_meta` store. In JavaScript, property getters allow methods to be executed when accessing a key from an object, e.g. `myObject.myValue` can have a getter for `myValue`. Getters do not allow value assignments, making them read-only. We use this read-only approach as the frontend application does not provide the ability to edit any documents, so the updates can be blocked. This is implemented to throw errors if the components try to mutate the model instances, which we would consider faulty behavior and want to prevent. Additionally, this configures properties to be evaluated lazily, calculating only the values that the rendering actually requires to operate. However, there is also overhead when using a function to calculate a value instead of using a plain mutable property. This could be mitigated by applying memoization to the calculated properties in future implementations. In addition to providing a schema-independent API, having classes allows the usage of property validation for components that get documents or search results passed into them. This is because properties in Vue components can be defined to check whether the provided arguments are instances of specific classes.

The implementation of the frontend provides the model `SearchHit` which only contain the minimal set of properties to display the result list and the `MetadataDocument` class. `MetadataDocument` instances are fully loaded documents with all their properties. These are used on the detail and research page. All models inherit properties from an abstract `PartialMetadataDocument` that includes common getter functions. The third model is `SimilarDocumentMetadataDocument` which is the most basic child of the `PartialMetadataDocument` class, as these documents basically just require a title to be shown in a list with an id property. The models are further described in when describing the data flow through each page.

6.2 General Layout of the User Interface

The content for web pages is encapsulated by the application layout which spans all pages. Like the pages, the application wrapper is also a Vue component. This applies a consistent styling for all pages by importing the CSS framework *Bootstrap*² which provides default typographic styles, a grid system and additional reusable parts, e.g. navigation menus or button styles. Additionally the application component defines other global styles through unscoped declarations and it provides the layout of the whole frontend through its HTML template. This contains two parts, a global navigation accessible on all pages and the content of the page. The latter is a placeholder component, which is replaced with the component which is responsible for rendering the currently active route. Next to the

²<https://getbootstrap.com/>

6. Implementing an Extensible and Maintainable Frontend Application

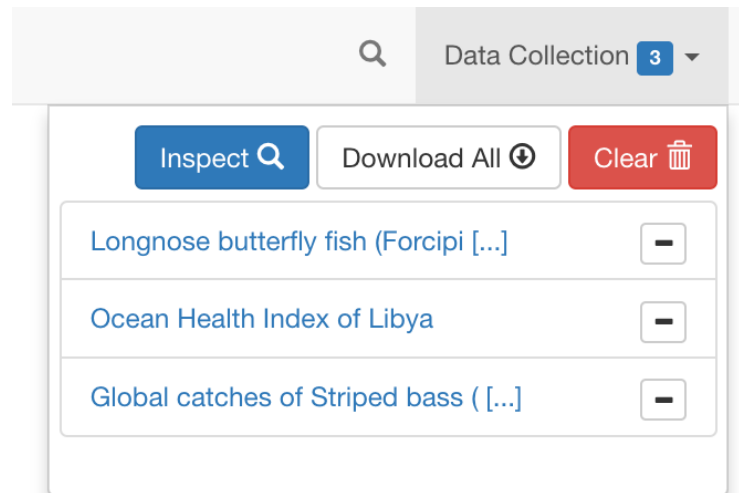


Figure 6.2. Screenshot of the opened drop-down menu showing the current data collection with buttons to remove documents, inspect them or download all their files. This menu is nested within the global navigation bar in the page header.

views, as mentioned in Section 6.1, the application component is the only one allowed to directly interact with the Vuex state. The `MainNavigation` is a stateless component that contains the navigation menu to the home and research page, a global search input and the data collection. The search input opens if the user clicks on an icon, giving access to the search field whenever desired on any page of the frontend.

Data Collection Quick Access Component As described in Section 4.4.3 the data collection should be a shopping-cart-like feature for users to select relevant documents to use for further research. Users can add documents to the collection using buttons on the document in the result list or its detail page. Generally, the research page provides the same functionality, but the `DataCollector` provides an always accessible quick access to managing the documents to inspect and the download functionality.

The `DataCollector` component implements this behavior and is located on the rightmost part of the navigation bar. A number badge next to the label shows how many documents are currently stored in the data collection. Clicking on the collection opens a drop-down block which shows the list of stored documents and allows for documents to be removed. Each document title is a link to the detail page of this document. The titles are limited to displaying a certain amount of characters to keep the string contained in a single row. Figure 6.2 shows the data `DataCollector` component as rendered in its open state. The drop-down provides an inspection button which links to the research page, allowing a more detailed inspection of the documents on a single page. Additionally it provides quick

6.3. Providing a Search User Interface

access to download all files contained in the collected documents. This download button is its own component (`MultiDownloader`) which builds a download URL that links to the zip-merger, which is described in Section 5.5.2. The request URL is constructed lazily when the button is clicked to reduce rendering overhead and will start the download of the zip file containing the files of all documents. Because the download is handled by the browser, it does not require any state as it behaves like a regular URL link. The button is disabled if no files are found in any of the documents in the collection, which is implemented as a computed property.

The state of the collection is handled by the Vuex module `researchEnvironment/collection` which is the same data source that the research page uses. We will see more about its state management when describing the research page in Section 6.5.

6.3 Providing a Search User Interface

The starting point for any user interaction in the frontend is through finding and exploring relevant research data. Because of this, the search feature is placed prominently on the home page of the frontend application. The home page provides a single search input for users to start the research data exploration. Typing text into this input dispatches the `setTerm` action from the `search/input` Vuex module which stores the state of the search input and its response objects. Following the conventions defined in Section 4.4.2, the action is dispatched by the home view, while the search input is its own stateless component. The `MainSearchForm` component is responsible for rendering the `SearchInput` component which are both mimicking the API for native search inputs, such that typing text into the inputs emits an input event. Submitting the search form redirects the search page with the value of the input as a query parameter in the URL, e.g. `/search?q=fishery`. This sets of the actual search request and is the basis for its functionality.

The functionality of the search page requires the rendering of results through components, in order to provide its service. However, a large part of its functionality is based around the interaction with the search engine through the Query API microservice and the underlying API of Elasticsearch. Therefore, we first handle search requests interactions and can see how the internal state of the search page is managed in Section 6.3.1. In Section 6.3.1 we then explore the request object used in the interaction with the search engine. Section 6.3.2 then shows how the results are rendered and what components are involved.

6.3.1 From Search Input to the Results

The search page of the application is responsible for rendering the results for a search query and has its own `/search` route and an example in Figure 6.3. When opened with a query parameter, e.g. `/search?q=fishery`, it automatically triggers the search request, otherwise the user has to first provide a search term and submit the request. Both ways trigger the send

6. Implementing an Extensible and Maintainable Frontend Application

action from the Vuex module `search/input` which triggers the search request to the Query API service.

Inside the module, the action sets the state to loading, which triggers the view to render a loading indicator, and then the current search query is extracted from the state. The action then passes the data to the library which handles the search engine interaction, called QueryAPI with its `sendSearchRequest` function. This function begins by using the search input from the user and constructing an Elasticsearch query with it. This enables the usage of term highlighting and faceting, as described in Section 6.3.1. After the query is constructed, it initiates the network request to the Query API microservice, searching for the metadata documents and then passing the response as its return value. When a request fails because of connectivity issues, the request will be retried automatically in case the Query API service is temporarily unavailable. The request uses the `metadata/search` endpoint of the Query API to search for the documents and supplies the request object as a JSON serialized query parameter using the GET HTTP Method. Using GET avoids preflight requests of the cross-origin restriction, as described in Section 3.2.3, and is semantically correct as it is a pure data retrieval and the browser is allowed to apply caching to the response. However, when sending JSON objects to a server, it is commonly done via a POST or PUT request method which can contain a request body, the backend service supports both methods.

After the search is concluded and the response available, the action extracts the relevant information and adds it to the state. This includes the list of documents, timing information, number of total results, aggregations/facets, and any other data which are necessary for rendering the search view, as seen in Figure 6.3. Following the convention defined in Section 6.1, we transform and normalize the documents resulting from a search into instances of the schema-independent `SearchHit` class. This class provides a read-only interface to only allow access to the most crucial information necessary for rendering the list of search results. These include a title, description and source URL, necessary for displaying the button to quickly access the source website. Title and description might also be available as highlighted strings, provided by Elasticsearch.

The Search Request Object and its Functionality In order to search for documents with the search engine, the query string from the search input must be preprocessed to form a valid Elasticsearch JSON query. A query transformation for the search provides this ability. The query syntax used in the input is a syntax that Elasticsearch natively provides, however the JSON object for requesting the data is enriched to use further functionality of Elasticsearch.

Listing 6.1 shows the query object used for requesting the results as displayed in Figure 6.3. We will now see how we can build requests such that we can provide our functionality based on this query. The `query` property with its nested `query_string` object provides the basic search functionality. The input from the user is appended here and will be parsed by Elasticsearch. In this example, the search string `longnose butterfly AND publisher.raw="SeaAroundUs"` searches for the keywords *longnose* or *butterfly* on documents that are

6.3. Providing a Search User Interface

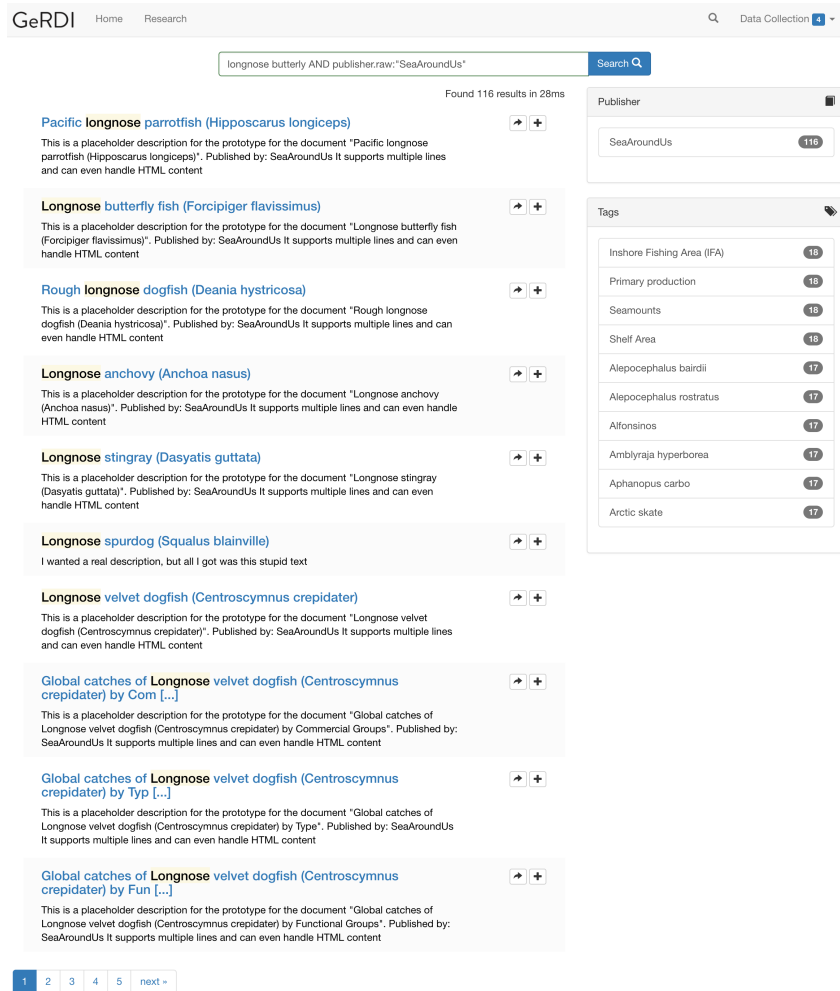


Figure 6.3. Screenshot of the search results displaying the interactive list of found metadata documents for a query.

6. Implementing an Extensible and Maintainable Frontend Application

```
1 {
2   "query": {
3     "query_string": {
4       "query": "longnose butterfly AND publisher.raw=\"SeaAroundUs\"",
5       "analyze_wildcard": true
6     }
7   },
8   "_source": [
9     "titles.title",
10    "weblinks"
11  ],
12  "sort": [
13    "_score",
14    "_uid"
15  ],
16  "highlight": {
17    "pre_tags": [ "@.highlight@" ],
18    "post_tags": [ "@/.highlight@" ],
19    "fields": {
20      "descriptions.description": {
21        "fragment_size": 500,
22        "no_match_size": 500,
23        "number_of_fragments": 1
24      },
25      "titles.title": {
26        "fragment_size": 500,
27        "no_match_size": 500,
28        "number_of_fragments": 1
29      }
30    }
31  },
32  "aggregations": {
33    "subjects.subject": {
34      "terms": {
35        "field": "subjects.subject"
36      }
37    },
38    "publisher.raw": {
39      "terms": {
40        "field": "publisher.raw"
41      }
42    }
43  }
44 }
```

Listing 6.1. JSON query sent to Elasticsearch for receiving the result from Figure 6.3.

6.3. Providing a Search User Interface

published by *SeaAroundUs*. The string `publisher.raw` is the path name of the property that has to be matched after flattening nested objects, as described in Section 3.4. The `analyze_wildcard` flag allows that the query strings can search for partial terms, like `long*` for any terms that begin with `long`. The `_source` definition provides a list of data that should be loaded from the original metadata document. Here we only require the title(s) and the `weblinks` to show a button to open the source webpage right in the list of results through the `VisitWebsite` action button. Explicitly loading these properties prohibits the search engine to load all properties from the documents, limiting resource usage and required network traffic. The possible properties that should be loaded are defined in conjunction with the `PartialDocument` model as required fields.

When Elasticsearch returns a list of results, the order of the returned documents is based on a calculated relevance score. However, this sorting is not stable when multiple results have the same score, leading to a random order. This hinders the sharing and bookmarking of search results by users because opening the same search URL again leads to a different result list. By using the `sort` property, we define a multi step sorting approach in which the relevance score has the highest priority when sorting, but a fallback is used when the score is equal. In this case, we sort the documents with the same score lexicographically by `id` (`_uid`) as it is the only property of the document guaranteed to be unique, thus forcing a stable result sorting.

In order to support highlighting of search terms on each result and relevant text extraction, we use the `highlight` API from Elasticsearch³. Defining the `highlight` property in the query object, allows us to declare which fields on the documents should be used for highlighting. We define that titles and description fields of the metadata documents should be processed for highlighting. Elasticsearch loads the values of these fields for each returned result and only returns the relevant sentences for the query. By setting `no_match_size` to 500, we force the results to include at least 500 characters of the values, even if no term was matched in the result. We also define the general maximum of returned text to only include up to 500 characters. Setting this limit in conjunction with a `number_of_fragments` of 1, we force the search engine to always return the field values, but with a maximum length. Especially when loading description texts which can be large, we would otherwise need to request the full description, but only display a short excerpt in the user interface. This can lead to significant reduction in network traffic for large text descriptions. We apply the same to the title of the document, but the displayed title of a document should always be displayed beginning from the first character, but still using the highlighting feature. In order to ensure this, we also load the full title via the `_source` definition, as mentioned above, and then merge the highlighting into the original string. The actual highlighting of strings is implemented by defining prefixes (`pre_tags`) and suffixes (`post_tags`), as shown in lines 17 and 18 of Listing 6.1, which are wrapped around words that matched the query. For more details on how the rendering of highlighted text is handled by the `HighlightedText` component see Section 6.3.2.

³<https://www.elastic.co/guide/en/elasticsearch/reference/5.4/search-request-highlighting.html>

6. Implementing an Extensible and Maintainable Frontend Application

The aggregations definition in the request object is used to get the data necessary to enable search faceting with the `FacetPanel` component, as visible in the sidebar of Figure 6.3. In the request object, the aggregations are defined with a key which is also used in the response to identify the data. Our implementation uses the name of the field we want to allow faceting for. The facets that are currently implemented use the `terms` aggregation which collects terms used in documents matching the query. The number of documents that match each term is then collected and the terms which match the most documents are returned. These terms are collected into a list of buckets in the response from Elasticsearch which is a tuple list containing the term and the number of documents. New term aggregations are configurable through the config file (`lib/config.js`) that contains a comma separated list of fields for which an aggregation should be computed. Additional aggregations will be rendered as facets automatically, but its name should be configured in the Search view component.

6.3.2 Displaying Search Results

The results page, as shown in Figure 6.3, shows the search query in an input and the response documents from the state in a list, linking to their respective detail pages. Above the result list, it is shown how many total documents match the provided query and how long the execution time of the search was. When number of results surpasses a configurable threshold, the results are shown on multiple pages which are navigatable using a pagination bar below the results. The page the user is currently viewing is reflected by adding the query parameter `page=X` to the URL of the web page with `X` being the page number. A sidebar to the right of the result list shows the facets for the data allowing users to further filter the results to specific properties, e.g. to only return results that contain the same tags or are from the same publisher. Each facet panel shows a list of possible value restrictions that can be applied. Each possible value provides an indicator signifying how many documents will be matched when applying the filter in order to help users understand if the additional filtering would lead to significant improvements.

The results are then rendered through the `SearchResultList` component which itself renders a `SearchResult` component for each search hit, displaying its title and description. Both texts can contain visually highlighted words if they matched the query using the `HighlightedText` component, for more info see below. The `SearchResult` component allows for components to be passed in that can provide a functionality in the context of this result on interaction. Currently, these actions are buttons to visit the website the data is located at the source repository (`ViewURL` in the metadata schema) and adding the document to the data collection. In order to be able to provide reusability for the `SearchResult` component, but with another set of actions, the actions are not declared within that component. Instead an array of objects is passed in, which defines the components to use and their callback function for handling user interactions. Components in these definitions must implement the same interface by allowing the `record` property for accepting instances of `PartialMetadataDocument` and an `action` property for the callback function. The component list is defined in the Search

6.3. Providing a Search User Interface

view and includes the components `VisitWebsite` and `AddToCollection`.

The sidebar contains a `FacetPanel` component for all aggregations that were requested and returned by Elasticsearch. Facets without any results are not displayed. The response from Elasticsearch contains only the internal name of the field the aggregation was based on. In order to make this human-readable, the `Search` view maps the name to a label and an icon displayed in the header section of each facets sidebar panel. In Figure 6.3, the sidebar contains facets for the publisher of the document and for filtering the results based on tags that the document might have.

The HighlightedText Component In order to show users the relevant part of the document that was used when matching the query, terms can be highlighted in the title and description of the document in the result list. When the search term is found within the description, only the sentences surrounding the term are displayed, the title however, is not cut in such a way to keep its readability. The `HighlightedText` component provides the rendering of the highlighted text. As described in Section 6.3.1, the highlighting of Elasticsearch surrounds terms that relate to the search query with `@.highlight.@` and `@/.highlight.@`. In order to apply the visual highlighting, we have to wrap these terms in a `<mark>` HTML tag, which can in theory be configured to be applied by the search engine. However, the Vue rendering framework does not allow to render HTML as Strings by default. HTML tags are always sanitized such that the characters `<` and `>` are replaced with `<` and `>`, which prevents browsers from evaluating these strings as HTML, rendering them as plain text. This is applied by default for the prevention of (Cross-Site-Scripting (XSS))⁴ attacks. Vue provides possibilities to explicitly avoid this restriction, but we not cannot safely apply these to information from metadata documents as they are harvested from external sources that might be compromised or not secured. In order to still be able to render the highlighting, we need to render strings as valid HTML while maintaining the security protection.

By using non-HTML delimiters (`@.highlight.@`), we can process titles and descriptions from the metadata document without destroying the highlighting tags. The complete removal of all HTML tags also helps with document descriptions that might contain tables or other HTML data which should not be displayed. In the next step we still replace the `>` and `<`, in case the removal of tags missed any tags. After that, the string can be considered sanitized and we replace `@.highlight.@` with `<mark>` and `@/.highlight.@` with `</mark>`. At last we enforce a maximum character limit and can safely render the resulting string without the XSS protection of Vue. The character limit is lower than the number returned by the search engine to accommodate the removal of HTML tags. Also the added `<mark>` tags are ignored when calculating the limit, as they will not get rendered and we want to limit the number of displayed characters, not just the underlying string.

⁴https://en.wikipedia.org/wiki/Cross-site_scripting

6. Implementing an Extensible and Maintainable Frontend Application

The screenshot shows a web interface for GeRDI. At the top, there is a navigation bar with 'Home' and 'Research' links, and a search bar with 'Data Collection' and a dropdown arrow. The main heading is 'Treaties and Conventions to which Lebanon (LBN) is a Member'. Below the heading, there are several panels:

- Description:** A placeholder text: 'This is a placeholder description for the prototype for the document "Treaties and Conventions to which Lebanon (LBN) is a Member". Published by: SeaAroundUs. It supports multiple lines and can even handle HTML content.'
- Geographic Information:** A map showing the location of Lebanon (LBN) in the Eastern Mediterranean region. The map includes labels for various countries and cities, such as Kibris, Gazimagusa, Limassol, and Beirut. A 'Street Map' dropdown is visible below the map.
- Tags:** A list of tags including 'Lebanon', 'Lebanon', 'Ministry of Agriculture', 'Ministry of Environment', 'Ministry of Agriculture', and 'LBN'.
- Authors:** A list of authors, including 'Peter Pansen (SeaAroundUs)'.
- Publication Info:** A section containing 'Publisher: SeaAroundUs', 'Publication Year: 2017', and 'Resource Type: Dataset > SeaAroundUs Data'.
- Web Links:** A section with a link to 'View: Treaties and Conventions to which Lebanon (LBN) is a Member'.
- Similar Documents:** A list of related documents, including 'Treaties and Conventions to which Lithuania (LTU) is a Member', 'Treaties and Conventions to which Jordan (JOR) is a Member', 'Treaties and Conventions to which India (IND) is a Member', 'Treaties and Conventions to which China (CHN) is a Member', and 'Treaties and Conventions to which Tunisia (TUN) is a Member'.

Figure 6.4. Screenshot of the detail page of a metadata document, displaying its available information through a set of panels.

6.4 Document Detail Page

The detail page, as shown in Figure 6.4, is responsible for visualizing all available metadata information for a single document. Additionally, this page is designed to provide further exploration of other documents by finding related documents, as described in Section 4.4.3. We generally achieve this by designing and implementing a convention for components that preview a document's metadata and providing new exploration possibilities through linking to other documents and search terms.

In order to display any of the information, we need to retrieve the data from the backend. The data retrieval and the state management is described in Section 6.4.1. In Section 6.4.3, we define an interface convention for all preview components in order to achieve the extensibility and maintainability goal described in Goal 2.1. We then see how the interface is used to implement each component throughout the section.

6.4.1 Retrieving the Document

When loading the `/details/<id>` route, the Detail view is loaded and triggers the data retrieval via the `id`. Following our state management conventions from Section 4.4.2, we use a Vuex module called `search/details` to load the documents into the state. This module exposes a single action for loading a document by `id`, called `get`. Whenever the route is initially called or changes while staying on the page, the action is called automatically. The `get` action begins by committing a `startQuery` mutation which sets a loading state and clears any cached responses. After that, the request is started by using the `getRecord(id)` function from the `QueryAPI` script. This uses `GET /metadata/document/<id>` from the Query API backend microservice to retrieve the document. In order to support identifiers that contain characters, like `/`, which are not URL-safe, they are URL-encoded before sending the request. The document is then resolved and transformed into an instance of the `MetadataDocument` class and written into the state. This allows the detail page to render the content that is only based on information on the `MetadataDocument`, while additional information can be loaded asynchronously.

In order to provide the similar document panels, as shown in the sidebar of the page Figure 6.4, we need to do asynchronous loading by starting another search request for the list of such documents. This is achieved within the `get` action, after the document is loaded, because the query requires data from the resolved document. The document is passed to the `getSimilarDocuments` function of the `QueryAPI` script which returns a list of documents. How the search for similar documents works, is described in more detail in Section 6.4.4. After resolving these documents, they are also applied to the state, allowing the list of similar documents to render.

6.4.2 Component Structure of the Detail Page

Figure 6.4 shows the Detail view for an example document. It consists of a header section which displays the main title of the document and action buttons. The title links to the website the metadata object was extracted from, called *view URL*, which is also linked to the more verbose *“Visit Website”* button on the right-hand side. The plus button adds the document to the data collection for further inspection with the research page. Clicking the button triggers the `addDocument` action from the `researchEnvironment/collection` Vuex module, like the search page allows. Downloading all files provided by the document is possible by reusing the `MultiDownloader` component from the data collection quick access as described in Section 6.2.

Below the title and buttons, the content is separated into two columns, as shown in Figure 6.4. The left side provides the content area for metadata that might require more spaces, while a sidebar on the right contains information that can be rendered with a low box height. The content area contains the descriptions stored in the document which is displayed through the `MultiDescriptionPreview` component. Below that we show an interactive map component (`GeolocationPreview`) when geographic information is stored in the document.

6. Implementing an Extensible and Maintainable Frontend Application

Tags on a document are displayed through the `SubjectsPreview` component, clickable for further exploration of the tag.

The sidebar on the right contains information about the data sources. At the top, the authors are listed through the `AuthorPreview` component, while the data of the publisher is rendered by the `PublisherPreview` component. The latter is shown as *Publication Info* in Figure 6.4. Below that panel we find the `WeblinksPreview` component, allowing users to visit websites that are related to this document. The last panel shows a list of documents that are similar to the one that is currently open. Its functionality is explained in detail in Section 6.4.4.

6.4.3 Metadata Preview Components

All components that render metadata from the document follow a specific interface. However, since the JavaScript language does not provide interfaces like some statically typed languages do, it is technically just a convention. Each component is a panel that accepts an optional heading and must accept an instance of the `MetadataDocument` class to extract the metadata it displays from. Since the `MetadataDocument` instances are read-only, the automatic rerendering of these components should work whenever the page loads a new document, because a new instance will be passed to the components. Vue will detect these changes and rerender the components accordingly. Without this guarantee, the specific fields of the document might be passed down separately.

PreviewPanel Component In order to implement a consistent style, the components use the `PreviewPanel` component which provides the basic markup and styling of all components as panels. These panels have a header section displaying a configurable label and a right-aligned icon. Each preview component sets a label as default value, allowing an override if necessary. In the following section we can see how each component achieves its data rendering based on the document.

Handling Descriptions Using the metadata schema of this prototype allows for multiple descriptions to be included in a single document. However, one of the descriptions will be treated as the main description such that it is always shown at the top. The main description is the one that does not have any additional `type` property as it is defined by a getter in the model class as the type defines alternative descriptions. The actual rendering of a panel that includes the text is provided by a nested `DocumentPreview` panel. Before rendering a description, all HTML line-breaks are replaced with regular line break characters (`\n`). The string is then stripped of any remaining HTML tags with the library `striptags`⁵, split up per line, and unnecessary spaces are trimmed. This allows for the rendering of each line as an HTML paragraph which handles the line breaking. Handling HTML tags might be

⁵<https://www.npmjs.com/package/striptags>

necessary, because metadata harvesters might extract data from webpages and not clean up the text.

Exploration through Metadata Providing source information about the data is crucial when working with research data and can also be used as a new starting point for additional exploration. Both types of source information are handled similarly in the sense that the displayed text is clickable in order to find more documents by the same author or publisher. In order to achieve this, we use an utility component `SearchLink` which generates links to the search page, automatically fills the search input to find these values, and starting the search. This is achieved by linking to the search page, calculating and appending the query parameter `?q=<searchQuery>`. For example, clicking on the publisher *SeaAroundUs* will search for `publisher:"SeaAroundUs"`.

The `AuthorPreview` uses the functionality to make the full name or its parts (given and family name) searchable. Additionally an author might belong to an organization which is behind the name in parenthesis and therefore also clickable. Since a document can have multiple authors, they are displayed as a list. The `PublicationPreview` component is not only providing information about the publisher, but also about its publication year, and in which category (*Resource Type*) the data is published.

Like the authors name, the tags (subjects in the metadata schema) are linked to the search page and therefore provide the ability to explore documents which have the same tag. The tags are rendered through the `SubjectsPreview` panel which enforces a tag length limit to avoid wrapping of tags across multiple lines. Another list of clickable metadata information is the list of *Weblinks* which are any number of links that the responsible author or even harvester service added to the document. These might include used sources that the researcher used when creating the data, data source links that the harvester service used when collecting the metadata, a related dataset, or any other links that can be relevant for users. The component can potentially support multiple types of links, however currently only the `ViewURL` type is implemented in the metadata schema.

Showing Geographical Metadata Metadata documents can contain geographical information about locations or areas that are relevant to understand the research data, e.g. where it was collected. We want to provide this information to the user of the frontend to help her or him with the relevancy evaluation of the data. The search engine stores and supports geographic information in the GeoJSON specification and as simple coordinate tuples. In order to use a unified interface for both types, the `MetadataDocument` model will automatically convert coordinate tuples into the GeoJSON format, allowing us to only support rendering of such. The location information is shown in an interactive map component, as shown in Figure 6.4. This component is the `GeolocationPreview` component which is a panel that wraps the `GeojsonMap` and the `MapProviderSelection`. The `GeojsonMap` is itself a wrapper component around the open source *Leaflet*⁶ library. Leaflet can render interactive maps and already

⁶<http://leafletjs.com/>

6. Implementing an Extensible and Maintainable Frontend Application

supports the GeoJSON specification as input data. When the component is rendered, it sets up several layers that, when combined, form the complete map. The base layer sets up the images, called *tiles*, which show the basic map and its geography. Tiles are loaded from the community-driven *OpenStreetMap* project⁷. Our frontend allows users to choose what tiles they want to use for the map through a menu that the `MapProviderSelection` component provides. This allows to switch between tile providers that are compatible⁸, we currently support the regular street map and topological tiles. Additional tile providers can be configured by adding a new instance of the `MapProvider` class with a name, tile URL, and a copyright notice to the `lib/MapProviders.js` file.

In addition to the tile layer, each GeoJSON object from the document must be applied to the map. Each object is its own layer and can either represent a single geographic point or span an area as a single or multiple intercepting polygons, such as the coastal waters of Lebanon in Figure 6.4. In order for users to be able to differentiate multiple areas, we give each layer an individual color. The color is calculated using a hashing function that receives the index of the GeoJSON object as input and returns a hexadecimal encoded color. After adding each layer, the map is rendered and displays the data as expected. We center and zoom the map such that it automatically displays all points and areas in the visible area. This is achieved by first creating a so-called `FeatureGroup` of all GeoJSON objects using Leaflet's API and then calculating a bounding box that wraps all points of the GeoJSONs.

6.4.4 Finding Similar Documents

The detail page is not only intended to display the metadata of a single document, but also provide means to enrich the information provided by the metadata and allow additional exploration of related data. In our prototype we provide the *Similar Documents* panel which is a proof-of-concept implementation of such functionality. This panel is not following the interface defined in Section 6.4.3 in only allowing a single `MetadataDocument` to get passed in. We need to break with this interface as the list of similar documents requires network interaction which is asynchronous and must be handled by a Vuex action. We consider the centralized state management as more important than the preview interface. The action that loads the similar documents is the `get` action of the `search/detail` module that also loads the document. After the instance of the `MetadataDocument` is resolved and the detail page is rendered, we can construct a search query from the loaded data and resolve the similar documents asynchronously. The resolved documents are converted into instances of a `SimilarMetadataDocument` model which inherits its properties from the `PartialMetadataDocument` class. All newly created instances are then added to the state and the `SimilarDocumentsPreview` component will render the titles of the documents instead of a loading indicator. Clicking on the title opens the detail page for the other document.

⁷<https://openstreetmap.org>

⁸https://wiki.openstreetmap.org/wiki/Tile_servers

Searching for Similar Documents We use the search engine to find similar documents based on data that the base document provides. The similarity of two documents is calculated by computing the intersection of their metadata and using its cardinality. This means that the more metadata information both documents share, the higher their similarity. However, in our implementation we do not treat every metadata information as equally important, but try to use semantically relevant fields. Using all fields might skew our results, e.g using all the words from the description might find documents that contain the same words which are not semantically relevant in descriptions like grammatical prepositions. In our prototypical implementation we use the list of tags (subjects) and the title of document, as they provide semantic value about a document and they are also available in our evaluation metadata documents. We can use the scoring functionality of Elasticsearch to implement the similarity calculation. The search results are sorted by a score that indicates how much of a given search query matches to a document in the result set, a higher score indicates higher relevancy of the result for the search. We use this as a similarity approximation by sending a search query that searches for documents that contain the same tags as the base document and that share words in the title. Listing 6.2 shows an excerpt of the query used, resulting in the list of similar documents shown in Figure 6.4. The query consists of several subqueries that can be used in order to identify a relevant document. The query for documents can contain the same tags (`subjects.subject` in the metadata schema). For this we extract the subjects from the instance of the `MetadataDocument` and add them to the `should` clause of the `bool` query. Queries in the `should` clause are not required to match, but increase the score of a document for each subquery that matches. Metadata document can contain any number of tags, because of that we add at most 50 tags to the search query. In addition to matching tags, we also use the title of the base document as a query for documents containing the same terms, as shown in line 19. By setting a `boost` value of 5, we indicate that we consider documents with similar names to be more relevant than matching keywords which have a boost of 1 by default. The specific boost value of 5 is just arbitrarily chosen and freely configurable in through the `lib/config.js` file. Searching for documents with the same title and keywords will almost always provide the base document as the most relevant result. To mitigate this, we use a `must_not` query that matches the id of the documents, excluding it from the results.

The `minimum_should_match` setting defines that at least one the `should` queries must match, otherwise a document will not be in the list of results. Without this setting, a base document that does not provide any similarity to other documents would match none of the provided queries and return find match all documents, as it is default behavior of Elasticsearch. The `size` setting limits the results to the five most similar documents and the `_source` setting limits the amount of data that the response documents contain. The source fields are fields that are required to be compatible with the `PartialMetadataDocument` class.

6. Implementing an Extensible and Maintainable Frontend Application

```
1 {
2   "query": {
3     "bool": {
4       "should": [
5         {
6           "term": {
7             "subjects.subject": "EEZ Area"
8           }
9         },
10        {
11          "term": {
12            "subjects.subject": "Shelf Area"
13          }
14        },
15        ...
16        {
17          "match": {
18            "titles.title": {
19              "query": "Catches By Data Layer in the Waters Lebanon",
20              "boost": 5
21            }
22          }
23        }
24      ],
25      "must_not": [
26        {
27          "term": {
28            "_id": "d5b1e52ded6acd4a7fec68acb72dd156"
29          }
30        }
31      ],
32      "minimum_should_match": 1
33    }
34  },
35  "size": 5,
36  "_source": [
37    "title",
38    "weblinks"
39  ]
40 }
```

Listing 6.2. Example query for finding similar documents by finding documents with the same tags and similar titles.

The screenshot shows the GeRDI Research Environment interface. At the top, there is a navigation bar with 'GeRDI', 'Home', and 'Research' links, and a search bar with 'Data Collection 3' items. Below the navigation bar, the main heading is 'Research Environment' with a subtitle 'Inspect multiple documents and preview their files.' To the right of the heading are two buttons: 'Download All Files' and 'Share Collection'.

The interface is divided into two main sections. On the left is a sidebar containing three document thumbnails. Each thumbnail has a title, a trash icon, and a 'File' button. The thumbnails are:

- 'Catches by EEZ by the Fleets of Algeria'
- 'Ocean Health Index of Syria'
- 'Longnose butterfly fish (Forcipiger flavissimus)'

The main panel on the right is titled 'Longnose butterfly fish (Forcipiger flavissimus)'. It contains several sections:

- Description:** A placeholder text: 'This is a placeholder description for the prototype for the document "Longnose butterfly fish (Forcipiger flavissimus)".' It also lists 'Published by: SeaAroundUs' and a note: 'It supports multiple lines and can even handle HTML content.'
- Authors:** Lists 'Peter Pansen (SeaAroundUs)'.
- Web Links:** Includes a link 'View: Longnose butterfly fish (Forcipiger flavissimus)'.
- Publication Info:** Lists 'Publisher: SeaAroundUs', 'Publication Year: 2017', and 'Resource Type: Dataset > SeaAroundUs Data'.

Figure 6.5. Screenshot of the research page.

6.5 Research Page

The research page allows users of the frontend to explore and evaluate multiple documents through a single interface. As described in the requirements from Section 4.4.3, it does not only provide access to metadata, but allows access to the scientific data that is stored on external repositories. The implemented prototype tries to provide the foundation to access and work with the data, specifically Goal 2.1 requires the preview of CSV data. It does so by allowing web browser based previews of the research data that can be extended in similar fashion with additional components. This aims to enable researchers to collect and evaluate research data interactively without first downloading and preprocessing the data. As a proof of concept, the research page allows users to preview structured data in the form of CSV and JSON files, while also allowing quick access to the metadata. Similar to the detail page, as described in Section 6.4, we also define conventions for interacting with research data that should provide the basis for additional functionality and maintainability.

Figure 6.5 shows the research page with three inspected documents. The top of the page contains quick access to download the files of all documents in the collection at once and next to it is the *Share Collection* button that allows researchers to share the current collection with other users.

The sidebar contains a panel (ResearchNavigation component) for each document that is

6. Implementing an Extensible and Maintainable Frontend Application

currently in the collection with a list of clickable information to inspect. On the right half of the page is the content area that renders the information the user wants to inspect. By default he or she can inspect a *Metadata* subpage for each document which reuses the components from the detail page, as described in Section 6.4.3, to show the information that is available through the *MetadataDocument* model. The *Geo-Data* subpage is extracted from the metadata list to keep the overall page height to a minimum, we also want to highlight geographic metadata, as it provides some interactivity like the file previews. Files that are contained in the document are also listed in the navigation panel. In addition to displaying the subpage navigation, each panel can also be used to remove its document from the panel via a button in the corner of the panel header. The header of each panel is also clickable to collapse the list, minimizing the panel to just show the header. This might be relevant for users with a large number of documents in the collection.

6.5.1 Accessing and Handling Files

When any subpage is accessed, the *inspectDocument* action is called from the *researchEnvironment/collection* Vuex module with the payload being the *MetadataDocument* instance. In the case of clicking on a file, the payload also contains information on which file should be shown. This also triggers the download process for the file's content. Downloading the file is possible by using the common data access interface of the *File Provider* microservice, as described in Section 5.5.1. Using this service allows the file download, because the cross origin restrictions of the browser can be avoided, otherwise the download would fail with a network exception.

Once the file is loaded, the content is added to the state into the *inspectedFile* property, but also added into an internal cache. The cache allows for users to reopen files quickly, enabling quick comparisons between files without redownloading them. A similar effect could also be achieved if the *File Provider* service would set HTTP headers that force the browser to cache the responses. If the browser's cache is used, the files will be stored by the browser, using the user's storage device instead of using the memory which is limited per browser window. However, this is not yet implemented in this thesis but could be added in the future with further evaluation that is outside of our scope.

Next to the files data, we also store the value of the *Content-Type* header from the response object into the state. Unfortunately, we currently only receive the response headers after the file was already downloaded, this can be mitigated by first loading the HTTP headers in a separated *HEAD* request. The *File Provider* service will respond to the request, but does not resolve the headers of the external file, which could be added in the future. When the *Content-Type* header is in the state, we can choose whether the file type is supported by a preview component to visualize the data. Some data types might need parsing. In our case we detect CSV files based on the *text/csv* content-type header. These files are then parsed using the *PapaParse*⁹ library which parses CSV files of different formats, e.g. it will

⁹<http://papaparse.com/>

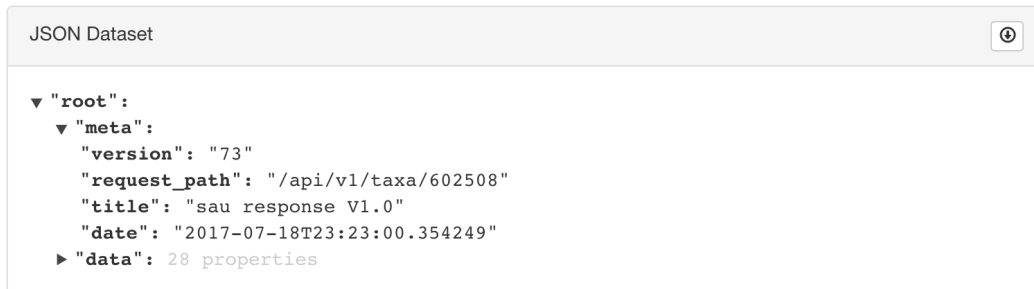


Figure 6.6. The FilePreview panel showing a JSON preview. The caret symbol allows to collapse and expand nested objects, allowing their traversal.

automatically detect if semicolons instead of commas were used as value delimiters. A caveat with the current implementation is that the browser might be blocked for a short period of time when large CSV files are parsed. Files that are too large to fit within the memory restrictions could also crash the window or even browser. This can be mitigated by using a WebWorker to handle the parsing in a separate process, however we do not implement this as it was challenging with the current setup of build tools that the frontend application uses. In addition to CSV, the frontend can also handle JSON files which do not require a parser library, because they are natively supported by the JavaScript environment. After parsing the supported types, the parsed data is stored in the state, rendering the page and the file preview.

6.5.2 File Preview Components

Similar to the metadata preview components as described in Section 6.4.3, file previews follow the convention that they receive one data property. However, while the metadata previews receive the `MetadataDocument` as input, the file previews receive the data in its preprocessed form, e.g. as parsed CSV file. Additionally, these components are only shown when the data is fully loaded as its type information is not available before. This allows for the guarantee that the component has access to the data, avoiding any duplication of code to handle a loading state. The components also only include the actual data preview, i.e. they do not provide the panel layout, further decreasing boilerplate code. The panel markup and styling is provided by the `FilePreview` component which wraps the `CsvPreview`, `JsonPreview`, and the trivial `TextPreview` components and switches them based on the data type. Additionally, the panel header contains the file label as its title, and a download button to directly access the source URL, bypassing the File Provider service. This might also be relevant in the case that the preview cannot handle the external data or the File Provider cannot access the data, e.g. because of a firewall. We also show the download button when the data type is not supported, therefore having no other preview available.

6. Implementing an Extensible and Maintainable Frontend Application

In addition to supporting the required CSV as structured data, we can also display JSON data with the `JsonPreview` component. Figure 6.6 shows how we display JSON data as an interactive preview. The tree-like structure of a JSON document can be used to navigate through the properties, by allowing the collapsing and expanding of nested objects. The implementation wraps the *Vue JSON Tree View*¹⁰ library component which provides this functionality. The `TextPreview` component is also similarly concise, as it just renders the text string while preserving line breaks.

Allowing large CSV Datasets CSV data is generally displayable as a table which HTML can natively render. However, when having a lot of rows, tables can decrease the render performance in the web browser, especially when initially adding the data and when scrolling the page. Libraries like *Clusterize*¹¹ mitigate this by only rendering the currently visible rows and reusing the existing DOM elements. With the `InfiniteTable` component, we wrap *Clusterize* and make it compatible with the component framework. *Clusterize* cannot be applied to tables directly, as table header columns are not compatible with this approach. In order to support this, we follow the example of *Clusterize* and use two tables, one which only contains the header columns and one for the data which is vertically scrollable. However, since the tables are independent, we need to synchronize them with user actions and the state of the data table. Table layouts in the browser are sized automatically based on the width of the data in the columns, which means that both tables can have different column widths. This requires two synchronizations: one for synchronizing the horizontal scrolling and one to set the width of the header columns to match the widths in the data table. The width synchronization is necessary whenever a user scrolls the data table vertically and the rows are replaced with a new chunk. A caveat is that the column width may change, whenever a chunk is rendered and the browser reapplies its table layout algorithm. This could be resolved by setting a fixed width of the data columns after the first chunk was rendered. Since the component wraps an instance of *Clusterize*, we also have to implement a function which is called before the component is removed from the DOM and destroy the *Clusterize* instance. We also support the situation that the component is still rendered, but the user clicks on another CSV file, which reuses the existing component. This update is passed onto *Clusterize* and the header sizes are recalculated.

When implementing the functionality, we noticed that the number of rendered frames by the browser dropped significantly whenever a new set of rows was rendered. We mitigated this by configuring the component to only render the data table once through the attribute `v-once` within the template. When this functionality is used, Vue will not register event listeners to the DOM elements and only traverse the DOM on first render. After that, the virtual DOM implementation of Vue does not compare with the state of the real DOM of the browser which is not necessary as the DOM will be manipulated and managed through *Clusterize*.

¹⁰<https://github.com/arvidkahl/vue-json-tree-view>

¹¹<https://clusterize.js.org/>

6.5.3 Managing the Data Collection

When selecting a collection of relevant data sets through the search or detail page, the list of documents is saved to the users machine through the browsers `localStorage` API which is a key-value store for strings. To achieve this, the list of ids from the documents is serialized to a JSON string and stored. Whenever the user reopens the web browser, the application triggers the `loadDocuments` action from the `researchEnvironment/collection` Vuex module. This action loads the list from the storage and triggers the `loadSources` action which is also used when users add new documents. It requests the complete `MetadataDocuments` from the Query API service from documents that are not yet in the state. When a document is already in the state, it will be loaded from a cache. Saving data to the `localStorage` also triggers a `storage` event in other tabs or windows of the web browser that have the same application open. We use this event to synchronize the state of all the tabs to have a shared document collection across all windows, allowing users to simultaneously inspect and add multiple documents.

The share functionality of the research page is using the same data flow. Whenever a user clicks the *Share Collection* button in the UI, a link to the research page is copied to the clipboard of the user. The URL is constructed dynamically with the current host from the web application and a the path contains the list of document ids separated by a comma, e.g. `/research/1234,7654/`. Using the router of the application, we defined the subroute to the research page to accept the list of ids. However, this route does not have a view component, instead we implement a `beforeEnter` function that is called whenever the route will be loaded. Within this function we trigger the `loadDocuments` actions with the list of ids extracted from the path and redirect to `/research`. The redirection will remove the document list from the path, avoiding the situation when the user adds or removes documents from the collection, but the URL still contains the old list. Another solution would be to have a read-only mode when a user opens a shared URL, only mutating the state when the user explicitly adds the documents to his or her own collection. With the current implementation, the users collection will be overwritten when opening a share URL of another persons data collection. Another caveat is that URLs have a length restriction of around 2000 characters which might be reached with a lot of documents. This can be resolved by implementing a new microservice which handles user sessions, allowing a collection to get an identifier which could be used instead of the list of document ids. Such a session service could also allow for collaborative collections that multiple users can add documents to, without the need to sending a new link to the collaborators when changing the collection.

Evaluation

In the following chapter, we evaluate whether the implementation fulfills the goals from Chapter 2 and the identified requirements from Chapter 4. We first verify that the implemented functionality of the microservices works as intended in Section 7.2. The functionality of the frontend is evaluated in Section 7.3. The evaluation of the frontend application not only includes a verification of its functionality but also whether the functionality provide a benefit for researchers. As an indication, we conduct an expert interview with two researchers in Section 7.3.3.

All evaluations follow the framework of the *Goal Question Metric paradigm (GQM)* [Basili and Rombach 1988]. The goals are defined in Chapter 2 and evaluated by first defining quantifiable questions. These will be answered by defining and using a metric for each component to evaluate. We follow this schema in the process of evaluating the parts of the system.

7.1 General Test Setup

Most evaluations require the same initial setup of services which we describe in this section. We show what versions are used, how they are started, and at what port they are running on afterwards. This is the required setup unless otherwise specified.

System Information The tests are run on a computer with an *Intel Core i7-4960HQ CPU 2.60GHz* CPU and *16 GB 1600 MHz DDR3* memory running *macOS Sierra (10.12.5)*. For the frontend tests, we require a web browser which is *Google Chrome* in version 59.0.3071.115. HTTP requests are send via the command line tool `curl` in version 7.54.0.

The services, except Elasticsearch, are using Node.js installed in version 7.7.2. For installing Node.js package dependencies we use `yarn`¹ in version 0.17.4.

Local Elasticsearch Instance In our tests, we use a locally installed version of Elasticsearch 5.4.0. For testing purposes we locally run Elasticsearch in version 5.4.0 using the docker image `docker.elastic.co/elasticsearch/elasticsearch:5.4.0` with the exposed port 9200. The index we generally use is called `datacite` and is created using the `create-index.sh` script from

¹<https://yarnpkg.com>

7. Evaluation

Appendix A6. Documents are added by using a HTTP POST request to `http://localhost:9200/datacite/seararoundus/<identifier of the document>/`.

Running JavaScript Services The Query API, Ingest API, File Provider, and File Merger are all build using the same conventions and frameworks. Before starting any of the services, its dependencies must be installed through the command `yarn` inside the directory of the service. After that the service can be started locally using the command `node ..`. The TCP port it binds to can be configured using the environment variable `PORT`. Environment variables can also be supplied by adding a `.env` file to the root of the service's directory. Each line contains the name of the variable followed by an an equal sign and the value. The frontend application also uses `yarn`, but does not respect the port configuration. In the following list we get into detail how each service must be configured.

Query API needs a link to the Elasticsearch instance. This is set via the environment variable `ELASTICSEARCH_BASE_URL=http://localhost:9200`. In our tests, the Query API is running on port 5000.

Ingest API also requires a link to the Elasticsearch. The service runs on port 6000.

File Provider requires the Query API endpoint to load the documents from. This is configured by setting `ELASTICSEARCH_BASE_URL=http://localhost:5000/metadata/document/`. Its port is 8000

File Merger requires the URL of the File Provider.

This is setup with `DOWNLOAD_SERVICE_BASE_URL=http://localhost:8000`. The port for the File Merger is 9000.

Frontend is different because it requires a build step. The necessary environment variable settings are `QUERY_API_ENDPOINT=http://localhost:8000/` and `DOWNLOAD_SERVICE_BASE_URL=http://localhost:8000`. After setting the variables, the frontend is build using `yarn run build`. This produces HTML and JavaScript files that we need to provide from an HTTP server. We use the `serve` command in 1.4.0 for this. This is installed via `yarn global add serve@1.4.0` and started via `serve -p dist/` in the directory of the frontend. This starts the frontend on `http://localhost:4000`.

7.2 Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

In the first part of our evaluation, we verify the functionality of the backend services.

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

7.2.1 Question: Can external repositories be integrated?

The architecture of the backend is designed to be compatible with a multitude of repositories. In this section we look at use cases with a proprietary data structure and how they can be integrated. However, these can merely be indicators, as the number of possible formats for providing metadata is potentially infinite.

Metric: Integration of SeaAroundUs The *SeaAroundUs* project provides data around the topics of sea biodiversity and fishery-based data. SeaAroundUs provides an HTTP API to retrieve metadata, but does not provide metadata schema standards like DataCite. In this evaluation we adapt the SeaAroundUs Harvester² provided by the GeRDI project to work with the Ingest API of the this prototype.

Base Functionality of the Harvester The basic implementation of the harvester uses the API of SeaAroundUs to retrieve the fishery data. However, SeaAroundUs does not use a concept of metadata documents that contain everything at once and does not provide any description texts, but rather very specific information about the data set, e.g. which type of fish is included in the data. The data can be accessed through a set of API endpoints that provide this information through a set of filters, e.g. catches by a specific nation. The harvester aggregates information from several endpoints and resolves relationships between the data. Documents are generated based on the filter used and metadata information, e.g. catches of a specific fish species by nation. It also generates a title for this document while most metadata is transformed into a keyword list.

An abstract harvester library³ provides the framework and general processing flow for harvesting metadata from external repositories. This involves three phases, beginning with the repository-specific harvesting implementation, in our case from SeaAroundUs. After this initial phase, the library transforms the generic document into a configurable JSON metadata schema. The last step is an upload of the documents to an Elasticsearch instance through its Bulk API.

Adapting the Harvester for the Ingest API In order to support the Ingest API service, we do not have to adapt the harvester itself, instead, we can achieve compatibility by adapting the underlying library. Fortunately, the metadata transformation phase supports a subset of the data schema that we use in this thesis through its DataCite-like export. Therefore, the necessary changes to support the Ingest API are limited to the Elasticsearch upload step. The Ingest API is functionally similar to the Bulk API of Elasticsearch, such that they both support newline delimited JSON serialization of documents as uploads. However, the Bulk API of Elasticsearch requires that each JSON document is prefixed by

²<https://code.gerdi-project.de/projects/HAR/repos/seararoundus>

³<https://code.gerdi-project.de/projects/HAR/repos/harvesterbaseLibrary/>

7. Evaluation

another JSON object that determines the index, type, and id of the document. The Ingest API does not support this because that information is inferred.

To support the upload to the Ingest API service instead of Elasticsearch, we introduced an additional boolean configuration flag called `use_ingest_api`. When set to `true`, the document prefix will be omitted from the upload. The URL used for uploading the documents was already configurable, however the type and index were automatically appended to the path of the URL. When the new configuration flag is used, this information will no longer be suffixed, instead the `import/batch` path is added to the URL. An additional adaptation is that the library will add the correct HTTP header when sending newline delimited JSON documents `application/x-ndjson` as defined in its specification instead of using `text/plain`. This allows for using the Ingest API as an export target for the harvester.

Verifying the Functionality In order to validate that the harvester is still allowing the import of documents, we harvest data once using the `use_ingest_api` flag and once without and then compare the results. For testing purposes we locally run Elasticsearch in version `5.4.0` using the docker image `docker.elastic.co/elasticsearch/elasticsearch:5.4.0` with the exposed port `9200`. We then create two indices with the same mapping. The mapping is created using the `create-index.sh <indexName>` script provided through A6. We create the index `seararoundus` for the data without the flag and `seararoundus-ingest` for documents that are uploaded through the Ingest API. The harvester service is run locally within a *GlassFish Server Open Source Edition 4.1.2* instance running on *Java SE Runtime Environment 1.8.0_45*. The base data was harvested only once and the raw data is stored to disk by sending `save_to_disk=true` and `read_from_disk=true` to the `PUT /resources/dev` endpoint of the harvester server. This verifies that the same base data is used for both imports and avoids external calls to SeaAroundUs. We run the Ingest API as described in Section 7.1.

Before each run, we restart the harvester service completely. After it started, we set the `read_from_disk` flags as described above and then configure the upload target by using a `PUT` request to `/resources/elasticsearch` with form URL encoded `url`, `index`, and `type`. Both the direct import into Elasticsearch and the import via the Ingest API are identical, except for these configurations. For the first run we set the `url` to link to the locally running Elasticsearch instance with both `index` and `type` being `seararoundus`. The full request body is `url=http%3A%2F%2Flocalhost%3A9200&index=seararoundus&type=seararoundus`. For the second run, we change the `url` to match the local URL of the Ingest API and add the `use_ingest_api=true` flag, resulting in the request body `url=http%3A%2F%2Flocalhost%3A6000&index=seararoundus-ingest&type=seararoundus&use_ingest_api=true`.

We can then start the harvest process via `POST /resources/harvest`. The status is checked using the corresponding `GET` request. Once the harvesting is finished, we start the conversion of the internal data structure into JSON documents through `POST /resources/datacite`. Using `GET` we can check for its status until it is finished. At last we start the upload process by using a `POST` request to the endpoint `/resources/datacite/submit` which blocks until the complete import is finished.

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

```
1 {
2   "size": 50,
3   "query": {
4     "function_score": {
5       "functions": [
6         {
7           "random_score": {
8             "seed": "this is my seed"
9           }
10        }
11      ]
12    }
13  },
14  "_source": true
15 }
```

Listing 7.1. Elasticsearch query to load 50 random documents

After both runs are completed, we have two indices which we need to compare. We do this with the `compare-indices.js` script Appendix A7. This script first checks that the number of documents is matching in the index `searoundus` and `searoundus-ingest`, which is the case with 43821 documents. We then sample 50 random documents from the base index `searoundus` with the query from Listing 7.1. The value from the `identifier` field is then extracted for each document and the list of ids is queried from the second search index `searoundus-ingest` with the query from Listing 7.2. We then verify that the number of results is 50 which is the case.

The content of the files should also be evaluated, but the harvester does not provide the same results when it is run multiple times. The harvester seems to not use the same identifier for multiple documents. When a different document with the same identifier is uploaded, the first document is overwritten. We verified this by running the harvester twice without the Ingest API flag and manually compared sample documents. This hinders further checks whether the correct documents were written to the search index.

Results In this evaluation we adapted the `SeaAroundUs` harvester from the `GeRDI` project to work against the Ingest API we introduced in this thesis. We have shown that the Ingest API is able to import documents from an external repository, here `SeaAroundUs`, by verifying that the same number of documents can be imported independently of whether the Ingest API was used. However, we cannot definitively say that the content of documents is written to the search index in both cases, because of non-deterministic behavior of the `GeRDI` harvester. While we have shown that the import of `SeaAroundUs` is possible, we could potentially add more data sources that are compatible to the `GeRDI` harvester, because we adapted the underlying library used by all harvesters to be compatible with our system. This however, requires further validation.

7. Evaluation

```
1 {
2   "size": 1,
3   "query": {
4     "constant_score": {
5       "filter": {
6         "terms": {
7           "identifier": [
8             "<documentId1>",
9             ...
10          ]
11        }
12      }
13    }
14  }
15 }
```

Listing 7.2. Elasticsearch query to load documents for a list of identifiers

7.2.2 Question: Does the File Provider allow file downloads from external services?

As described in Section 4.4.5, the displaying of file data in the browser requires the File Provider. We verify that the service provides this functionality as a basis for further evaluations.

Metric: File can be downloaded by document and file identifier In order to verify that files can be accessed, we must first verify that the URL schema can be used to retrieve the correct file from the store. We don't just want to ensure that files can be retrieved at all with an HTTP GET request to the download endpoint of the File Provider, but also that it is the correct file. For this test, we set up a local HTTP server using the library `serve-static-throttle`⁴ in version *1.0.2*. This server allows the download of static files and that provides two files with known checksums, its throttling capabilities are not relevant for this tests, because the download speed is not measured. These files are `hello_world.txt` with the checksum `c897d1410af8f2c74fba11b1db511e9e` and `hello_space.txt` with checksum `915075b72060c7ca9b94aca48cee61f0`.

The tests will be run on the same system that executes an instance of the File Provider and the HTTP server. An instance of the File Provider service is as described in Section 7.1.

For our tests, we add an example metadata document shown in Listing 7.3 to the search engine. This document provides several file identifiers we use for multiple evaluations in this chapter. For the purpose of this test, we will use the files with the identifiers *text* and *file with spaces*. The second identifier contains a space character to verify that the File Provider treats URL-encoded identifier strings correctly, as spaces need to be encoded to form valid URLs. We start the download of the first file with the program `curl` and

⁴<https://www.npmjs.com/package/serve-static-throttle>

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

```
1 {
2   "identifier": "filetest",
3   "titles": [{ "title": "File Test" }],
4   "files": [
5     {
6       "identifier": "text",
7       "label": "Hello World",
8       "url": "http://localhost:3000/hello_world.txt"
9     },
10    {
11      "identifier": "file with spaces",
12      "label": "Hello Space",
13      "url": "http://localhost:3000/hello_space.txt"
14    },
15    {
16      "identifier": "json",
17      "label": "JSON Dataset",
18      "url": "http://localhost:3000/test.json"
19    },
20    {
21      "identifier": "csv1",
22      "label": "CSV Dataset (Comma)",
23      "url": "http://localhost:3000/csv1.csv",
24      "type": "application/csv"
25    },
26    {
27      "identifier": "csv2",
28      "label": "CSV Dataset (Semicolon)",
29      "url": "http://localhost:3000/csv2.csv",
30      "type": "application/csv"
31    },
32    {
33      "identifier": "csv3",
34      "label": "CSV Dataset (Spaces)",
35      "url": "http://localhost:3000/csv3.csv",
36      "type": "application/csv"
37    },
38    {
39      "identifier": "zip",
40      "label": "ZIP Dataset",
41      "url": "http://localhost:3000/test.zip",
42      "type": "application/zip"
43    },
44    {
45      "identifier": "large",
46      "label": "Huge Random Dataset",
47      "url": "http://other-service/largefile.bin",
48      "type": "application/binary"
49    }
50  ]
51 }
```

Listing 7.3. File access evaluation document

7. Evaluation

pass it to the `md5` program to calculate the download checksum, which returns the hash `c897d1410af8f2c74fba11b1db511e9e`. The full command is `curl http://localhost:8000/download/filetest/text | md5`

For the second test, we use the *file with spaces* file identifier to test the URL encoding and also whether the File Provider returns the correct file based on the identifier. The URL is constructed with the encoded version of the identifier (`file%20with%20spaces`). The download of that file results in the MD5 hash `915075b72060c7ca9b94aca48cee61f0`. Both checksums match the initially provided files, proving that the expected files were returned correctly by the File Provider.

7.2.3 Question: Does the File Provider restrict the throughput of the file download?

The File Provider acts as proxy between a client and an external repository. We want to verify that this happens with minimal impact to the possible throughput. However, we limit the evaluation to whether the software itself provides a bottleneck. In real usage scenarios is the throughput largely dependent on the network infrastructure and its load.

Metric: Download Timings We want to verify that the download of files through the File Provider only causes a minimal amount of overhead for the users when compared to a download directly from the source. This can be achieved by instructing the source file server to provide a file of known size at a constant transfer rate. We then measure the download timings regarding the time until the HTTP headers are resolved and the time until the complete file has finished downloading. The HTTP headers are the first data sent from the server and are sent in a block. Once the headers are sent, the bytes of the file are transmitted to the client. The File Provider can only send headers after the metadata document is retrieved and the file information is extracted, making the header timing an indicator for the time it takes to load this information. Additionally, once a web browser loaded the headers, it can display a download indicator. This introduces additional latency over loading files directly. After the headers are loaded, the File Provider will begin the download and stream the data, for which the connection to the external server has to be established.

We want to focus on files that were passed from the source server to the client directly, not measure the network latencies between the server and File Provider. To minimize network latency factors, we test everything on a single machine. This machine executes the downloading client (`curl`), the File Provider and the source server. The source server is implemented as a static file HTTP server which allows setting the maximum download speed. We use the Node.js module `serve-static-throttle` in version *1.0.2* to achieve this. The metadata document used for this test is the `filetest` document shown in Listing 7.3. We use the file with the identifier `zip`, linking to a file that is exactly 100 megabytes in size. By throttling the download speed, we can check if the measured download speed matches

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

the throttle setting. The latency of loading the document from the store is not measured in this evaluation. In order to accurately measure its impact, we would require statistically relevant test of Elasticsearch's performance when loading a single document. Having a constant time for this latency, allows us to calculate the throughput of the isolated File Provider more accurately. To achieve this, we modify the File Provider to return the `filetest` document synchronously, bypassing the external storage.

Executing the Tests The test is executed with a throttling setting of 10 MB per second. For testing we use `curl` with the following command to output the average download speed: `curl -o /dev/null -w "%{download_speed},%{time_total}" http://localhost:8000/download/filetest/zip`

The download is executed 10 times and results in an average download speed of 10.08MBps and a total download time of 9.92s . All values are rounded arithmetically to two decimal places. To verify the throttling we use the same approach to directly download the file from the throttled server and receive the same (rounded) throughput 10.08 and total time of 9.91s . We also verify that the throttling is not enforced by external factors by reconfiguring the source server to throttle at up to 5 megabytes per second, resulting in 5.02MBps and 19.92s of total time for both cases.

At last, we want to verify whether the File Provider might introduce a limit to the download speed. To test this, we can run the same test, but without any throttling. For the test we use a larger file of 10gb , resulting in 456.39MBps with a total download time of 22.88s for accessing the file via the File Provider. When accessed directly from the source server, we measured 585.38MBps with an average download time of 17.61s .

Conclusion In this evaluation, we measured the time and throughput of the file download when accessing files through the File Provider and when loading directly from the source. By comparing the measurements for both cases we can see that the measurements are almost identical. With a large file the usage of the File Provider decreased the total throughput by 22.04% (arithmetically rounded to two decimal digits). However the throughput of around 456.39MBps is still higher than the theoretical throughput of 1 gigabit ethernet connections which is 125MBps . The reasons for this upper bound are unknown, but might be limited to the overall limit of socket throughput on Unix machines.

7.2.4 Question: Does the File Merger service bundle downloads from external sources?

In order to support the frontend functionality to download multiple files, we use the File Merger. Its basic functionality must be verified.

Metric: The zip server can/cannot download multiple files In order to test whether the File Merger can bundle files, we need to add a test document to the system. For testing purposes, we reuse the `filetest` document from previous evaluations, as shown in Listing 7.3.

7. Evaluation

```
1  [{
2    "fileIdentifier": "csv1",
3    "documentId": "filetest"
4  }, {
5    "fileIdentifier": "json",
6    "documentId": "filetest"
7  }, {
8    "fileIdentifier": "text-2",
9    "documentId": "filetest2"
10 }]
```

Listing 7.4. File merge request for the zip service

To verify that the files can be extracted from multiple documents, we also create a copy of `filetest`, named `filetest2`. The title of the document and the file identifiers within the the copy are appended with the suffix `(-2)`. The files are accessible through a local HTTP server and the File Provider is running on the same machine and has access to the files. The File Merger service is setup as described in Section 7.1.

In our test, we download the files `json` and `csv1` from the document `filetest`. From `filetest2` we also add the file with the identifier `text` to `files` array. The JSON request that identifies which files should be downloaded is shown in Listing 7.4. In order to send the request, we open `http://localhost:9000/merge?request=<JSON>` in Chrome, with `<JSON>` being URL-encoded. While the endpoint also supports using POST with the JSON as body, we use GET to use the same endpoint the frontend application uses internally. The resulting URL is then called from a web browser and starts the download of the `download.zip` file.

Results After accessing the URL, the web browser starts downloading the zip files. The resulting file is `download.zip` is extracted and reveals the file `_download_sources.txt`. Additionally, it contains the directories *File Test* and *File Test 2*. The `_download_sources.txt` contains the original URLs of the file and its content is shown in Listing 7.5, matching the behavior defined in Section 5.5.2. Files from `filetest` are located within the *File Test* directory, named `csv1.json` and `test.json`. The directory *File Test 2* contains the file `hello_world.txt` from the metadata document `filetest2`. In order to verify that the files are identical to the original files, we calculate md5 checksums for all the files from the extracted `download.zip` and compare them to the checksums of the files from the source HTTP server. The checksums match, proving that the correct files were downloaded.

7.2.5 Question: Do the file services stream data to the client?

Metric: Memory usage In Section 5.5 we state that the implementation of the File Provider and File Merger are both minimizing the memory usage with their stream based processing. We can verify this by downloading a large file and observe whether the memory usage of both services is constant. In our test, we generate a file containing random bytes with the

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

```
1 File Test:
2 csv1.csv => http://localhost:3000/csv1.csv
3 test.json => http://localhost:3000/test.json
4
5 File Test 2:
6 hello_world.txt => http://localhost:3000/hello_world.txt
```

Listing 7.5. The `_download_sources` file contained in the bundled `download.zip` file

```
1 [{
2   "documentId": "filetest",
3   "fileIdentifier": "large"
4 }]
```

Listing 7.6. The zip merge request for compressing a large file.

size of 8.5gb. The file is generated via `dd if=/dev/urandom of=largefile.bin bs=1048576 count=8912896`. The file is then made accessible via a static file HTTP server, like in the other tests of this chapter. However, here the services were deployed on two separate virtual machines running on the same host.

We use the metadata document `filetest` with the file with the identifier `large` to load the file URL from. The file download is started by using an HTTP GET request to `/download/filetest/large` for the File Provider. For the zip merge we use the `/merge` endpoint with the request from Listing 7.6. We only access a single file to avoid the automatic parallel download of the current implementation and only test the base case. Downloading multiple files at once uses buffers to download the data into, which would make it impossible to verify whether the base case is only working on data streams. Before the test is started, we download arbitrary files as a warmup phase. This is done to decrease memory fluctuations due to the just in time compilation of Node.js.

Measuring Memory Usage with Rancher Rancher provides runtime information of containers via a websocket stream. In order to use this, we use Ranchers API to load the container statistics endpoint for a specific service. This resolves a URL and token to be used when establishing a websocket connection. Once the subscription is implemented, we receive the latest state of the resource usage in a regular interval, grouped by a unique id for each container. In order to track these changes, we provide a `logdumper.js` script that writes the sources usage into a CSV file. The script is included in Appendix A7.

Results Figure 7.1 shows the memory usage of the File Provider service for the whole timespan of downloading the large file, including some seconds before and after to show the base usage of the service. The raw data is plotted in light gray, while the bold represents

7. Evaluation

a regression using the loess function⁵ in the *R* statistical computing environment. We can see that the memory usage is fluctuating around 60MB, starting from a baseline of around 40MB. The deviations from the average might result from the download throughput of around 25MBps that is garbage collected in regular intervals. The spike in memory usage around the 50s mark, however, is unclear. This could be caused by network fluctuations, leading to short lived buffers while the response chunks are sent to the client. However, these increases in memory usage are still not near enough to the size of the file and are still freed up after several seconds. This indicates that the File Provider does indeed stream the information without downloading the file first, but rather passes on the data directly to the client.

The memory usage of the File Merger is shown in Figure 7.2 and provides a more constant memory usage. While the compressed zip is potentially smaller than the source file, the resulting compressed file cannot be so significantly. This is because the source file stream only contained pseudo-random bytes which have a high entropy and don't provide repeating artifacts that can be compressed. However, there is slight upwards trend of the overall memory consumption. This might indicate a memory leak, but is potentially the growing compression lookup table used by the zip algorithm. However, this is speculation and should be further inspected, especially whether the memory is correctly garbage collected.

7.2.6 Question: Is the architecture scalable to support high load scenarios?

The infrastructure is designed to be scalable to a large set of users interacting with the system. In order to evaluate this, we have to evaluate to what extent the provided services are scalable. We differentiate between the stateful search engine and stateless services, such as the file services. The scalability of the message broker is not evaluated, as it is only a proof-of-concept implementation. However, according to its documentation, the message broker software Apache Kafka provides distribution to multiple servers as a core functionality [Kafka Introduction].

Metric: Statelessness of Services Services that do not contain any relevant and mutable state that is necessary to respond to requests can be scaled horizontally, assuming the underlying infrastructure allows it [Amazon Web Services 2016]. Statelessness refers to the any incoming requests being handled independently, without any sessions or altering of internal state of the used service which requires synchronization. However, if they are a bottleneck and depend on other services, adding new instances can only handle more incoming requests if the underlying service can handle the additional load. This applies to the File Provider, Query API, and Ingest API as their scalability is bound to the

⁵<https://stat.ethz.ch/R-manual/R-devel/Library/stats/html/loess.html>

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

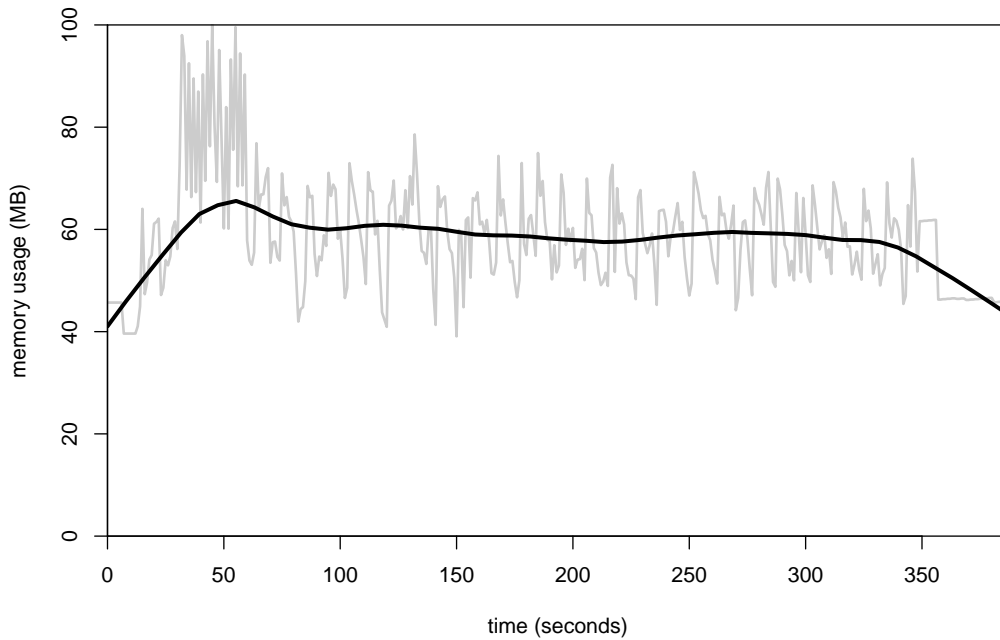


Figure 7.1. Memory usage of the File Provider while streaming multiple gigabytes to a requester

Elasticsearch on which they depend to answer queries. The scalability of Elasticsearch will be evaluated in the next part of this chapter.

Assuming the scalability of the underlying search engine, we can freely scale the query and Ingest API horizontally and consider them stateless. The File Provider service depends on the Query API to retrieve the metadata document, but then only streams file downloads from source repositories. The metadata document is resolved independently for each request and so is the file, making the File Provider stateless. The File Merger uses the File Provider for each request and combines the files, but the compression is also independently handled for each request, without side-effects to others. Therefore, the File Merger is also stateless by design.

Metric: Load test of the search engine In order to verify the scalability aspects of the underlying search engine, we simulate an increasing number of concurrent users of the system.

7. Evaluation

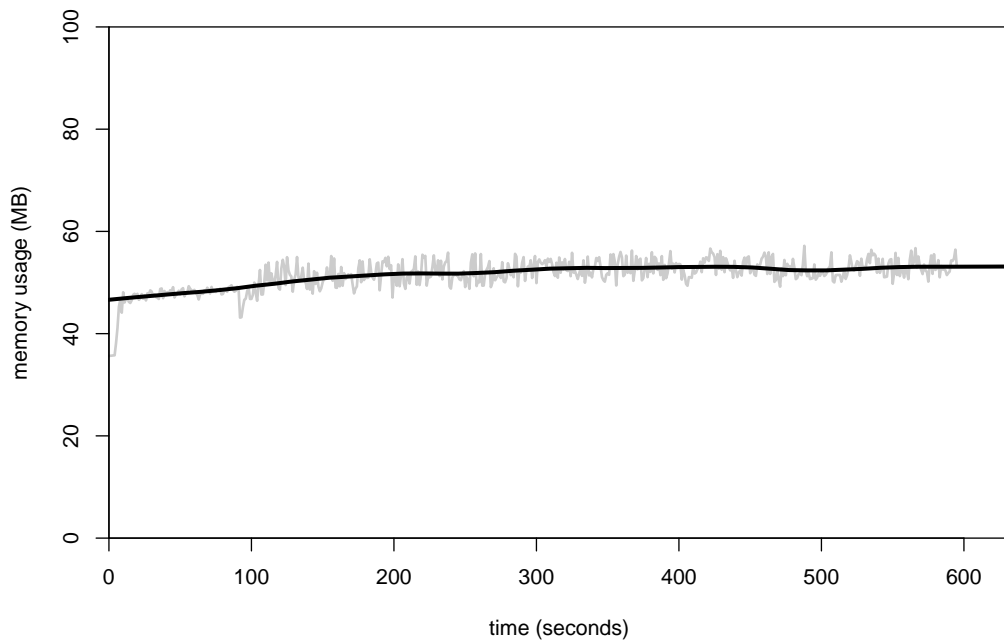


Figure 7.2. Memory usage of the zip while streaming and compressing a large file.

Environment Setup To test the search engine, we have to provide a scalable infrastructure. For this, we have two physical machines, both using a 32 core *Intel Xeon CPU E5-2650 0 2.00GHz* and 126GB of DDR3 RAM with 1600 MHz. They are connected via a *Extreme Network Summit x350-48t* gigabit switch with a bandwidth of 94.5Mbit/s, measured with the command line tool *iperf3*. One machine is used to run the data nodes of the search engine and the other runs the test script, Query API, load balancer and the Elasticsearch master node. They are necessary to simulate a search infrastructure as it might be found in a production environment. We refer to this machine as *load test node* and configure the *Required Container Labels* to `service.type=loadtest`, only allowing specifically configured service to be deployed. For deploying and encapsulating the search data nodes, we use four virtual machines on the first physical node with a bridged network adapter running within *VirtualBox version 5.1.22-115126 Debian jessie*. Two of those machines have eight cores and 20GB of memory assigned, we will reference them as *base hosts*. The other two machines use four cores with 10GB of RAM, we will refer to them as *scalability hosts*.

These four virtual machines and the second physical machine are configured as hosts in a *Rancher Cattle* environment with version 1.6.2. We use host labels and container labels

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

to have fine-grained control over which host is used for which service. All services for running the load tests are grouped within a single stack, except for services that are infrastructure services which Rancher automatically deploys and manages. The service setup is provided as deployable `docker-compose.yml` and `rancher-compose.yml` files in Appendix A7. The configuration consists of a service for the Elasticsearch instances running the image `docker.elastic.co/elasticsearch/elasticsearch:5.4.0`. In order to mitigate the effects of the difference in memory of the base and scalability hosts, we configure the JVM to use four gigabyte of RAM by setting the environment variable `ES_JAVA_OPTS` to `-Xms4g -Xmx4g`. The cpu cores are pinned to only use the first four cores. However, the number of cores reported to the Docker container is still eight on the base hosts, which Elasticsearch uses to calculate the number of internal worker threads [Elastic Reference]. Unfortunately the official Elasticsearch Docker image does not allow overriding the automatic detection of cores, but we can override the number of cores in the `thread_pool` configuration. To achieve this, we set the number of workers to 7 based on the formula for the default value $(N * 3) / 2 + 1$ with N being the number of cores which we want to emulate to be 4 cores [Elastic Reference]. The configuration is done via the environment variable `thread_pool.search.size`.

In order to have a more realistic scenario, we use an instance of the Query API as the endpoint for the load test. This is linked to a Rancher load balancer as `elasticsearch` which is configured to pass traffic on port 9200 to the data nodes of the Elasticsearch cluster. We then configure the Query API service to use the linked load balancer by setting the environment variable `ELASTICSEARCH_BASE_URL` to `http://elasticsearch:9200`.

For the base case and first test, we deploy one Elasticsearch node on the base host virtual machines. We ensure this by specifying a Rancher service scheduling rule that the host must have a label `use.search=true` which we then add to the base hosts. Additionally, we set the rule that the service of the same name must not be deployed on the same host, forcing that only one instance can be deployed. After setting up the service, we use the `create-index.sh` script from Appendix A6, and upload the test metadata documents to the `datacite` index. We then set `number_of_replicas` to 1 via a PUT request to the `datacite/_settings` endpoint of Elasticsearch with the request from Listing 7.7. Elasticsearch will then automatically copy the search index data while the Elasticsearch nodes are in the state `unhealthy`. This gives us a total of two index replicas, because there is always the primary version of the index and the copy. In the following paragraph, we always refer to the total number of replicas. We start with two replicas, because the automatically created monitoring data indexes of Elasticsearch require two replicas.

Using Apache jMeter for Load Generation For generating the load, we use *Apache jMeter* in version 3.2. Through various iterations, we noticed that simulating a large set of users with a single instance of jMeter can lead to errors and a high variance in HTTP response times. To mitigate this, we define a jMeter server service in Rancher and run eight instances using `jmeter-server` on the load test node. Additionally, we need a `jmeter-master` service that controls these instances and aggregates the measurements. Both Rancher

7. Evaluation

```
1 {  
2   "index": {  
3     "number_of_replicas": 3  
4   }  
5 }
```

Listing 7.7. Elasticsearch settings to configure the number of replicas an index should have

services are using a self-defined Docker image that includes the testing scenario and generates the data used for the requests. The Dockerfile to build the image and the jMeter testing plan are included in Appendix A7. Each instance is allowed to use 10GB of memory for the JVM. The `jmeter-server` Rancher service is configured to reuse IP addresses when restarting containers, because we explicitly need to list these IPs in the environment variable `MASTER_SERVER_NODES` of the `jmeter-master` service. Linking both services, allows full duplex communication between all instances.

Generating Test Requests When running the test, we want to simulate search requests that are similar to what real users could search for. Unfortunately, there is no such logged data available, instead we generate a set of requests from SeaAroundUs and FaoStat metadata documents, which we harvested as described in Section 7.2.1.

The requests use the same functionality as the frontend, including the same settings for term highlighting and aggregations. Overall, for each simulated user we generate 2700 queries which consists of single term queries for 100 randomly selected country names and 100 of the most common terms from the example data which were extracted with the `significantQTypes` aggregation⁶ from Listing 7.8. In order to simulate users that use facets to narrow down their search, we suffix a random country name with `AND subjects.subject="<term>"`, where `<term>` is one of the significant terms. We use 100 of those faceting queries in our test data. To simulate multiple search terms, we use the titles of 1000 random documents which we extract with the query Listing 7.1. The same documents are also used to extract 400 tags to simulate users that click on tags of other documents in the detail view. Additionally, we also add 1000 queries for similar documents to each randomly loaded document. We do this by extracting their title and tags which are used with same requests the frontend sends, as described in Section 6.4.4.

Each simulated user in the jMeter test scenario uses a randomly shuffled list of requests when sending HTTP requests to the search engine. In order to have deterministic results across multiple test executions, we use the jMeter servers IP address as the seed for all randomization steps mentioned above. The data generation script `request-builder.js` is included in Appendix A7.

⁶<https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-significantterms-aggregation.html>

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

```
1 {
2   "query": {
3     "query_string": {
4       "query": "*",
5       "analyze_wildcard": true
6     }
7   },
8   "aggregations": {
9     "significantQTypes": {
10      "significant_terms": {
11        "field": "subjects.subject",
12        "size": 100
13      }
14    }
15  },
16  "size": 0
17 }
```

Listing 7.8. Elasticsearch query to extract the 100 most significant terms for all documents

Test Execution In order to test the scalability, we want to increase the load on the search engine over time while observing the number of requests the system can handle. Based on the point in time where the number of requests per second no longer increases linearly with the concurrency, we can estimate how many users can use the system. By comparing these measurements with different number of search index replications, we can see the impact of adding new instances.

In our test we use eight jMeter servers which each simulate up to 100 requests in parallel, configured through the environment variable `USERS` for the `jmeter-server` instances. By setting `RAMP_UP_TIME` to 300, each server will add a new thread every three seconds. This time defines the time span in seconds it should from the first simulated user to 100. After this time frame, we keep the tests running for additional 60 seconds. We add the same environment variables to the `jmeter-master` service and start it. This automatically begins the jMeter automatic test. In order to warm up the cache on all Elasticsearch nodes, we run the tests twice, but ignore the results of the first iteration. While the second iteration is executing, we collect resource usage of the search nodes, Query API and load balancer. As soon as the test is completed, the `jmeter-master` instance starts an HTTP server allowing the download and preview of the measured response times.

Once the tests are finished, we retrieve the data and prepare the next iteration with an additional search index replica. We achieve this by first adding the `use.search=true` label to a scalability host using the Rancher user interface to increase the scale of the Elasticsearch service and increase the `number_of_replicas` for the Elasticsearch cluster. We do this by sending Listing 7.7 as the body of a PUT request to the `/datacite/_settings` endpoint of a Elasticsearch node. After the health checks of the services are back to the status healthy, we start the tests again, including the cache warm up. This procedure is repeated to also use the second scalability host virtual machine.

7. Evaluation

Table 7.1. The aggregated results from the test runs. For each number of replicas tested, we can see the number of requests executed and how many requests per second on average. *Average* and the percentiles are the average response times in milliseconds. The percentiles are groups of how the average response time for p percent of the requests, e.g. 90% of the requests received a response in 2.2s using 2 replicas.

Replicas	Requests	Req/s	Average	90% Line	95% Line	99% Line
2	119194	293.78	1323	2209	2753	5076
3	116195	286.30	1359	2268	3078	6339
4	121311	266.27	1298	2162	3481	8038

Results The measurements from jMeter provided results as shown in Table 7.1. We can see in our measurements, that the number of replicas has a negative effect on the average number of requests per second. When using only two replicas there are 293.78 requests per second, with three replicas 286req/s, and with four replicas only 266.27req/s. This is also shown in the response times which increased with the number of instances.

The raw response times for all requests are shown in Figure 7.3, Figure 7.3, and Figure 7.5 for each number of replicas. In these figures we can see that the response times vary based on the number of simultaneously active threads, especially after the 360s when the threads begin to shutdown. Figure 7.6 shows the combined smoothed graphs for the response times of all replica scenarios. To improve readability, the y-axis is limited to a maximum response time of three seconds. The regression was calculated using the loess function of R with a span of 0.1. We can see that the increase in response time is linear until around the range of 230 to 260 seconds of testing and the behavior is similar for any number of replication up until the marked 230s after starting the test. The number of simulated users is linearly increasing every three seconds, adding one user for each of the eight jMeter instance. Because of this we can calculate that there are 616 threads active. This was calculated using the formula $\lceil t/RAMP_UP * USERS \rceil * i$ where t is the point in time 230, the RAMP_UP of 300, 100 USERS per jMeter instance, and i for the eight jmeter-server instances.

Unfortunately, the collection of resource data did not work as expected and therefore we did not analyze the results in time. The CPU measurements are not one value per second, but rather an ever-increasing sum. However this was not resolved by calculating the difference between measurements.

Discussion We can see that the number of replicas in this example has a negative effect on the number of requests per second that the search engine can answer, with four replicas allowing around 27 request per second less. This indicates that the current implementation does not gain performance by adding additional search engine replicas. However, this is a mere indication as the test environment was not ideal, especially the missing resource usage information is necessary for further conclusions. This information could be used to evaluate whether the single Query API instance introduced a bottleneck.

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

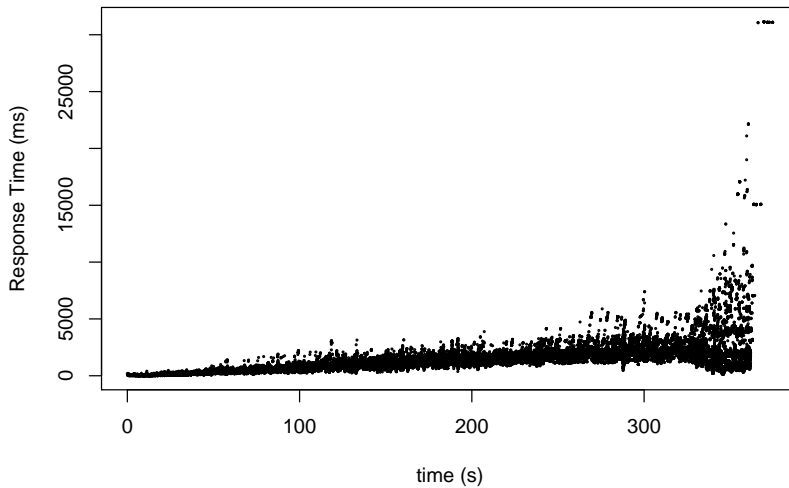


Figure 7.3. All response times over time for 2 replicas. The response time values are shown at the point in time, when the request was initially started.

Additionally, the configuration for Elasticsearch is not a realistic setup, because of the change thread pool settings and low RAM allocation and difference between the base and scalability hosts.

The test plan with jMeter also requires improvements for accurate measurements as the measurements are suspicious. The logs actually contain values beyond test duration for each thread. After 360s each jMeter server destroys its internal threads while the requests are setup to loop forever. Figure 7.7 shows the response times when running two replicas and the red line indicates when threads should begin to shutdown. There are several significant outliers beyond that threshold that might skew the results. This might lead to non-deterministic results when threads are just down that are still waiting for a response. This could potentially be a reason for the extreme outliers in response times. The jMeter plan could be adapted to include a fix duration per thread, allowing a graceful shutdown. Additionally, the Query API instance could introduce a bottleneck, so this service would also require its own scalability analysis.

In the case that the system is actually not scalable, further research should evaluate whether the network configuration between the Elasticsearch nodes introduces a bottleneck and how requests are actually routed within the VPN by Rancher.

7. Evaluation

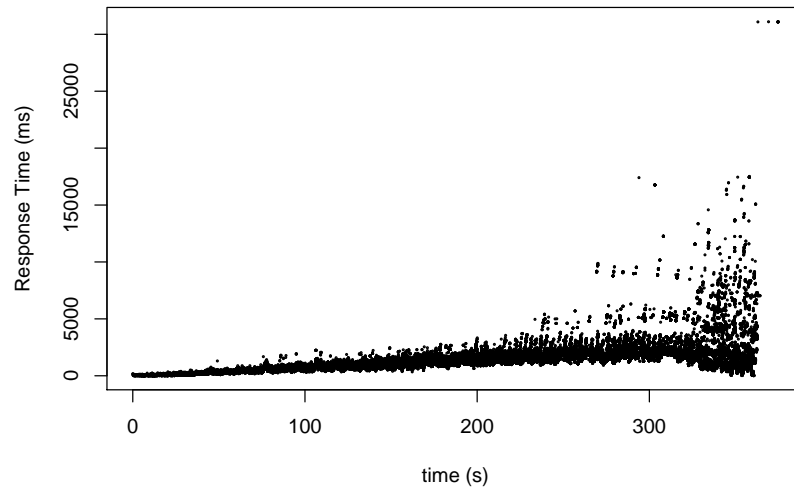


Figure 7.4. All response times over time for 3 replicas. The response time values are shown at the point in time, when the request was initially started.

7.2.7 Threats to Validity

The validation methods used in this section are limited by assumptions and methodology issues during our tests which provide a threat to the *internal validity* of our measurements. Threats to the *external validity* are issues that might hinder the generalization of our findings to a broader field [Runeson and Höst 2008].

Internal Validity The tests we used in our evaluation for the backend system and architecture are mostly verified low latency scenarios, running on a single physical machine. Using a different system or infrastructure may lead to different results in the measurements and show invalid hidden assumptions about how the services can be executed and connected. In the discussion of the results for each metric, we identified potential problems that might have influenced the measured results. Generally, we tried to control as many factors as possible, so as to only measure the influence of a particular factor. However, the scalability evaluation from Section 7.2.6 has several additional problems that we addressed in its discussion of the results.

External Validity The examples used in our evaluations are specific for showing a single aspect of the system which implies several assumptions that communication between

7.2. Goal: Scalable Software Architecture for the Web-based Research Environment to Support External Repositories

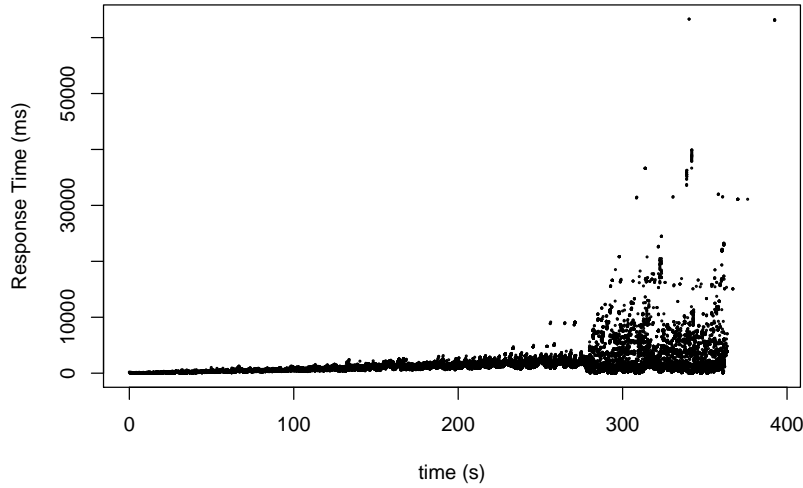


Figure 7.5. All response times over time for 4 replicas. The response time values are shown at the point in time, when the request was initially started.

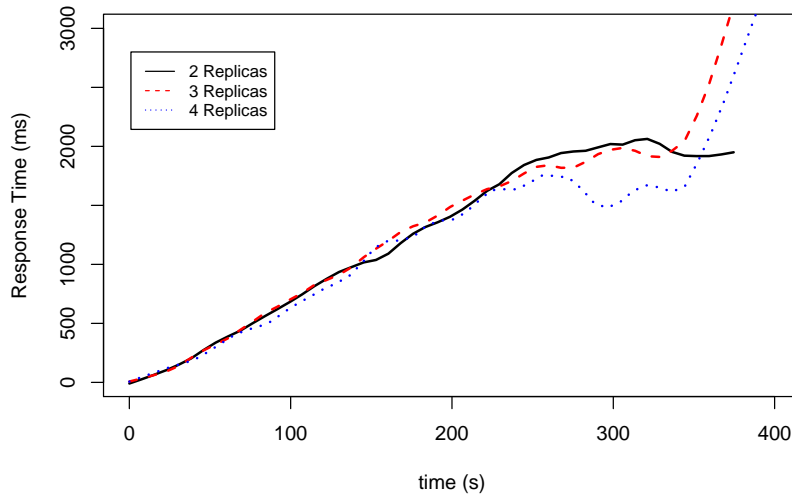


Figure 7.6. Combination of smoothed response times over time for all test cases

7. Evaluation

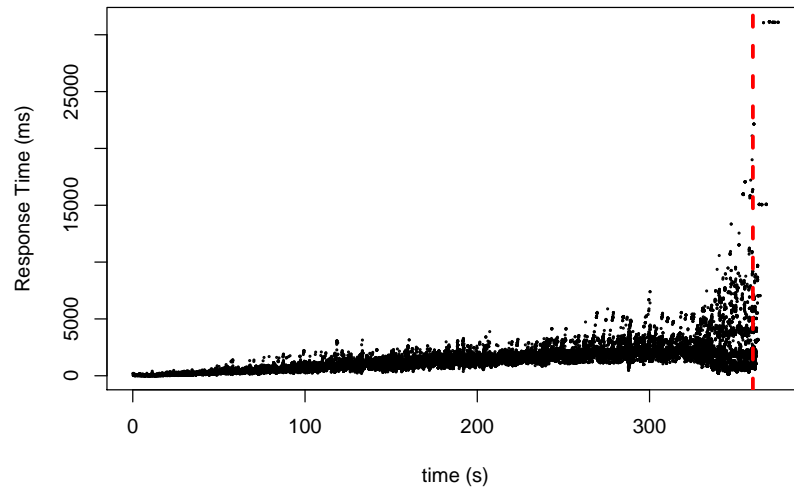


Figure 7.7. All response times over time for 4 replicas

components is possible and reliable. The metrics in this chapter can not be applied to every possible combination of external factors. In real world scenarios, especially when dealing with external HTTP services, services are distributed across multiple machines and introduce unforeseeable situations and edge cases. The findings might not be applicable when the system deals with external services, network traffic of real users, or in a distributed infrastructure.

7.3 G2.1: Web Browser User Interface

In this chapter we evaluate whether we achieved the requirements defined by Goal 2.1.

7.3.1 Question: Does the use of facets narrow down search results?

When working with a multitude of data sources, users potentially find a large list of search results from various sources and scientific fields. Therefore, the frontend must allow users to quickly narrow down the list of results to documents that are relevant for them.

Metric: Number of result documents after applying a facet Searching for documents in the frontend applies the aggregation functionality of Elasticsearch to the result set

7.3. G2.1: Web Browser User Interface

to generate possible facets in the frontend. When clicking on a facet, the search term gets appended with `AND <field>=<value>`. Assuming that the aggregation functionality of Elasticsearch works as expected, we have to determine whether appending the string actually narrows down the results. The current implementation of the facets only supports keyword based facets which only allows for one keyword to be matched against a specific field for each document. Other than the name of the field, the *Publisher* and *Tags* facets are implemented identically. This allows us to only test one field, in order to verify the functionality of both.

To test this, we add two sets of documents to the index of the search engine with two publishers, in our case one for 48821 documents from *SeaAroundUs* and one for 74 documents from *FaoStat*. The *SeaAroundUs* data was ingested as described in Section 7.2.1, *FaoStat* was harvested analogously using the harvester⁷ from the GeRDI project. Each document only has exactly one of those publishers.

Verifying the Functionality As defined in Section 7.1, we use the *Google Chrome* web browser to access the frontend application on `http://localhost:4000` and enter text into the search input. We start by matching all documents in the index by using `*` as the search query. This provides a list of results and displays the total number of results as 43895 which matches the expected number when finding both *SeaAroundUs* and *FaoStat* documents. The sidebar of the page then displays the *Publisher* facet with the total number of documents for both publishers as shown in Figure 7.8.

We expect the *SeaAroundUs* documents to no longer appear in the list of results, when using the *FaoStat* facet. Clicking on it changes the search term to `* AND publisher.raw:"FaoStat"` and loads a new list of results. The displayed number of results is 74 and *SeaAroundUs* is no longer displayed in the facet, indicating only the documents from *FaoStat* are found.

In order to verify that multiple facets can be applied, we first need to find a document set that is small enough to observe a similar effect. When searching for `manure AND subjects.subject:"Swine, market"` we receive three results as shown in Figure 7.9. All values shown in the *Tags* facet panel show that they match three documents, which does not allow narrowing. However, we can emulate such a facet by manually appending `subjects.subject:<term>` to the search query. In order to find a suitable term, we use the detail pages of all three results and find the *Emissions (CH4)* tag that is only contained in the first result (*Manure Management*). Appending the subject to the query results in `manure AND subjects.subject:"Swine, market" AND subjects.subject:"Emissions (CH4)"` as the full query string. After submitting the search request, we only see the *Manure Management* document as the single search result.

Results With this test we have shown that the approach to handle facets by appending the search query does filter the search results to documents that match a specific keyword term. This evaluation only applies to keyword based facets, because others are not implemented in this prototype.

⁷<https://code.gerdi-project.de/projects/HAR/repos/faostat/>

7. Evaluation

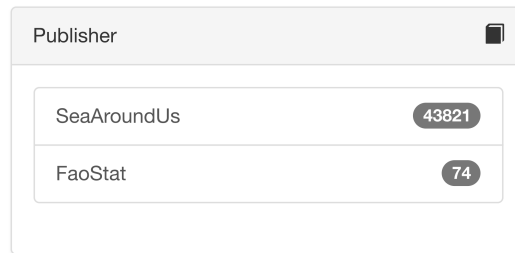


Figure 7.8. Publisher facet when matching all documents.

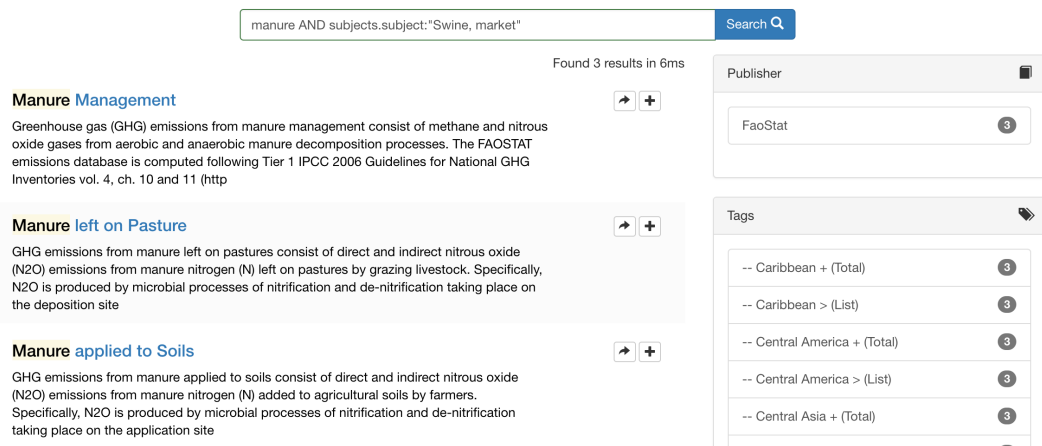


Figure 7.9. Search results page when searching for manure AND subjects.subject:"Swine, market"

7.3.2 Question: Is it possible to preview CSV data in the browser?

In Section 2.2.1 we identified the goal of displaying CSV data as previews within the web browser. We now verify that the functionality works as expected.


Metric: The frontend can display CSV values as a table The evaluation of the CSV preview component uses the same general setup as described in Section 7.2.2. Specifically, it reuses the `filetest` document shown in Listing 7.3 and the HTTP server setup, including the File Provider service and static file server. However, for the purpose of this metric we use different file identifiers. The CSV files `csv1`, `csv2`, and `csv3` from the document are added to the file server. Each file corresponds to at least one test case.


The file `csv1` provides random comma-separated values with a header row. The randomly generated values only contain alphanumeric ASCII characters and its rows delimited by `\n`.

7.3. G2.1: Web Browser User Interface

- 1 Number,NotPlanet," column with space"
- 2 2312,Pluto," Entry with Space"

Listing 7.9. File csv3 with quoted strings and spaces

File Test 

CSV Dataset (Comma) 

s...	first	last	...	word	s...	first	last	...	word	s.
1	Sylvia	Clark	53	mapgafo	1	Cynthia	Schmidt	41	ivrucu	
2	Sean	Mason	49	re	2	Troy	Fox	21	egkop	
3	Eleanor	Lawson	39	vuw	3	Walter	Manning	50	usha	
4	Lloyd	Rivera	33	rome	4	Hester	Reeves	56	ojapi	
5	Tillie	Blair	52	rigbeflow	5	Wesley	May	60	zojotol	
6	Joe	Moreno	61	ogjerpok	6	Todd	Patterson	25	okabedto	
7	Ian	Riley	52	sunruw	7	Trevor	Silva	43	idu	
8	Lawrence	Alexander	29	ohgote	8	Charlie	Reeves	19	opucuaaf	
9	Alex	Aguilar	42	bun	9	Inez	Moreno	60	cideb	
10	Ethel	Duncan	33	rahnira	10	Cecelia	Jensen	37	ambopem	
11	Georgie	Howard	46	orla	11	Annie	Jones	26	todas	

Figure 7.10. Rendered CSV data for files csv1 and csv2, showing 10000 rows of randomly generated data.

We created 10000 test rows via an online CSV generator⁸ with the input seq, first , last ,age,word repeated four times with commas in between. The file csv2 has the same values, but we replace all commas with semicolons. At last csv3 is shown in Listing 7.9 and uses quotation marks to wrap strings that contain spaces.


The tests use the File Provider and the frontend. The setup of both services is defined in Section 7.1. We begin by opening `http://localhost:4000/details/filetest` in Chrome and add the document to the data collection via the plus button in the top right corner of the detail page. After that we navigate to `http://localhost:4000/research` and see the navigation panel in the left sidebar.

We then click on the three files *CSV Dataset (Comma)* (csv1), *CSV Dataset (Semicolon)* (csv2), and *CSV Dataset (Spaces)* (csv3) to view them.

⁸<http://www.convertcsv.com/generate-test-data.htm>

7. Evaluation

File Test



Number	NotPlanet	column with space
2312	Pluto	Entry with Space

Figure 7.11. Rendered CSV data for file csv3

Results The table rendering in Figure 7.10 shows the result when rendering both csv1 and csv2. This shows that it is generally possible to render both comma and semicolon delimited CSV files. Additionally, it also indicates that the automatic delimiter detection works in this two base cases. Figure 7.11 shows the rendering of the file csv3, indicating that quoted columns and headers are respected when parsing the CSV files.

7.3.3 Question: To what extent does the application provide value for the research process?

Evaluating the frontend application does not only require the verification that the functionality works as expected, but also whether the implemented features provide value for researchers. We do this by conducting an expert interview with researchers as a form of qualitative empirical research to gain insight into domain specific knowledge [Meuser and Nagel 2009]. In the context of this thesis, the insights are based around how researchers find and retrieve research data. The metric for answering the question is the feedback and assessment of researchers that need to find and access research data in their daily work. The full transcribed interview (in German) can be found in Appendix A9.

Interviewee Selection Conducting these interviews requires the selection of interview partners. For this we use the case study of the *Environmental, Resource and Ecological Economics (EREE)*⁹ group from the University of Kiel which uses economic data to gain insight about the ecological environment. The group is, amongst other things, aggregating several data sources to publish reports, making the data search, evaluation, and aggregation a crucial part of their work. Some of their data sources, i.e. SeaAroundUs, are integrated as an evaluation data source in the prototype. Additionally, they are community partners of the GeRDI project and one of their first use cases. This implies that they already have knowledge about the project, its goals, and requirements. This is, in addition to the

⁹<https://www.eree.uni-kiel.de>

dependency to external data sources, the reason for interviewing them about the potential value of this prototype implementation, especially the functionality of the frontend. Since both researchers are German, the interviews were conducted in German.

Semi-Structured Expert Interviews According to Kaiser [2014], expert interviews can be semi-structured or structured in their realization. Structured interviews provide a fixed set of questions to be answered in the course of the interview without much deviation. This type of interview is about retrieving a specific set of information from the interviewee. For evaluating the value of the frontend, we use a semi-structured approach. A semi-structured interview imposes less restrictions on the structure of an interview. Like the structured interview, they also follow a set of questions for the interview, but are more open to deviation, e.g. new questions based on the content of the conversation. In our evaluation, we want to have some freedom in the structure to enable discussions and gaining insights that were not expected during the preparation of questions. We further enable an open discussion about the value and challenges of the application by having a group interview with multiple researchers. This allows for discussions between the interviewed parties, in our case two researchers from the same research group.

Designing the Interview Guide Before beginning the interview, a privacy and confidentiality agreement between all parties of the interview is discussed and defined based on the requirements and wishes of the interviewees. The purpose and goal of the interview are explained in an introduction phase. This phase includes a presentation of the final state of the frontend application, its features, and their intention, providing a first impression to the researchers. During the introduction, they are encouraged to ask comprehension questions about the software and its functionality. A focus within the introduction is to define the scope of the prototype and how its functionality might provide a basis for further development.

The introduction is followed by broad warmup questions about how the person interacts with research data in their daily work and what data sources are used. The intent of these questions is to learn about the researchers context and to gain first insights on what functionalities of the software might be relevant for the discussion. We can use this information to identify potential data sources for the application. Additionally, the researchers are asked what criteria they use when evaluating which data might be relevant for their research. As the last part of the warmup period, the interviewees should state their first impression, shifting the focus to the main interview segment.

The main interview is about the functionality of the frontend and its relevancy in the work of the scientists. This includes questions how and which functionality could be integrated into the data research work of the scientists. We also want to get assessments of the specific pages of the frontend. This includes, to what extent the faceting feature might help with finding the relevant data sets, to what extent the data collection provides value. The questions are also aimed at exploring what is necessary to transform the prototype

7. Evaluation

into a valuable tool. At last, the interview is concluded with a grading on a scale from zero, if the application does not provide any usable value at all, to ten when the application will be the new main research tool.

Introduction Summary Both interviewees began by introducing their work and how they work with research data. This was separated into the process for a previous project and their current work. Previously, they were collecting data from SeaAroundUs and the *Food and Agriculture Organization of the United Nations (FAO)*. These data sets were gathered through their websites and, in the case of the FAO Statistics Division, with a specific software, which allows filtering within the data. Data from these services was combined and made compatible, which is challenging as they use different classifications. Data from current projects was gathered from various sources, manually collected, or provided by third parties.

Search Page The interviewees considered the search page as the basic requirement for gaining access to multiple data sources. In its prototype state, the search was considered clear and concise, but might provide more of a challenge when connected to a larger set of data sources. The facets (filters) are mentioned as crucial for handling a diverse set of data sources and are a key requirement for exploration. Additional facets should include the scientific field the data belongs to and also time-based facets, e.g. which years the data is related to. A concern of an interviewee was that the number of facet values might be so high that it would require additional exploration functionality.

Detail Page The detail page was not deemed to be that relevant for them, as they would directly access the source website in most cases. However, it was noted that the citation information is important and that data is shown in the sidebar of the detail page. Generally, metadata should be updated as their data sources are aggregations over long periods of time and data is added to existing documents. Information about the last update date of the data should be indicated. One of the researchers mentioned that the metadata could contain information about the type and used standards of the data, in order to decide whether two data sets are compatible.

Research Page In the context of the research page, one interviewee noted that the data collection would be heavily used, requiring an additional search filter for finding the stored documents in the large list. The discussion with the researcher also included whether multiple files should be shown on the research page with a dashboard, but it was said that this is not important, as it could be too much data in their use case. According to the researchers, the persistent collection of data would provide a central place to save their current research data, avoiding a large number of web browser bookmarks and tabs. Overall, both researchers noted that the data collection is their favorite tool of the frontend and considered it helpful.

7.3. G2.1: Web Browser User Interface

Overall Assessment The overall feedback from the interviewees was very positive. When asked about an assessment from zero to ten how the prototype is usable in their research work, both answered with an eight. However, this is under the assumption that more data sources are integrated into the system and it is also just a first impression, as they would need to spend more time with the application to form a definitive opinion. The approach of the system to combine a large number of data sources was the main positive aspect of the system, requiring a solid search and filter functionality. However, both researchers agreed that the data collection was the best new tool provided by the software. Overall, the whole provided functionality was considered useful.

Conclusion The results from the interviews indicate that the frontend application might provide a valuable tool in the daily work of researchers. Suggestions made by the researchers include extensions of metadata and adding new facets based on the data, i.e. adding the scientific field to the data and additional time data. The current version of the frontend is already compatible with these extensions, as facets can be added through a configuration. Only time based facets would require additional implementation. Such functionality is largely based on what the metadata provides, shifting the focus to the metadata schema used. For the use cases of the interviewees, the frontend application prototype can provide a valuable tool. However, more requirement analyses and interviews with a larger and more diverse group of scientists are required to form a well-informed conclusion.

7.3.4 Threats to Validity

Internal Validity The functionality of the frontend was tested in a tightly controlled local setup, however results may be different if the system is running on another machine or within another web browser. The user interface was used by the implementor who might unwittingly use in-depth knowledge and previous experiences during the execution of the tests. The effectiveness of search faceting was measured through the number of results for a specific value, the underlying Elasticsearch aggregations however, can be approximations in certain scenarios. However, these are unlikely when running in local setup without additional write operations or any distribution. The interview from Section 7.3.3 are dependent on personal factors and the feedback from researchers could be influenced by external factors. Additionally, personal interviews each interviewee individually could provide different insights and feedback.

External Validity In real world scenarios, high latencies, unexpected server errors, and many more factors can influence how the application can react to users. Data like CSV can also vary significantly, especially with different delimiter standards, character encodings, or even corrupt files. When connected to external resources, with large search index sizes or distribution, the system could behave differently due to unforeseeable situations. Our

7. Evaluation

evaluation can only provide an indicator that the functionality works as expected in a narrow field of tested use cases.

Similarly, the results from the interview can only provide a broad opinion rather than fact, as the number of interviewed researchers is too small for conclusive results.

Related Work

This thesis implements a first prototype and proof-of-concept functionality inspired by the GeRDI project [Grunzke et al. 2017]. It shares the goal of supporting researchers in their daily work with research data, focusing on a search functionality. However, we provide additional functionality like the data preview that are not described in the context GeRDI project. In the following chapter we describe similar projects and which aspects are different from our prototype.

Several other projects are working towards improving access to research data and fostering its reuse. For research specific domains there are projects like *PANGAEA*¹ which allows researchers in the earth and environmental sciences to publish and find data sets [Schindler et al. 2012]. The service focuses on the storage and general access to the data sets. Similar to this thesis, it provides a search engine with extensive faceting functionality which also allows for filtering based on domain-specific categories, like what devices were used to collect the data. While providing extensive capabilities for its domain it does not allow research data to be searchable across disciplines.

The *Zenodo*² project does provide publishing data access across a wide range of scientific fields. It not only focuses on raw data but also allows upload of academic papers, presentations, and also software. Similarly to this thesis, it allows users to preview the data before downloading. However, it does not provide functionality for researchers to collect relevant data sets and share these collection like the frontend of this thesis. The data that Zenodo indexes is not integrated from external repositories and therefor cannot integrate with existing infrastructures.

The *DataCite* project not only provides a metadata standard, but also implements a search engine³ spanning multiple external repositories [Brase 2009]. However, its functionality is focused on finding metadata in order to cite it. It does not provide direct download access to the data but instead displays the metadata and links to the source repository via the DOI. The faceting functionality is limited to only resource types, publication year, and the data repository is located at. This thesis uses a DataCite compatible metadata schema internally and provides similar functionality. However, we display more of the metadata in the frontend application and provide direct access to the files, including web browser-based file previews.

¹<https://pangaea.de>

²<https://zenodo.org>

³<https://search.datacite.org>

8. Related Work

*Research Data Australia*⁴ also provides a search engine based on metadata from various external research data repositories [Burton et al. 2012]. External services are integrated selection of multiple metadata retrieval standards, i.e. OAI-PMH, while the original data repository is responsible for storing the data [*Providing Metadata Records*]. This is similar to our approach of using schema specific microservices that harvest metadata from the source repositories. Similar to the data collection implemented in this thesis, the web application also allows users to save interesting documents through its *MyRDA* functionality. However, these collections cannot be shared with other people which we allow. Our thesis also provides the ability to preview the data sets in the browser without downloading it beforehand. This is not implemented in Research Data Australia.

⁴<https://researchdata.ands.org.au>

Conclusion & Future Work

In this thesis, we presented an approach for a distributed software architecture to provide a web application that helps scientists find and explore research data. To achieve this, we first identified the requirements and features to pursue and subsequently designed the architecture accordingly. Based on this, we designed the architecture of the search engine and how documents can be added to its index. The proposed microservice architecture for harvesting metadata from external research repositories is designed to support several possible metadata standards. We described how to handle several possible scenarios when integrating a new repository as a harvesting source and how the architecture can aid in the process. For this we also defined conventions and restrictions on how the harvesters should be integrated. We also designed and implemented two services that allow researchers to download the research data from external sources through a uniform interface and also support bundling of files from multiple sources into a single download. These services also provide a guideline regarding the integration of additional functionality into the architecture.

The implemented web based user interface is designed to provide the bridge for the functionality of all services and apply the separation of concerns from the microservice architectural pattern to the interface. Through this and additional conventions, we implemented a component based application with a strong focus on extensibility and maintainability. Apart from search functionality, the interface provides a data collection in order to let scientists store the relevant data sets for their work. We also implemented the ability to let users share their collection with colleagues. Additionally, we implemented web browser based previews of research data. This data is stored at external repositories and are accessible through our uniform data access service.

Through a qualitative interview with two external researchers, we evaluated that the application would likely be integrated in the daily work of these scientists. Especially the data collection was highlighted as a positive addition to the functionality. In addition to the interview, we also evaluated several functionalities of both our services and the user interface. We integrated an existing research data repository with our search engine, indicating its applicability to external repositories. In addition to the functional verification, we also measured performance indicators of the services. We verified that the data access services provide minimal overhead in terms of download throughput and server-side memory usage. Additionally, we tested the scalability of our architecture. Our measurements indicate that the search engine cannot handle more concurrent search requests if more

9. Conclusion & Future Work

servers are added to the infrastructure. On the contrary, the search engine did handle less requests with an increased number of instances. However, we pointed out several flaws with our testing setup and might have influenced the results.

In conclusion, we suggest the usage of this prototype as the foundation for implementing additional functionality and for further research. However, this requires some work to resolve the scalability issues and integrate several external repositories, before the system can no longer be considered a mere prototype.

Future Work In this thesis, we do not implement all features necessary for usage in real world scenarios. The backend requires the support for metadata standards in order to integrate existing external repositories. The internally used metadata schema is currently based on DataCite but might not be a suitable candidate. Especially the usage as a mapping configuration for Elasticsearch is not ideal because not all metadata information is necessary for the search functionality. This produces an unnecessary resource overhead.

While the architecture is inspired by the microservice architectural pattern, it is not fully following this approach. The search index is used as a central data storage and used by the File Provider service that do not necessarily require the full search functionality. However, the service is adaptable to use its own data storage because its implementation can provide multiple retrieval strategies. A local document store could subscribe to documents on the message broker to update. The evaluation in Section 7.2.6 indicates that the scalability of the search engine is compromised. In order to use the search engine in a public scenario this must be resolved. Scaling should also be done automatically based on load monitoring.

While all services provide health check endpoints, they do not accurately represent the state of the system. The implemented health checks only check whether the service is reachable over HTTP but not if it can provide its functionality. Especially, the health check for Elasticsearch is not ideal as instances are considered unhealthy when the replicas are synchronizing data which does not necessarily mean that the node cannot answer requests.

The interviewed researchers in Section 7.3.3 identified that the faceting and filtering of search results in the frontend is an essential part when dealing with a lot of data sources. Additional research should be done on what type of filtering is necessary for researchers and how they can be implemented. The frontend application provides the technical infrastructure to support additional facets. Generally, the frontend can implement more features in regards to data exploration. However, this requires an analysis of available metadata values that could be combined to provide related documents. The search functionality is limited to the text query syntax which Elasticsearch natively provides. However, this does not provide fine-grained control over the search, e.g. treating documents with a matching title with a higher score. A proprietary query syntax could be defined that allows the frontend to identify which information is required to resolve the search results and building the query accordingly. This can also be used to visualize what facets the user already selected.

Bibliography

- [Amazon Web Services 2016]. Architecting For The Cloud: Best Practices. *Amazon Web Services Whitepapers* (2016). (Cited on pages 41 and 86)
- [Barth 2011] A. Barth. *The Web Origin Concept*. RFC 6454. RFC Editor, 2011. URL: <http://www.rfc-editor.org/rfc/rfc6454.txt> (visited on 06/03/2017). (Cited on page 7)
- [Barth et al. 2008] A. Barth, C. Jackson, C. Reis, T. Team, et al. The Security Architecture Of The Chromium Browser. *Technical report* (2008). (Cited on page 7)
- [Basili and Rombach 1988] V. R. Basili and H. D. Rombach. The TAME project: Towards Improvement-oriented Software Environments. *IEEE Transactions on software engineering* 14.6 (1988), pages 758–773. (Cited on page 75)
- [Berners-Lee et al. 2005] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. RFC Editor, 2005. URL: <http://www.rfc-editor.org/rfc/rfc3986.txt> (visited on 06/04/2017). (Cited on page 5)
- [Bondi 2000] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance. In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, 2000. (Cited on page 8)
- [Brase 2009] J. Brase. DataCite - A Global Registration Agency for Research Data. In: *Cooperation and Promotion of Information Resources in Science and Technology, 2009. COINFO'09. Fourth International Conference on*. IEEE, 2009, pages 257–261. (Cited on pages 20, 41, and 105)
- [Bray 2014] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. RFC Editor, 2014, pages 1–16. URL: <http://www.rfc-editor.org/rfc/rfc7159.txt> (visited on 08/01/2017). (Cited on page 24)
- [Brisco 1995] T. Brisco. *DNS Support for Load Balancing*. RFC 1794. RFC Editor, 1995. URL: <http://www.rfc-editor.org/rfc/rfc1794.txt> (visited on 08/01/2017). (Cited on page 8)
- [Burton et al. 2012] A. Burton, D. Groenewegen, C. Love, A. Treloar, and R. Wilkinson. Making Research Data Available in Australia. *IEEE Intelligent Systems* 27.3 (2012), pages 40–43. (Cited on page 106)
- [*Computed Properties and Watchers*] V. Contributors. *Computed Properties and Watchers*. URL: <https://vuejs.org/v2/guide/computed.html> (visited on 08/02/2017). (Cited on page 15)
- [*Introduction to Vuex*] V. Contributors. *Introduction to Vuex*. URL: <https://vuex.vuejs.org/en/intro.html> (visited on 07/09/2017). (Cited on page 17)
- [DB-Engines.com 2017] DB-Engines.com. *DB-Engines Ranking of Search Engines*. 2017. URL: <https://db-engines.com/en/ranking/search+engine> (visited on 06/19/2017). (Cited on page 23)

Bibliography

- [*What is Docker*] Docker. *What is Docker*. URL: <https://www.docker.com/what-docker> (visited on 06/05/2017). (Cited on page 12)
- [Docker 2016] Docker. *Docker for Virtualization Admins*. 2016. URL: <https://goto.docker.com/rs/929-FJL-178/images/Docker-for-Virtualization-Admin-eBook.pdf> (visited on 06/05/2017). (Cited on page 12)
- [Ecma International 2013] Ecma International. *The JSON Data Interchange Format*. Standard 404. 2013. URL: <https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf> (visited on 06/12/2017). (Cited on page 6)
- [Elastic Reference] Elastic. *Elasticsearch Reference - Thread Pool*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/5.4/modules-threadpool.html> (visited on 07/29/2017). (Cited on page 89)
- [Elastic 2017a] Elastic. *Nested Datatype*. 2017. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/5.4/nested.html> (visited on 06/10/2017). (Cited on page 10)
- [Elastic 2017b] Elastic. *Zen Discover*. 2017. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/5.4/modules-discovery-zen.html> (visited on 06/12/2017). (Cited on page 9)
- [European Commission 2012] European Commission. *Commission Recommendation On Access to And Preservation of Scientific Information*. 2012. URL: http://ec.europa.eu/research/science-society/document_library/pdf_06/recommendation-access-and-preservation-scientific-information_en.pdf (visited on 02/08/2017). (Cited on page 1)
- [Fehling et al. 2014] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns*. Springer Verlag Wien, 2014. (Cited on pages 9, 21, and 29)
- [Felter et al. 2015] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE. 2015, pages 171–172. (Cited on pages 12, 13)
- [Fielding et al. 1999] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, 1999. URL: <http://www.rfc-editor.org/rfc/rfc2616.txt> (visited on 06/03/2017). (Cited on pages 6 and 24)
- [Kafka Introduction] A. S. Foundation. *Apache Kafka: A Distributed Streaming Platform - Introduction*. URL: <https://kafka.apache.org/intro> (visited on 07/26/2017). (Cited on page 86)
- [GeRDI Project 2017] GeRDI Project. *GeRDI Search Architecture Options*. 2017. URL: <https://wiki.gerdi-project.de/display/GeRDI/Search+Architecture+Options> (visited on 03/01/2017). (Cited on pages 21, 22)
- [Grunzke et al. 2017] R. Grunzke, T. Adolph, C. Biardzki, A. Bode, T. Borst, H.-J. Bungartz, A. Busch, A. Frank, C. Grimm, W. Hasselbring, A. Kazakova, A. Latif, F. Limani, M. Neumann, N. T. de Sousa, J. Tendel, I. Thomsen, K. Tochtermann, R. Müller-Pfefferkorn, and W. E. Nagel. Challenges in creating a sustainable generic research data infrastructure. *Softwaretechnik-Trends* 37.2 (2017). (Cited on pages 1 and 105)

- [Haerder and Reuter 1983] T. Haerder and A. Reuter. Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys (CSUR)* 15.4 (1983), pages 287–317. (Cited on page 9)
- [Heorhiadi et al. 2016] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic Resilience Testing of Microservices. In: *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*. IEEE. 2016, pages 57–66. (Cited on page 41)
- [Kaiser 2014] R. Kaiser. *Qualitative Experteninterviews: Konzeptionelle Grundlagen und praktische Durchführung*. Elemente der Politik. Springer Fachmedien Wiesbaden, 2014. (Cited on page 101)
- [Lewis and Fowler 2014] J. Lewis and M. Fowler. *Microservices - A Definition of this New Architectural Term*. 2014. URL: <https://martinfowler.com/articles/microservices.html> (visited on 03/01/2017). (Cited on pages 8 and 35)
- [Meuser and Nagel 2009] M. Meuser and U. Nagel. The Expert Interview and Changes in Knowledge Production. *Interviewing Experts* (2009), pages 17–42. (Cited on page 100)
- [Newman 2015] S. Newman. *Building Microservices*. "O'Reilly Media", 2015. (Cited on page 8)
- [Pampel et al. 2013] H. Pampel, P. Vierkant, F. Scholze, R. Bertelmann, M. Kindling, J. Klump, H.-J. Goebelbecker, J. Gundlach, P. Schirmbacher, and U. Dierolf. Making Research Data Repositories Visible: The re3data.org Registry. *PLoS one* (2013). (Cited on page 5)
- [Paskin 2010] N. Paskin. Digital object identifier (DOI) system. *Encyclopedia of Library and Information Sciences* 3 (2010), pages 1586–1592. (Cited on page 35)
- [RancherLabs 2017] RancherLabs. *Networking in Rancher*. Version 1.6. 2017. URL: <http://docs.rancher.com/rancher/v1.6/en/rancher-services/networking/> (visited on 06/07/2017). (Cited on pages 14, 15)
- [Rescorla 2000] E. Rescorla. *HTTP Over TLS*. RFC 2818. RFC Editor, May 2000. URL: <http://www.rfc-editor.org/rfc/rfc2818.txt> (visited on 06/03/2017). (Cited on page 6)
- [Roush 2001] E. T. Roush. Cluster Rolling Upgrade Using Multiple Version Support. In: *Proceedings of the 3rd IEEE International Conference on Cluster Computing*. IEEE Computer Society. 2001, page 63. (Cited on page 14)
- [Runeson and Höst 2008] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14.2 (2008), page 131. URL: <https://doi.org/10.1007/s10664-008-9102-8> (visited on 07/28/2017). (Cited on page 94)
- [Sadalage and Fowler 2012] P. J. Sadalage and M. Fowler. *NoSQL Distilled: a Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012. (Cited on page 9)

Bibliography

- [Schemers 1995] R. Schemers. Ibmname: A Load Balancing Name Server in Perl. In: *LISA*. Ninth System Administration Conference. USENIX Association, 1995. (Cited on page 8)
- [Schindler et al. 2012] U. Schindler, M. Diepenbroek, and H. Grobe. PANGAEA - Research Data enters Scholarly Communication. In: *EGU General Assembly Conference Abstracts*. Edited by A. Abbasi and N. Giesen. Volume 14. EGU General Assembly Conference Abstracts. 2012, page 13378. (Cited on page 105)
- [*Providing Metadata Records*] A. N. D. Service. *Providing Metadata Records*. URL: <http://www.ands.org.au/online-services/research-data-australia/collections-registry/providing-collection-descriptions> (visited on 08/01/2017). (Cited on page 106)
- [van Kesteren 2014] A. van Kesteren. *Cross-Origin Resource Sharing*. W3C Recommendation. <http://www.w3.org/TR/2014/REC-cors-20140116/>. W3C, 2014. (Visited on 06/03/2017). (Cited on pages 7 and 34)
- [Vaquero et al. 2011] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically Scaling Applications in the Cloud. *SIGCOMM Comput. Commun. Rev.* (2011). (Cited on page 8)
- [Zalewski 2008] M. Zalewski. *Browser Security Handbook, part 2*. 2008. URL: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki#Same-origin_policy (visited on 06/03/2017). (Cited on page 7)
- [Zimmermann 1980] H. Zimmermann. OSI Reference Model – The ISO Model Of Architecture For Open Systems Interconnection. *IEEE Transactions on communications* 28.4 (1980), pages 425–432. (Cited on page 6)

Appendices

A1 Query API

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-query-api>

Commit Hash:

5492da38695656344bdfd1414fee70d6e6abd0b0

A2 Ingest API

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-ingest-proxy>

Commit Hash:

51fad965a4a747e4370d2cf72f361182b3fa0d41

A3 File Provider

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-file-provider>

Commit Hash:

e1a91d11e19ecf74be52cb82ae9a60b20c575c1a

A4 File Merger

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-zip-merger>

Commit Hash:

ab5dbbf31c82061861b4b490c125a1b38228d557

A5 Frontend Application

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-frontend-prototype>

. Appendices

Commit Hash:

9e066dfd2d3d36c257d162b290f245de19a39bf4

A6 Elasticsearch Mapping

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-elasticsearch-mapping>

Commit Hash:

`fa8da4a942aa659676c379fbbf21c3f785b2dc12`

A7 Evaluation Scripts

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-evaluation>

Commit Hash:

`6ed7f279926171f99affab1ae40f46acc4199e38`

A8 DataCite to Elasticsearch Transformer

Git Repository URL:

<https://build.se.informatik.uni-kiel.de/mne/thesis-datacite-transformer>

Commit Hash:

`d8061fb60b53bfabf6b4891ba6df38097676c002`

A9 Interview Transcript

Teilnehmer: Mathis Neumann (I), Person A & Person B.

I: Ich würde euch kurz darum bitten, kurz zu beschreiben, wie euer Alltag aussieht mit wissenschaftlichen Daten, was ihr so macht im Allgemeinen, damit ich dann darauf referenzieren kann, wenn ich meine Auswertung darüber schreibe. Also fangen wir am besten von links nach rechts an, aus meiner Perspektive.

A: Okay, ich fang mal an mit der aktuellen Situation: Ich arbeite momentan recht viel mit Daten, zum einen mit Daten, die wir selber erheben durch Interviews *[bei anderem Projekt]*, also in der Fischerei-Community, und ich krieg halt die Rohdaten aus den Interviews und verarbeite die mit Stata, bearbeite die, editiere die, sodass andere Leute den fertigen Panel-Datensatz haben am Ende. Ein zweites Datenprojekt basiert auf Daten von *[externer Quelle]*, das ist auch ein Panel-Datensatz zum Kaufverhalten von Menschen in Deutschland. Da haben wir einen Datensatz bekommen von *[externer Quelle]* und auch der musste erstmal bereinigt werden und sowohl die Daten aus dem *[anderen]* Projekt als auch die von *[externer Quelle]* werden dann nach der Bereinigung und Aufarbeitung in Stata weitergenutzt, um diverse Sachen zu modellieren und beschreibende Statistiken und so was zu berechnen. Dann historisch, also in Bezug insbesondere auf das WWF-Projekt, da hab ich viel mit Daten von SeaAroundUs gearbeitet, auch da wieder Daten per Hand runtergeladen und

. Appendices

dann zusammengefasst, sodass wir letztendlich die aggregierten Daten hatten, die wir brauchten für die Analyse. Ja, ich hab nur SeaAroundUs gemacht, glaub ich...ja. Das waren Fangdaten überwiegend.

I: Das heißt, die Arbeit mit den Daten war hauptsächlich nachher das Aggregieren, das Filtern und dann darauf basierend irgendwelche Statistiken praktisch zu errechnen, oder wie lässt sich das zusammenfassen?

A: Schon, also das Sammeln war an sich nicht so schwierig, weil SeaAroundUs das schon alles zusammen hatte, aber tatsächlich zeitlich auf jeden Fall der größte Aufwand war, die einzelnen heruntergeladenen Files zusammenzupacken, zu aggregieren und dann weiter für Modellierungen und Schätzungen und so zu nutzen, genau.

I: Gut, dann gehen wir weiter.

[an B gewandt] Fangen wir doch mit der Historie an, im gleichen Kontext.

B: Okay. Also, jetzt muss ich wieder mein Gehirn zusammenkriegen, weil das war so viel. Also ich hab mit FAOStat-Daten gearbeitet, das heißt, die hab ich über die Internetseite sozusagen heruntergeladen und da auch schon gefiltert und dann teilweise auch nochmal...also teilweise hab ich die...die hab ich hauptsächlich mit Excel bearbeitet tatsächlich und die Ergebnisse, die ich da rausgefiltert habe, wurden später sozusagen in Stata eingepflegt. Teilweise um die mit *[Name von A]*s Daten zu mergen und teilweise sind sie dann an *dritte Person* gegangen, der dann...ich weiß gar nicht, mit was der arbeitet, mit welchem Programm.

A: Matlab?

B: Genau, mit Matlab. Also, das ist so der eine Weg. Dann hab ich gearbeitet mit Daten von FAO, von der Fisheries Statistics Division. Da hab ich tatsächlich sozusagen eine Application heruntergeladen, die das Filtern und das Betrachten der Daten schon einfacher macht, und hab darüber auch gefiltert und aussortiert, hab die Daten auch wieder in Excel bearbeitet und die sind genauso dann auch wieder teilweise wie die Ergebnisse zu *[Name von A]* über Stata dann zur Auswertung gegangen und teilweise zu *[Name von dritter Person]*.

I: Diese Applikation, die von FAOStat da, von...

B: Von den Fisheries?

I: Genau. Was hat die so für Funktionalitäten ganz grob geliefert, also war das auch eine reine Suche und dann Filtern der Daten oder hat die auch schon Analysen durchgeführt oder ähnliches?

B: Also es kann sein, dass man damit auch Analysen durchführen kann, das hab ich nicht gemacht. Erstmal meine ich, dient die einfach dazu, die Daten sozusagen anschaulicher zu machen. Die werden genau, also man lädt sich sozusagen Datensätze runter dazu, also man lädt sich die Applikation runter und dann die Datensätze, und dann kann man innerhalb dieser Datensätze halt sich verschiedene Sachen angucken und dann vor allem filtern. Also es gibt dann, gerade was die Fischarten angeht, unterschiedliche Klassifizierungen, nach denen kann man das dann aufsplitten, oder nach Handelsklassifizierungen, also das ist sehr kleinteilig teilweise. Und dann halt auch die Länder und

die Zeitreihe und das kann man alles...genau. Also ich hab das hauptsächlich zum Filtern sozusagen verwendet und zum Aggregieren eben auch ganz gut.

I: Was waren da so die Hauptfilter, die du benutzt hast? Also waren das, du hattest erwähnt "Länder" und "Zeitraum" zum Beispiel und auch die Kategorisierung, was ist da so für dich das Ausschlaggebende?

B: Also für uns war tatsächlich auch am ausschlaggebendsten, glaub ich, die Kategorisierung, weil das war einfach wichtig, dass man das gut hinbekommt. Es wäre wichtig gewesen, dass man es noch hätte besser machen können, das geht nicht so gut damit, aber das würde ich so am wichtigsten bezeichnen, weil wir mussten es ja zusammenbringen mit den SeaAroundUs-Daten und das war relativ schwierig, sozusagen da die gleiche Ebene, die gleiche Produktebene sozusagen zu finden.

I: Also praktisch verschiedene Datensätze zu vergleichen, die die gleichen Klassifizierungen benutzen.

B: Genau, richtig, genau. Bei SeaAroundUs war das nochmal wie? Ihr hattet es nach Länge, ne?

A: Genau. Und Art.

B: Länge und Art. Und das geht halt irgendwie nicht bei der FAO Fisheries-Datensammlung. Da konntest du dann schon nach Art, aber die Arten sind dann teilweise dadurch, dass das Handelsdaten sind, halt sehr unterteilt. Also da hast du dann Sardinen in Öl, in Salz, in...

A: In trocken.

B: Genau, getrocknet. In Mehl, in Öl. Also halt so völlig...genau. Und das war dann natürlich... Und die haben dann zwar unterschiedliche Klassifizierungen, wo man auswählen kann, aber nicht die, die...

I: Nicht die man braucht.

A: Also es war halt differenzierter, feiner aufgegliedert also die SAU-Daten und das war schwierig, die dann so zu aggregieren, dass es wieder mit der SAU kompatibel ist, weil es halt...man kannte halt nicht die Gesamtzahl der Kategorien, das heißt man musste das halt per Hand durchgehen, was ist denn jetzt hier genau 'Sardine'?, und dann war das jetzt halt aufwendig.

B: Das hab ich auch angefangen, aber dann abgebrochen. Genau...also die Kategorie fand ich wichtig, klar Zeitreihe eingrenzen, aber ich mein, das kann man auch, wenn man die Daten runtergeladen hat, also da schmeißt man die einfach mal raus, das ist jetzt nicht so wild.

I: Sind das jetzt also die...sind die Daten, die dort angeboten werden, immer praktisch die gesamten oder sind das so, ich sag mal, ein Datensatz pro Jahr oder ähnliches?

B: Nee, das ist immer die gesamte Zeitreihe, die angeboten wird, die verfügbar ist. Und ich überleg gerade, was ich noch gemacht hab. Was auch noch ein relativ wichtiges Filtertool war, war das auch FAO-Schätzungen mit drin und die kann man dann auch noch sozusagen rauskicken, drinlassen, denen bestimmte Werte zuweisen, das fand ich auch noch recht wichtig, dass man da sozusagen, dass man das gut bearbeiten kann, und das geht da recht gut mit dem Tool.

. Appendices

A: Vielleicht nochmal kurz ergänzend zu den SAU-Daten. Da war der Download auch der komplette Datensatz, also wenn ich mir...auf der Homepage konnte man so differenzieren, ich möchte nur ein Land mir angucken oder nur die Fangmenge, weil beim Download war die komplette Zeitreihe, mit allen mit Länge, mit Fischarten, mit Handelsnamen und...

I: Über alle Länder oder dann auch nur über das eine Land?

A: Über...na über alle Länder, die dann in dem Large Marine Ecosystem, also dem Ökosystem gefischt haben.

I: Ach so, okay. Gut, dann gehen wir noch mal auf heute ein. Wie sieht das denn heute aus mit der aktuellen Daten-Arbeit sozusagen?

B: Also die Daten, mit denen ich momentan arbeite, die hab ich alle schon, das heißt, ich bin gerade nicht am runterladen. Momentan lasse ich hauptsächlich Schätzungen laufen. Das heißt, also ich weiß...ich mach das ja auch mit Stata, ich weiß jetzt nicht, ob da jetzt noch irgendwas für dich dran interessant ist, wenn ich nichts mehr runterlade. Wenn ich runterladen würde, müsste ich tatsächlich, hab ich jetzt lange nicht mehr geguckt. Also die eine Internetseite, wo ich die Daten herhabe, die Konfliktdaten, die haben das komplett neu gemacht, das Interface zum Runterladen, das heißt, da müsste ich mich reinfuchsen. Und die Comtrade-Daten habe ich tatsächlich von [erwähnt dritte Person] bekommen.

I: Also per Hand sozusagen.

B: Genau, per Hand, per USB-Stick. Das heißt, die hab ich nie runtergeladen. Ich weiß aber auch nur, also ich hab es mal versucht, ich weiß, dass es relativ schwierig ist. Deswegen, da kann ich jetzt gerade gar nichts zu sagen. Genau.

I: Gut. Wunderbar, dann hab ich auch schon einen Überblick darüber, welche Datenquellen ihr so verwendet. Deshalb auch [unverständlich]. Was sind denn so die...ach so. Das Wichtigste ist natürlich zuerst, da wir jetzt den Part abgearbeitet haben, jetzt würde ich gerne so die ersten Eindrücke von dem Prototyp gerne aufnehmen. Ja, fangen wir gleich mit dir an, [Name von A]. Was ist dir so aufgefallen grundsätzlich, wie ist so der erste Eindruck?

A: Der erste Eindruck ist sehr gut. Ich finde es sehr übersichtlich und auch intuitiv bedienbar, glaube ich, weil es halt dicht an Google angelehnt ist, zumindest die Suchfunktion. Ich finde auch, dass sich alles weitere dann gut erschließt, fand auch alle vorgestellten Tools sinnvoll tatsächlich. Und ich müsste jetzt überlegen, ob mir noch irgendwas fehlt.

I: Da kommen wir sonst gleich noch drauf.

A: Okay. Noch eine kurze Rückfrage. Und zwar zu dem, es gab ja die, ich glaube, wenn man über Research reingeht, konnte man ja in die Daten reingucken, ne?

I: Genau.

A: Genau. Und war da schon die Option, dass man die da schon irgendwie aggregieren kann? Nee, ne?

I: Nein, das ist aktuell nur eine reine Vorschau, um dann zu entscheiden, ist das relevant? Und wenn es einem gefällt, dann eben downloaden.

A: Ja, das finde ich auch gut, dass man quasi auf allen Seiten runterladen kann, dass

man nicht irgendwie wieder zurückklicken muss. Also so finde ich das sehr gut.

I: Wunderbar, vielen Dank.

[an B gewandt] Jetzt zu dir.

B: Ich hab auch noch zwei, drei Rückfragen. Und zwar genau auf dieser Seite, das ist ja jetzt sozusagen, die soll zeigen die drei Datensätze, die ich jetzt in meiner Data Collection habe, richtig?

I: Genau.

B: Genau. Und welchen zeigt sie jetzt? Zeigt sie alle zusammen?

I: Die zeigt an, was ich auswähle, also grundsätzlich ist das hier die Übersicht "alle" und sobald ich hier irgendwie draufklicke, zeigt sie entsprechend halt den Wert an aus dem Domain, was einen gerade interessiert. Also angedacht, kann ich ja vielleicht so sagen, angedacht hatte ich, hier noch so eine Art Dashboard zu bauen, das so ein bisschen übersichtlich die Daten darstellt. So richtig mit meinen Testdaten konnte ich es aber nicht gut darstellen, deswegen hab ich es einfach rausgelassen. Also würdet ihr sagen, dass die datenübergreifende, also nein, die Darstellung von mehreren Daten gleichzeitig wichtig ist, oder reicht es so, dass man praktisch einzeln hier durchklicken kann und sagen: "Das ist das, okay, das ist das"? Oder ist es wichtig, dass die zum Beispiel direkt untereinander angezeigt werden oder vielleicht sogar, wenn das irgendwie automatisch geht, aggregiert werden oder wie auch immer?

B: Also ich finde in der Übersicht es nicht so wichtig, dass alle nebeneinander angezeigt werden. Ich kann mir vor allem vorstellen, dass, wenn...man hat ja dann vielleicht auch nicht nur drei, sondern vielleicht ein paat mehr, dass dann schnell unübersichtlich wird. Ich glaube, es wäre dann wahrscheinlich sinnvoll, dass man in der linken Spalte noch eine Möglichkeit hätte zu suchen, eine Stichwortsuche zu machen, weil so wie ich mich kenne, wenn ich Zugriff auf dieses coole Tool hätte, dann würde ich das nur ansammeln und rechts wäre dann wahrscheinlich, also meine Data Collection wäre wahrscheinlich ziemlich voll und dann müsste ich ewig scrollen, um halt dahinzukommen, ne? Das heißt, es wäre halt sinnvoll, oben einfach eine Stichwortsuche zu haben, und dann kann man, keine Ahnung, Catches oder so eingeben und dann zeigt es halt alle Datensätze an.

I: Also da zeigt er dann die aus der Collection, die Dokumente in meinem Fall, ich nenn sie Dokumente, die praktisch zu Catches relevant waren?

B: Genau.

I: Okay. Ja, das ist ein sehr guter Punkt, den hab ich mal aufgeschrieben.

B: Und das gleiche ist mir beim Filter auch noch aufgefallen, den du auf der Startseite hattest, rechts.

I: Ich geh mal zurück. Also ich such jetzt mal wieder nach Algerien, oder...? Okay.

B: Nee, du hattest diesen Filter auf der rechten Seite. Ich glaub, das war dann schon, wenn du auf einen Datensatz gegangen bist.

I: Den hier? Also die ähnlichen Dokumente?

A: Der Filter war auf der ersten, oder? Das waren doch die einzigen Filter, die du hattest.

. Appendices

I: Also auf der...

B: Hieß es nicht "Filter"?

I: Ich hab es, glaub ich, nicht "Filter" genannt. Also was ich hier gemacht hatte, war, einzuschränken, ich möchte die Datensätze aus nur SeaAroundUs haben oder nur hier zu bestimmten Schlagwörtern, die in den Ergebnissen gefunden wurden.

B: Ja, okay. Dann hab ich mir wahrscheinlich nur "Filter" aufgeschrieben, weil ich mir dachte: "Gibt es einen Filter?"

I: Tendenziell sind es ja Filter, also es ist ja jetzt nicht eine falsche Bezeichnung.

B: Genau. Was ich mich gefragt habe, könnte man da auch eine Suche einfügen? Weil je nachdem wie groß und wie umfangreich sozusagen meine Ergebnisliste ist, wird auch das ja länger auf dem Bildschirm, das heißt, es macht auch Sinn, da oben eine Stichwortsuche zu haben, wo ich dann zum Beispiel eingebe, also gerade Ländernamen tauchen ja wahrscheinlich recht häufig auf, ne? Und es kann halt sein, dass die Liste recht lang wird, die ist ja jetzt noch recht kurz, oder dass man durch das Fenster, durch das Tags-Fenster auch scrollen musste, das heißt, da fände ich, glaube ich, auch eine Stichwortsuche sinnvoll, dass ich dann sagen kann, ich will nur Algerien sehen oder nur Spanien oder...

I: Tendenziell kann man ja aber auch immer in die Suchanfrage jetzt schon reingehen, einfach die Wörter, die man relevant oder die man weiter gefiltert haben möchte, noch mit reinsetzt.

B: Ja, okay. Das stimmt natürlich.

I: Weil jetzt, also was jetzt aktuell passiert, da werden, glaube ich, die zehn oder zwanzig meistgenutzten angezeigt, das heißt, es ist nicht ewig lang, sondern halt nur die relevantesten.

B: Okay.

A: Also ich glaube, ich würd da auch...ich überleg gerade, ob ich... Hat man das schon mal gehabt, dass du im Filter noch mal eine Suchfunktion hast?

B: Na ja, so zum Beispiel bei der FAO-Seite wäre das teilweise sehr hilfreich gewesen...

A: Ja, okay.

B: Weil diese Kästchen eben nur so schmal waren und deswegen hab ich das im Hinterkopf, wo du dann eben noch mal auswählen konntest sozusagen, also das waren eben die Filter und dann ist regional, also Region, ist es irgendwie...hast du die Länder, hast du die Produkte und wenn man oben hätte sozusagen suchen können - ich glaube, mittlerweile haben sie das, dass du dann oben sozusagen bei Produkten eine Suchfunktion hast, und dann gibst du ein, keine Ahnung, "Ei", "Eier" oder "Bohnen" und dann wird dir halt alles, was "beans" ist, wird dir halt angezeigt. Und dann kannst du die alle mit reinnehmen in deinen Datensatz.

I: Ist die Suche denn so, die Suche, die die anbieten, ist die denn auch so, ich sag mal, übergreifend über alle oder sucht die nur nach dem Namen?

B: Nee, da sucht man sozusagen direkt im Dat-, also...

I: Ach so, also die sucht praktisch in den CSV-Dateien.

B: Genau. Also man geht zum Beispiel auf...ich weiß gar nicht mehr so genau, wie da

die Struktur ist...

A: Also du hast zum Beispiel einen Datensatz mit Konsumgütern und dann willst du aber jetzt nur Hering haben.

B: Genau.

A: Und dann musst du dich sonst halt durch Obst, Gemüse, Schinken, Käse, Milch, Bla scrollen oder dann sagen können "Ich möchte nur Subset Hering".

B: Genau, und du hast halt noch andere Filteroptionen, die du auch auswählen kannst, nämlich zum Beispiel "nur Hering für Deutschland" oder "für Europa" oder "für die ganze Welt". Und dann kannst du noch aussuchen, je nachdem in welchem Datensatz du gerade bist, ob du Export- oder Import-Daten willst oder beides oder Preise oder weiß ich nicht. Aber gerade wenn du halt dann einen Filter hast, deswegen hatte ich mir wahrscheinlich "Filter" aufgeschrieben, weil das für mich so, weil mich das daran erinnert hat.

I: Es sind ja Filter.

B: Genau. Es ist halt, finde ich, sinnvoll, zum Beispiel wenn man manchmal ja auch gar nicht genau weiß, ...

I: ...was einem die Daten so bieten?

B: Genau, und vielleicht auch wenn man gar nicht so genau weiß, wonach man sucht, sondern man sagt so: "Ich guck mal, was gibt's denn da so?" Und da ist es vielleicht ganz sinnvoll, wenn man halt sagen kann, ich guck jetzt mal hier oben Daten zu Algerien und dann... Ich weiß ja nicht, ist ja jetzt auch nur ein Beispiel mit den Ländern, aber es könnten ja auch andere Tags sein, die da auftauchen, dass man sagt: "Ah, und jetzt guck ich noch mal, gibt es in den Daten zu Algerien...?" Also sozusagen eine Subsuche wäre das.

I: Okay. Also kann ich ja ein Beispiel, glaub ich...was hab ich denn? Vielleicht finde ich da was zu "Anchovy" bei Algerien, weiß ich nicht genau. Na ja, nee, leider nicht. Aber grundsätzlich haben wir auch zum Beispiel hier die Fischarten aufgelistet und darüber kann man dann vielleicht noch weitere Sachen finden. Also das wollte ich nur so als Bemerkung..

B: Okay, also das heißt, man kann das theoretisch auch da oben dann.

I: Genau, es ist dann jetzt schon in dem Sinne, ich sag mal, zum Großteil möglich, indem man selber die Suchanfragen abietet, aber wenn du sagst, dass man manchmal gar nicht so genau weiß, wo man hinwill, sag ich mal, und einfach nur schaut, dann seh ich ein, dass man da vielleicht andere Hilfsmittel geben muss. Gut.

B: Vor allem, was ich halt gedacht hab, war, weil du vorhin meintest, dass ja manchmal die Biologen Daten verwenden, die eigentlich von Geologen kommen, das heißt, manchmal ist es halt auch überraschend, und das ist ja dann da nicht gefragt, man weiß ja manchmal gar nicht... Also woher weiß ich denn, dass ein Biologe einen Geologen-Datensatz verwenden kann oder will? Ja? Oder...ja und dann fehlt mir noch ein Tool, wo ich sozusagen...

I: Ein exploratives.

B: Ja, beziehungsweise vielleicht die Ergebnisse noch sortierter habe nach...Fachrichtung?

A: Wonach sind die Ergebnisse jetzt sortiert?

. Appendices

I: Die sind jetzt sortiert nach der Relevanz, also Schlagwörter praktisch, sortiert. **B:** Kannst du zum Beispiel nach der Fachrichtung? Weil ich seh ein, dass es jetzt, also sich auf wenige Datensätze sozusagen beschränkt, also noch recht übersichtlich ist, aber in dem Moment, wo es größer und komplexer wird, fände ich es, glaub ich, sinnvoll, wenn ich "Algerien anchovy" eingabe oder irgendwas anderes, dass ich sehen kann, ist dann die Frage, sozusagen aus welchen Fachrichtungen es Datensätze dazu gibt oder zu welchen Fachrichtungen sozusagen die Datensätze gehören, die ich sehen kann.

A: Das stimmt, das könnte, ich überleg gerade...also ich war jetzt gerade so bei Klima, dass man halt dann nicht, weiß ich nicht, immer nur Wetterdaten sucht, die Fischbestände beeinflussen, dass man dann auch nicht die mit drin hat, die physikalisch irgendwas anderes, weiß ich nicht, El Niñobeschreiben oder so. Wobei, die würden jetzt auch Fische beeinflussen, aber so ein bisschen, dass man vielleicht, ich hab überlegt, ob man das an die Quelle vielleicht koppeln könnte.

I: Aber dann müsste man ja wissen, welche Quelle was bedeutet, also jetzt zum Beispiel jemand, der FAOStat liest aber selber Geologe ist, weiß vielleicht gar nicht, was damit gemeint ist.

A: Ja.

B: Das stimmt.

A: Oder eine Quelle hat halt auch verschiedene Fachrichtungen, die bedient werden.

I: Genau. Von Geomar gibt es ja hier so eine...

A: PANGAEA?

I: Genau, PANGAEA. Die hat ja auch recht viel, alles zum Thema Meer und Ozean, aber schon diverse Fachrichtungen da drin.

A: Das stimmt.

I: Also für euch wäre relevant noch, dann eine Option zu haben, dass man wie hier zum Beispiel jetzt auf der rechten Seite das dargestellt ist, dass hier zum Beispiel die in der Kategorie der Daten, dann steht da zum Beispiel, genau so wie jetzt beim Publisher das steht, würde vielleicht da stehen: "Kategorie 'Biologie' haben wir 500 Datensätze jetzt gefunden" zum Beispiel.

B: Genau.

A: Wobei das tatsächlich schwierig werden kann, das zu unterscheiden.

B: Ja, ich glaub auch, dass die Kategorisierung wahrscheinlich recht heikel ist, aber ich glaube schon, dass es helfen kann. Also es muss ja nicht in der Ergebnisliste danach sortiert sein, aber dass man die Möglichkeit hat zu sagen: "Ich möchte jetzt nur mal die Artikel sehen, die aus der Biologie kommen." Oder nur mal die...also Artikel, die Daten sehen, die daher kommen, aus, keine Ahnung, weiß ich nicht, ne?

I: Okay. Gut, dann geh ich mal langsam weiter, damit wir auch fertig werden. Was sind denn so die Kriterien, nach denen ihr die Daten praktisch vorfiltert? Also was sind so die typischen Merkmale, auf die ihr achtet, wenn ihr die Daten betrachtet?

B: Bevor wir sie runterladen oder wenn wir sie analysieren?

I: Ich würde sagen, bevor ihr die runterladet. Also schon praktisch in der allgemeinen

"ich schau mal, was die Welt an sich sozusagen zu bieten hat" schon gleich zu wissen, das ist für mich nicht relevant, was ist vielleicht relevanter. Was sind so die typischen Merkmale, ist das eher so: Welche Quelle ist das? Oder zu welchen Ländern gehört das? Oder zu welchem...ja. Irgendwie geographische Informationen.

A: Also ich glaube, sicherlich die Quelle spielt eine Rolle, dass man schon sich eine Quelle sucht, die halbwegs renommiert und anerkannt ist.

B: Ja.

A: Für mich sind dann auch wichtig Zeiträume, klar, letztes Update, Einheiten. Länder vielleicht, je nachdem wie die Fragestellung ist, aber lieber differenzierter als zusammengefasst, also lieber auf Länderebene als dass ich jetzt Kontinentebene hab oder so, weil man eventuell noch selber selektieren kann, was einem wichtig ist und was nicht. Und ich glaube auch mittlerweile unter anderem auch erfahrungsbedingt aus dem WWF-Projekt Kompatibilität mit anderen Sachen, also wenn ich jetzt, weiß ich nicht, mehrere Datensätze als Quelle nehme und ich sehe, dass einer davon total rausschlägt, was die Kategorisierung angeht oder den Zeitrahmen oder die Fläche, die abgedeckt wird...

I: Also ob die irgendwelche Standards nutzen zum Beispiel.

A: Genau. Oder zumindest irgendwie das so zu haben, dass man das eine aggregieren kann, dass man das selbe Level hat. Also, und da sind halt wieder Einheiten wichtig und Bezeichnungen.

I: Also wenn man zum Beispiel sieht, die benutzen hier die und die Klassifizierung in den Daten als, ich sag mal, Überschrift zum Beispiel. Okay. Gut...ja, wenn ihr das Tool jetzt zur Verfügung hättet direkt, wie würdet ihr es denn in den Alltag integrieren? Würdet ihr es in den Alltag integrieren? Und wenn ja, in welchem Rahmen, was für Funktionen könnte es erfüllen?

B: Also ich denke, ich würde es integrieren, und wenn ich weiß, ich suche Daten, dann würde ich eher GeRDI als Google nehmen. Und wenn GeRDI mich dann nicht weiterbringt, würde ich zu Google gehen wahrscheinlich. Aber ich würde es schon, glaube ich, schon nutzen, weil ich glaube, das ist schon kürzer, als wenn man sich durch die ganzen Homepages friemelt. Also man kann hier direkt das, was man sonst eh bei SeaAroundUs machen müsste, kann man sich halt darüber sparen. Ja, würde ich, glaube ich, schon nutzen.

B: Ich würde es auch nutzen, ich finde vor allem diese Möglichkeit, dass du alle Daten sozusagen, du dir alle Datensätze sozusagen im Überblick angucken kannst, das finde ich ganz gut.

I: Also die Research-Page.

B: Genau, richtig. Weil sonst hat man halt immer seine ganzen Lesezeichen auf allen möglichen Internetseiten und muss dann immer gucken, wo war das jetzt und wo war das jetzt? Und da hat man sozusagen einen Anlaufpunkt, wo du hingehst und sagst, da sind die Daten, die ich halt irgendwie möchte. Was mich noch zu der Frage bringt, wie die Aktualisierung von Datensätzen dann dokumentiert wird, weil wenn ich jetzt zum Beispiel suche und ein halbes Jahr später draufgehe und gucke und sehe irgendwie, die FAO hat in

. Appendices

der Zwischenzeit aktualisiert, seh ich dann die aktualisierten Daten, seh ich, dass es ein Update gab, oder seh ich die alten Daten?

I: Gut, das ist dann natürlich eine Frage, je nachdem wie derjenige, der dafür zuständig ist, die Daten zu besorgen, das handelt. Grundsätzlich würde ich sagen, entweder sind die Daten zum Beispiel gruppiert nach Ländern, je nachdem wie die Daten aufgebaut sind, jetzt ist es, glaube ich, so...wobei aktuell wird der aktuelle Stand geladen, also insofern gab es das Problem bisher noch nicht, aber dann wäre es entweder so, dass es pro Aktualisierung ein neues Dokument gäbe, weil es zum Beispiel dann ein fester Zeitraum ist, jährlich zum Beispiel, oder es würde irgendwie vermerkt sein: "Dieses Dokument wurde aktualisiert am..." Also diese Daten werden auf jeden Fall schon gesammelt. Zumindest wann die, der Dienst sozusagen, der die ganzen Daten lädt von SeaAroundUs oder FAOStat, da wird schon gemerkt, wann die Daten bezogen wurden.

B: Genau. Ja, nee, doch, ich würde es auch nutzen, auf jeden Fall.

I: Wunderbar, das hör ich doch gerne. Ja, dann haben wir das eigentlich auch schon abgehandelt. Ich geh mal kurz durch meine Liste der Fragen, die ich hier noch habe. Genau, was sind denn so für euch die persönlich relevantesten und die nicht so relevanten Funktionalitäten? Also beispielsweise, ich liste mal ein bisschen auf: Also die allgemeine Suche, die Filterung innerhalb der Suche, die Datensammlung, das hast du eben schon angedeutet, dass das vielleicht das relevanteste sei, oder ist es eher die Detailseite zu einem gefundenen Dokument?

B: Soll ich? [*A stimmt zu*] Also die Detailseite zu einem gefundenen Dokument finde ich nicht so wichtig, weil das hab ich theoretisch auch, wenn ich auf den Link gehe und auf die Internetseite gehe. Was ich wichtig finde, ist in diesem Zusammenhang vor allem die Such-, also eigentlich alles drei, die ersten drei Sachen, die du genannt hast. Also die Suche, das Filtern ist aber auch wichtig, weil ohne Filtern, macht das Projekt für mich irgendwie, also macht es irgendwie keinen Sinn, ne? Dass ich halt all diese Daten zur Verfügung gestellt bekomme, da muss auch ein guter Filter da sein, dass ich halt die Möglichkeit habe, halt auch dann gut auszuwählen.

I: Aber da reicht dir eine reine Suche, wo man praktisch immer selber genauer angeben könnte, was man genau möchte, reicht da nicht aus, sondern du möchtest auch, dass dir vorgeschlagen wird: "Hey, wir können für dich folgende Untergruppierungen machen" oder so.

B: Genau, also ich glaube, ich fände eine Kategorisierung nach Wissenschaftsfach nicht also nicht verkehrt, da muss man halt sehr vorsichtig mit sein, dass man da nicht Sachen übersieht, oder es muss eben die Möglichkeit geben, das halt also optional sozusagen mit anzuklicken wie bei den Tags oder es halt zu lassen. Genau. Und die Datenkollektion. Was, glaube ich, auch noch sinnvoll wäre, ich glaube, das hatte [Name von A] schon gefragt, ist, ob ich die Daten auch gemergt runterladen kann. Das geht noch nicht, nä? Also ob ich die Daten, wenn ich jetzt drei Datenkollektionen habe, ob ich die dann auch direkt, nicht nur dass ich drei einzelne Daten runterlade, sondern ob ich die dann auch direkt zusammengefasst runterladen kann.

A9. Interview Transcript

I: Also das musst du definieren, was du mit "zusammengefasst" meinst. Meinst du, dass du einfach mit einem Klick alle Daten bekommst, oder meinst du, dass die Daten auch...beispielsweise aus Algerien haben wir folgende Daten, aus Libyen haben wir die anderen, die haben aber die gleichen Grundlagen sozusagen und die werden dann auch in eine, ich sag mal, große Tabelle wieder gepackt.

B: Genau, das meinte ich.

I: Okay.

B: Also das ist auch die Frage, ob das geht, ne?

A: Ich wollt' grad sagen, das ist, glaube ich, schwierig.

B: Aber wär' auf jeden Fall cool.

A: Also wenn die von einer Seite kommen, dann in der Regel sind ja Einheiten und so dann vergleichbar, also das würde gehen. Aber so wenn du jetzt zum Beispiel SeaAroundUs und FAO verschneiden willst, das passt ja hinten und vorne nicht. Das geht nicht.

B: Nee, genau. Aber das müsste eigentlich mit einem...das, vermutlich, müsste eigentlich leicht zu programmieren sein, oder? Weil, ich mein, dann muss wahrscheinlich gecheckt werden...

I: Das ist tatsächlich nicht so einfach.

B: Nee?

I: Also da ist...ja, man kann natürlich sagen, bei CSV-Daten ist es noch vielleicht einigermaßen gut machbar, weil man da sagen kann: "Okay, die haben die gleichen Überschriften und so."

B: Genau, richtig.

I: Aber selbst da ist es dann gefährlich, was passiert im Fehlerfall? Also wenn die nur zufällig die gleichen Daten haben, weil wenn jetzt zum Beispiel eine Datei heißt vielleicht "Name" und "Wert", die hat nur diese zwei Spalten, dann könnten da ganz viele Daten zusammengefügt werden, die eigentlich gar nicht zusammengefügt gehören.

B: Stimmt, das hat viel mit dem Labeling...ja, stimmt.

I: Genau so wie, ich sag mal, beim einen steht halt "Time", beim anderen steht auch "Time", aber der eine benutzt das und das Zeitformat, der andere benutzt nur die Uhrzeit oder sowas. Oder "Minuten seit 1970".

B: Das ist auch zu aufwendig sozusagen, weil ich weiß das selber, dass man dann immer in den Codebooks nachgucken muss, wie was jetzt gemeint ist, wie welche... Okay, hmm.

I: Gut, das heißt, noch mal um das kurz zu wiederholen: Wie ist da so die Abstufung? Also ich hab jetzt rausgehört, dass die Data Collection für dich das Wichtigste ist, oder?

B: Ja, und der Filter. Ich würde schon sagen, auch der Fil-, also beziehungsweise sagen wir mal so, dass unter dem GeRDI-Projekt mehrere Quellen sozusagen zusammengefasst sind, das finde ich, ist eigentlich das Wichtigste. Und dann ist aber das Anschlusswichtigste sozusagen, dass ich die Möglichkeit dann habe, auch die Sachen gut zu finden, und da gehört halt für mich die Suche und die Filterfunktion dazu. Und dass ich das dann zusammensammeln kann, ist halt super. Und den Überblick haben.

A: Ich finde auch, also gerade die Data Collection hilft halt, was du vorhin schon

. Appendices

meintest, es hilft halt zu vermeiden, dass man nicht 20 Seiten offen hat in seinem Browser und dann eventuell aus Versehen eine zumacht und dann sitzt man da und denkt sich: "Verflixt, wie bin ich denn jetzt an diesen Datensatz gekommen?" Und dann hat man sich den Link nicht notiert und dann hängt man da. Das lässt sich halt damit vermeiden, das ist ganz cool. Und ein Quell quasi, also jetzt angenommen, ich würde ein Paper schreiben, dann kann ich ja nicht sagen, "Ich arbeite mit Daten von GeRDI", sondern die Quellen sind ja quasi auch durch die Verlinkung zur Seite angegeben.

I: Genau, also das ist so ein bisschen das, was auf der Detailseite die Seitenleiste versucht anzugeben, wer ist halt der Autor, wer ist der Veröffentlichter? **A:** Also denke ich auch, dass dieser Warenkorb quasi das beste Feature ist, und halt dem vorgesetzt muss ja dann eine Suchfunktion sein, ne?

B: Finde ich auch gut.

I: Gut... Ja, also da ihr ja schon gesagt habt, dass es jetzt schon im Alltag helfen könnte, lass ich die Frage, ob noch Funktionalitäten fehlen, mal weg. Es sei denn, ihr habt da jetzt noch spontan... Nicht. Also ich sehe Kopfschütteln.

A: Vielleicht für später eventuell ein Sprachtool noch, also für den Fall, jetzt FAO und SeaAroundUs sind auf Englisch, aber wenn man vielleicht nationale Statistiken sucht und dann hängt man da nachher mit Tschechisch. Ich weiß nicht, ob...

I: Also die Unterstützung, dass es mehrere Sprachen...

A: Ja, zumindest dass diese beschreibenden Texte irgendwie auf Englisch sind, ich weiß nicht, ob...oder man muss sich dann eben damit rumschlagen, dass es nicht der Fall ist, aber...

I: Das heißt, entweder ist es in den Daten schon als, ich sag mal, internationale Beschreibung oder man hat vielleicht so ein Übersetzungstool, was mal funktioniert und mal nicht, damit man zumindest eine Übersicht bekommen kann?

A: Genau.

I: Okay.

A: Dass zumindest Stichwörter irgendwie erkannt werden. Halt eher für nationale Statistiken, glaube ich, wäre das relevant.

I: Gut. Ja, dann würde ich jetzt eigentlich noch mal nur allgemein fragen so: Wie würdet ihr das bewerten zwischen "0 - ist für mich aktuell gar nicht nutzbar" zu "10 - es wird mein Hauptrecherche-Tool in meinem Alltag", was würdet ihr da so vergeben?

A: Schon 'ne 8.

B: Ich würde, also jetzt angenommen, dass da noch mehr Daten eingepflegt sind sozusagen, würde ich auch sagen, eine 8.

I: Okay. Gut. Ja, dann noch mal komplett offen, noch irgendwelche Fragen, die ich nicht gestellt habe, die ihr gerne hören möchtet, oder möchtet ihr noch weitere Anregungen oder Feedback geben? Dann ist jetzt der richtige Zeitpunkt.

A: Wurde schon mal versucht, innerhalb des GeRDI-Projekts mit den anderen beiden Standorten, die Datenbanken von denen noch mit reinzunehmen? Also ich glaub, die Münchner sind noch mit drin, ne? Und Leipzig?

A9. Interview Transcript

I: Genau, es gibt Dresden, glaub ich, ich glaub, es ist Dresden, und München, die haben eigene Gruppen. Da wird die ganze Zeit schon mit denen interagiert, also da wird schon ordentlich interviewt und sich unterhalten. Hab ich aber wenig Aktien drin, also ich weiß nicht genau, wie da der aktuelle Stand ist, da hab ich mich seit Februar ungefähr von gelöst. Ich schau auch noch mal bei mir rüber, ob ich noch irgendwas vergessen habe... Genau, also...ja, allgemein die Frage: Findet ihr, dass das Finden von wissenschaftlichen Daten erleichtert wird durch das Tool oder einen Mehrwert darauf bietet?

A: Also ich hab es jetzt noch nicht echt ausprobiert, aber ich würde schon denken, dass es hilft.

B: Ja, ich würde auch denken, dass es hilft. Ich kann mir vorstellen, ich hab gerade noch gedacht, dass man gerade...also die offensichtlichen Sachen findet man ja auch über Google, aber so Sachen, so Statistiken, die so ein bisschen versteckt sind, dass das halt auch wertvoll wäre, die dann mit drin zu haben. Das heißt, dann wäre halt die Frage sozusagen, wie da euer Suchmechanismus funktioniert. Welche Sachen ihr aufnehmt, welche nicht. Dann stellte sich bei mir noch so die Frage, es gibt halt auch viele Daten, für die man bezahlen muss, auch vielleicht gering bezahlen muss, die aber vielleicht Interesse daran haben, dass die Daten über GeRDI tatsächlich einsehbar sind, zumindest für Forschungseinrichtungen, das heißt, es wäre vielleicht auch noch eine Frage, wie ihr das Projekt sozusagen, das bewirbt, und auf solche Sachen zugeht, auf solche Institute und Unternehmen. Wie geht ihr mit anderen Projekten um, die solche Datendienste anbieten? Also sollen die integriert werden? Genau, das ist jetzt so das, was mir noch spontan einfällt.

I: Okay.

B: Aber ich hab zum Beispiel jetzt Daten für Fischmehl gesucht und keine gefunden in offiziellen Statistiken oder es war sehr schwierig, und hab dann aber auch einen Dienst gefunden, der das auch anbietet, der auch sozusagen Daten zusammenzieht aus allen möglichen Bereichen.

I: Also ich lehn mich mal aus dem Fenster und würde behaupten, wir nehmen alles, was kommt.

B: Okay.

A: Aber zumindest das langfristige Ziel, dass man schon versucht, so andere große...

I: Ja, dass man möglichst breit gefächert ist. Genau. **B:** Ja, und das einzige, was ich noch, was mir noch so ein bisschen auf der Seele brennt, ist, wenn es dann so breit gefächert ist, also es ist einfach dann eine unendliche Vielzahl an Fachrichtungen, die dann wahrscheinlich da aufplopt, wenn ich Algerien eingabe. Es muss irgendwie eine gute...also jetzt ist es halt noch übersichtlich, aber ich glaube, es wird schnell unübersichtlich, und man bewegt sich immer so in seiner Filterblase, wir so für die Ressourcen-Ökonomie, dementsprechend Fischereidaten, und dann, glaube ich, hat man nicht so im Überblick, dass das recht schnell recht unübersichtlich werden kann.

A: Aber meinst du nicht, dass man jetzt, also angenommen...

B: Es kommt drauf an, wie man dann sucht.

A: Genau. Wenn du halt sagst "Catches: Algerien", dann würdest du ja vermutlich auch

. Appendices

da landen, auch wenn da noch Schulnoten und sonstwas auch zu finden wäre. Also ich glaube...

B: Das stimmt, aber wenn ich mir zum Beispiel Google Scholar angucke, bin ich manchmal halt überrascht, was mir angeboten wird, wenn ich doch schon eine sehr definierte Suche eingebe mit drei oder vier Schlagwörtern. Dann denk ich manchmal so: "Öh, okay?" Dann komm ich auf einmal bei Sachen raus, wo ich denke: "Nee, also das hat für mich jetzt, ist für mich überhaupt nicht interessant." Deswegen komm ich da so drauf.

I: Alles klar, dann schreib ich mir auf "Übersichtlichkeit bei vielen Datenquellen als Herausforderung für die Zukunft". Ja, alles klar, dann bin ich durch und bedanke mich recht herzlich, dass ihr euch die Zeit genommen habt.

A und B: Ja, gerne.