

## MASTER

### Creating a generalized framework to support the computational study of historical literature

Kortleven, David

*Award date:*  
2021

[Link to publication](#)

#### **Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain



Department of Mathematics and Computer Science  
Architecture of Information Systems Research Group

# Creating a generalized framework to support the computational study of historical literature

*Master Thesis*

David Kortleven

**Supervisors:**  
Bettina Speckmann  
Andrew Salway

Eindhoven, July 2021



---

## Acknowledgements

*Glory be to God for all the beauty around us and inside us. He is good, all the time.*

I would like to thank my supportive supervisor **Bettina Speckmann** for guiding me to and through this amazing project. I am also very grateful for the onboarding, support, brainstorming sessions, and guidance in simplifying my explanations by **Andrew Salway**.

Thank you **Arianna Betti** for providing me with your insights, and sharing your delightful music to listen to while studying. I thank **Maria Chiara Parisi** for spending all the time to help me understand the perspective of researchers, and giving very helpful suggestions in putting everything on paper.

I thank my parents for supporting me with everything that lead to this thesis. And finally I thank my loving wife who supported me through every moment of this process.





## **Abstract**

Research in various humanities fields, such as history of ideas, involves historical literature research. This starts with defining a corpus, i.e. a set of texts specified by bibliographical data, relevant to the research project and acquiring copies of these texts in order to analyze them.

Researchers get online access to increasingly large federated catalogs containing bibliographical data (such as WorldCat) and repositories of digitized texts (such as Google Books). A new approach to history of ideas, computational history of ideas, aims to employ these developments to enlarge the evidence basis for a wide-scope historical investigation.

A challenge this new approach faces is that the bibliographic data available online varies in quality and structure. This is problematic since humanities researchers, for example, must know precisely what edition of a book they are analyzing, and that they have identified all the books relevant to their research project.

This thesis presents a technical and user interface design of a framework that is created in collaboration with the Concepts in Motion team from the University of Amsterdam, representing the users of this framework. The framework supports researchers in the process of compiling text corpora using online catalogs and repositories. This way, bigger corpora with more accurate bibliographic data can be modeled and explored in a user-defined model. In addition, the framework keeps a detailed history of activity within the framework. This allows researchers to investigate and ensure the legitimacy of conclusions drawn from results from the framework. Other features include discussions and access control. The framework also prepares for the future integration of other features such as automated analysis of texts, and applications in other fields.

Next to presenting a design, this thesis also involves the implementation of a subset of the design in a production-ready way. This allows future projects to use the results of this thesis to deploy the framework for use in actual research projects.

# Contents

<b>Contents</b>	<b>ii</b>
<b>List of Figures</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Bibliographic Data . . . . .	2
1.2 Workflow of a Historian of Ideas . . . . .	3
1.2.1 Towards Computational History of Ideas . . . . .	4
1.2.2 Challenges . . . . .	6
1.3 Related Work . . . . .	6
1.3.1 Related Tools . . . . .	6
1.3.2 Data Storage Technologies . . . . .	7
1.4 Scope of Project . . . . .	8
<b>2 Description</b>	<b>9</b>
2.1 Users of the Framework . . . . .	9
2.2 Generalized Framework Description . . . . .	10
2.3 Definitions . . . . .	11
2.4 Requirements . . . . .	11
2.4.1 Functional Requirements . . . . .	12
2.4.2 Non-functional Requirements . . . . .	15
2.5 Design Scope . . . . .	15
<b>3 Architecture</b>	<b>16</b>
3.1 Main Challenges . . . . .	16
3.2 Approach . . . . .	17
3.2.1 Interface Points . . . . .	17
3.2.2 Application Backend . . . . .	17
3.3 Considerations . . . . .	19
3.3.1 Monolith versus Micro-Services . . . . .	19
3.3.2 Open Source versus Closed Source . . . . .	20
3.3.3 Deletion versus Archiving . . . . .	21
<b>4 Design</b>	<b>22</b>
4.1 Definitions . . . . .	22
4.2 Design Choices . . . . .	23
4.2.1 Interface Points . . . . .	23
4.2.2 Projects . . . . .	24

---

4.2.3	Authentication . . . . .	24
4.3	Dynamic Data Model . . . . .	24
4.3.1	Class Design . . . . .	24
4.3.2	Database Design . . . . .	27
4.3.3	Interaction . . . . .	28
4.3.4	UI Design . . . . .	28
4.3.5	Feedback Sessions . . . . .	31
4.4	Query Engine . . . . .	31
4.4.1	Process . . . . .	31
4.4.2	Query DSL Semantics . . . . .	32
4.4.3	Class Design . . . . .	37
4.5	Exploring . . . . .	40
4.5.1	Technical Design . . . . .	41
4.5.2	Database Design . . . . .	41
4.5.3	UI Design . . . . .	41
4.5.4	Feedback Sessions . . . . .	47
4.6	Importing . . . . .	48
4.6.1	Class Design . . . . .	48
4.6.2	Database Design . . . . .	49
4.6.3	UI Design . . . . .	49
4.6.4	Feedback Sessions . . . . .	52
4.7	Collaborating . . . . .	54
4.7.1	Class Design . . . . .	54
4.7.2	Database Design . . . . .	55
4.7.3	UI Design . . . . .	55
4.7.4	Feedback Sessions . . . . .	56
<b>5</b>	<b>Implementation</b>	<b>59</b>
5.1	Method . . . . .	59
5.1.1	Front End . . . . .	59
5.1.2	Back End . . . . .	60
5.2	Results . . . . .	61
<b>6</b>	<b>Conclusion</b>	<b>65</b>
6.1	Future Work . . . . .	66
	<b>Bibliography</b>	<b>68</b>
<b>A</b>	<b>General Class Diagram</b>	<b>70</b>
<b>B</b>	<b>Wireframes of the Graphical User Interface</b>	<b>72</b>
<b>C</b>	<b>Screenshots of the implementation of prototype</b>	<b>88</b>

---

# List of Figures

1.1	Group 1 entities of FRBR . . . . .	3
1.2	Workflow history of ideas . . . . .	4
1.3	Data model from the practice of computational history of ideas . . . . .	5
1.4	JSON example . . . . .	7
2.1	Conceptual data model definition . . . . .	11
3.1	General architecture of the framework . . . . .	18
3.2	General architecture of the framework pointing out the division between back end and front end deployment . . . . .	20
4.1	An altered version of the general architecture, describing the relation between API and GUI. . . . .	23
4.2	Class diagram describing the Dynamic Data Model . . . . .	25
4.3	Database design describing the Dynamic Data Model . . . . .	27
4.4	Wireframe of interface for creating a data object . . . . .	29
4.5	Wireframe of interface for selecting a referenced data object . . . . .	29
4.6	Wireframe of interface for showing a data object . . . . .	30
4.7	Query Engine conceptual process . . . . .	32
4.8	Query DSL Structure . . . . .	33
4.9	Class diagram of Query Engine Controller Classes . . . . .	38
4.10	Class diagram of Query Engine Query Classes . . . . .	38
4.11	Class diagram of Exploring module . . . . .	41
4.12	Exploring database design . . . . .	42
4.13	Wireframe of interface for exploring data objects . . . . .	43
4.14	Wireframe of interface for selecting columns to display in a table representing data objects . . . . .	43
4.15	Wireframe of interface for adding data objects to a collection . . . . .	44
4.16	Wireframe of interface for writing a query using the query DSL . . . . .	44
4.17	Wireframe of interface for writing a query using visual components . . . . .	45
4.18	Wireframe of interface for showing an overview of all collections . . . . .	46
4.19	Wireframe of interface for showing an overview of all data objects in a collection . . . . .	47
4.20	Wireframe of interface showing the concept of facets . . . . .	48
4.21	Class diagram of the Importing module . . . . .	49
4.22	Database design of the Importing module . . . . .	50
4.23	Wireframe of interface for starting an import . . . . .	51
4.24	Wireframe of interface for finishing an import . . . . .	52

4.25	Wireframe of interface of a curation session . . . . .	53
4.26	Class diagram of Collaborating module . . . . .	54
4.27	Database design of Collaborating module . . . . .	56
4.28	Wireframe of the interface to manage comments and actions . . . . .	57
5.1	Screenshot of the interface for creating an import . . . . .	62
5.2	Screenshot of the interface of a curation session . . . . .	62
5.3	Screenshot of the interface of the Exploring module while querying using visual components . . . . .	63
5.4	Screenshot of the interface of the Exploring module while querying using the query DSL . . . . .	63
A.1	Class diagram summarizing the whole framework . . . . .	71
B.1	Wireframe of interface for creating a data object . . . . .	72
B.2	Wireframe of interface for selecting a referenced data object . . . . .	73
B.3	Wireframe of interface for updating a data object . . . . .	73
B.4	Wireframe of interface for showing a data object . . . . .	74
B.5	Wireframe of interface to select a data type to start exploring data objects of that data type . . . . .	75
B.6	Wireframe of interface for exploring data objects . . . . .	76
B.7	Wireframe of interface for selecting columns to display in a table representing data objects . . . . .	76
B.8	Wireframe of interface for writing a query using the query DSL . . . . .	77
B.9	Wireframe of interface for writing a query using visual components . . . . .	77
B.10	Wireframe of interface for adding data objects to a collection . . . . .	78
B.11	Wireframe of interface for showing an overview of all collections . . . . .	78
B.12	Wireframe of interface for showing an overview of all data objects in a collection . . . . .	79
B.13	Wireframe of interface for exporting all data objects in a collection . . . . .	79
B.14	Wireframe of interface for starting an import . . . . .	80
B.15	Wireframe of interface for finishing an import . . . . .	81
B.16	Wireframe of interface showing all created imports . . . . .	82
B.17	Wireframe of interface for starting a new curation session . . . . .	82
B.18	Wireframe of interface of a curation session . . . . .	83
B.19	Wireframe of interface showing all curation sessions . . . . .	84
B.20	Wireframe of the interface to manage comments and actions . . . . .	85
B.21	Wireframe of the interface to manage actions . . . . .	86
B.22	Wireframe of the interface to manage comments . . . . .	87
C.1	Screenshot of the login interface . . . . .	88
C.2	Screenshot of the interface for selecting a project . . . . .	89
C.3	Screenshot of the interface for selecting an import or creating a new import . . . . .	89
C.4	Screenshot of the interface to finish an import . . . . .	90
C.5	Screenshot of the interface of a curation session . . . . .	90
C.6	Screenshot of the interface for setting up a curation session . . . . .	91
C.7	Screenshot of the interface of the curation session while curating import records . . . . .	91
C.8	Screenshot of the interface to start exploring data object of a data type . . . . .	92
C.9	Screenshot of the interface for filtering data objects using visual components . . . . .	92
C.10	Screenshot of the interface for filtering data object using a query in the query DSL . . . . .	93

## *LIST OF FIGURES*

---

C.11 Screenshot of the interface showing validation errors when an invalid query was entered . . . . .	93
C.12 Screenshot of the interface showing a single data object . . . . .	94

# Chapter 1

## Introduction

History of ideas is the field of research within the humanities that studies the historical development and decline of concepts (ideas). It entails studying historical works, in the form of books, articles, or any other textual works. This requires researchers to have access to a sufficiently sized corpus, a collection of written texts, containing enough works that comprise all resources relevant to their research. In addition, researchers need to have access to a sufficient amount of time to analyze and investigate such a corpus. Both requirements include resources which availability is often limited to researchers. With the rise of modern technology, such as increasingly powerful computers, the world wide web, researchers can get access to much larger corpora. Together with the development of computational tools used in disciplines such as Machine Learning and Natural Language Processing, a new approach to the study of the history of ideas, dubbed computational history of ideas (Betti and Van den Berg, 2016), is pioneered by the e-Ideas project from the Concepts in Motion team at the University of Amsterdam<sup>1</sup>.

These days, researchers have access to increasingly large digital federated catalogs of bibliographical data, and repositories containing digitized copies of historical texts. However, these resources are of variable quality, and different repositories use different structures to store the bibliographical data. Therefore there is a need of researchers for support in processing these large repositories, by having tools to carefully decide what resources are relevant to their research. This will help the researcher to create a larger evidence basis and be able to account for this basis. An approach to this challenge is presented in the form of hierarchical models that describe the editorial history of literature, as we discuss in Section 1.1. This proves to be a helpful approach to, for example, historians of ideas, who for their research must know precisely which edition of a book they are analyzing, and that they have identified all books that are relevant to their research, e.g. "all 18th-century books written in Latin on the topic of biology".

This thesis project aims at providing the technical and user interface design and implementation of a framework that supports researchers in the process of constructing corpora to perform analysis on, using online catalogs and repositories. The researcher can model and explore bigger corpora with more accurate bibliographic data in a user-defined model. The framework also includes an activity history that allows the researcher to investigate and validate the legitimacy of conclusions drawn from results from the framework. The design of the framework also includes a feature for discussions amongst users, and access control. The framework will be designed in such a way that it can be extended to support different tasks such as text analysis in the future. This project is performed in collaboration with the Concepts in Motion team, by using their input to construct a design, and testing the implementation using their insights.

---

<sup>1</sup><https://conceptsinnotion.org> (accessed: 2/7/2021)



Next, we describe the concept of bibliographic data, the workflow, and corresponding challenges of a (computational) historian of ideas in detail, followed by an overview of related work, and a description of the scope and structure of this thesis.

## 1.1 Bibliographic Data

An important concept in historical literature research is the one of bibliographic record. A bibliographic record is an entry in a catalog, bibliography, or any other bibliographic database, that describes a bibliographic resource such as a book, using bibliographic metadata such as the title, name of the author, and date of the first edition.

An example of a repository containing bibliographic data is WorldCat, provided by the Online Computer Library Center (OCLC) which combines bibliographic data of thousands of libraries into one repository (OCLC, 2021). Being one of the largest bibliographic databases, it is a great resource, however, the large amount of sources for the catalog is one of the main reasons that available bibliographic data is of varying quality and structure. For example, some bibliographic records miss specific metadata, or store the author of a work in the same attribute as the title. This varying quality and structure explains the need for a framework that allows researchers to make precise decisions and perform accurate normalizations on the available data for their research. An example of a repository that contains digitized copies of textual resources is Google Books, which contains an extensive set of resources that are already digitized. However also here, due to the massive amount of sources, the quality and structure of these sources are variable.

Finding a bibliographic record, indicating the existence of a work, such as a book, journal, or article, does however not necessarily provide all the insight into the exact history of a work that a researcher might need. A work, being an abstract representation of an intellectual or artistic creation, often is not a static object, just written once, producing a single artifact, but has its own history. A work is usually published multiple times in different editions, and each edition can consist of numerous physical copies. This hierarchy can be modeled by a relational model.

This concept and the use of a relational model describing bibliographic resources and their relevant entities is exemplified in detail by the Functional Requirements for Bibliographic Records (FRBR) (Functional Requirements for Bibliographic Records, 2016). FRBR aims to provide a standardized model to describe bibliographic records and their hierarchical relations as can be seen in Figure 1.1.

This model describes the following entities, which make up the so-called Group 1 entities:

1. A work, which represents a unique creation, for example, the tragedy *Romeo and Juliet* by William Shakespeare.
2. An expression, which represents an intellectual or artistic form of a work, for example, the book, the film, or the opera of *Romeo and Juliet*.
3. A manifestation, which represents an instantiation of an expression, such as the first edition of the book *Romeo and Juliet*, or the translation into German.
4. An item, which represents a physical exemplar of a manifestation. For example, a physical book of the German translation of *Romeo and Juliet*

A bibliographic record can describe any of these entities. However, it usually describes a manifestation, including attributes that, for example, point to a specific edition of a work. But a

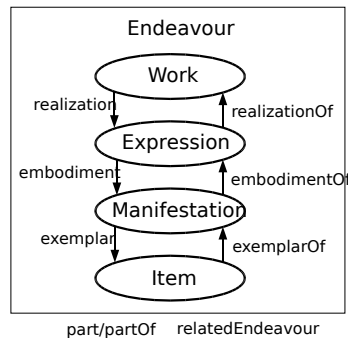


Figure 1.1: The Group 1 entities in FRBR, representing products of intellectual or artistic endeavor.

Source: Jakob Voss, WikiMedia

bibliographic record can also describe only a work without describing a specific manifestation of such a work.

Next to describing so-called Group 1 entities that are products of an intellectual or artistic endeavor (such as books, articles, or symphonies), FRBR also prescribes a model for Group 2 entities, entities responsible for intellectual or artistic content, such as a person or an organization, and Group 3 entities, subjects of intellectual or artistic endeavor, such as a concept, object, event or place.

There are already tools that support tasks related to literature research, such as Zotero for constructing corpora, or Microsoft Excel for maintaining bibliographic data in spreadsheets. These however do not suffice, as is discussed in more detail in Section 1.3.1. One of the main reasons is that these tools do not support the use of hierarchical models to describe bibliographic records. The prototype developed by the Concepts in Motion team does support a form of hierarchical models, but is not sufficient for a production environment.

## 1.2 Workflow of a Historian of Ideas

In order to understand what the framework resulting from this project should support, we describe in more detail how the research of a historian of ideas is shaped. Next, we explain how this workflow is adapted when considering computational history of ideas. This process can be summarized through the following steps, also visualized in Figure 1.2:

1. **Defining a research question:** The researcher defines a question or topic they want to investigate.
2. **Defining the corpus criteria:** The researcher describes corpus criteria, that work as guidelines to retrieve and include or exclude works that make up the exact corpus that is relevant for answering the research question.
3. **Find all resources matching the corpus criteria:** The researcher searches through (online) catalogs, such as WorldCat provided by the Online Computer Library Center (OCLC), uses bibliographies, or any other procedure to find bibliographic records, that match the corpus criteria.

4. **Find textual resources to construct the corpus:** The researcher finds all textual copies, items as described in Figure 1.1, described by selected bibliographic record matching the corpus criteria, to construct a corpus.
5. **Perform analysis and draw conclusions:** The researcher performs analysis on the constructed corpus, and uses the results as textual evidence that can validate their conclusions.

The last step, performing analysis and drawing conclusions, actually includes multiple substeps. We group these substeps into the final step since they are beyond the scope of this project.

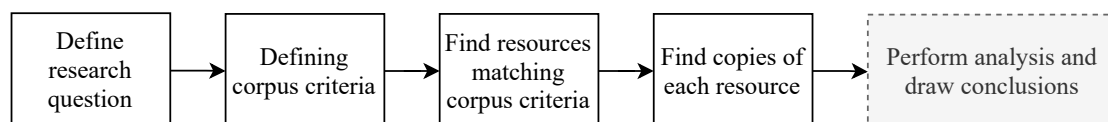


Figure 1.2: The typical workflow of research of a historian of ideas.

These steps are describing an ideal situation. In practice, each step of the research will include numerous challenges not yet addressed, and the steps will often not occur in a strictly linear fashion. In fact, a research project will involve an iterative process of going back and forth between each step, eventually leading up to the final conclusion.

### 1.2.1 Towards Computational History of Ideas

Corpus analysis in the traditional approach of history of ideas is a time-consuming process requiring close reading, and can also introduce bias since a human being is performing the close reading. Computational history of ideas, as a new approach to history of ideas, aims at supplementing the traditional approach by enlarging the evidence basis for a wide-scope historical investigation. To this end, the researcher applies computational techniques and tools to construct and analyze historical corpora. Using such computational techniques, the researcher can use larger corpora as textual evidence that validate their conclusion in a more efficient, and more transparent way. Considering the main topic of this thesis, namely supporting corpus construction, one of the most significant changes is that in Step 4, the researcher needs machine-readable corpora. Therefore, the researcher wants to find digitized copies, which represent items in a machine-readable way. The researcher can no longer use a physical book or unprocessed scans to do the analysis, since these are not machine-readable so computational tools can not be used on them. Only items that are digitized can be useful as input for the analysis. Additionally, to successfully use computational tools, digitized copies need to be of the highest quality. Having the highest quality digitized copies will namely optimize the efficiency and accuracy of the analysis of computational tools.

The notion of digitized copies, being machine-readable representations of an item, is not yet captured in the FRBR model. In practice, researchers in computational history of ideas have started to address the issue themselves, however. For example, the Concepts in Motion team have suggested adaptations to this model, leading to, for example, the model in Figure 1.3 (Parisi and Betti, 2020a). An extra entity is added, namely Digitized Copy, describing the digitized representation of a physical item. Extending the model with such an entity makes it more granular. This allows the researcher to answer certain questions such as, what is the best digitized copy of a work or manifestation that is available? This is useful since it contributes to the more general aim of building high-quality digital corpora for text-mining purposes, which

is an important requirement for computational history of ideas. Note that in addition to adding the digitized copy entity, this model also removed the Expression entity, as the expression of a work in literature study is always textual, and added an entity Person, representing authors, publishers, editor, and translators, borrowing from the Group 2 entities from FRBR.

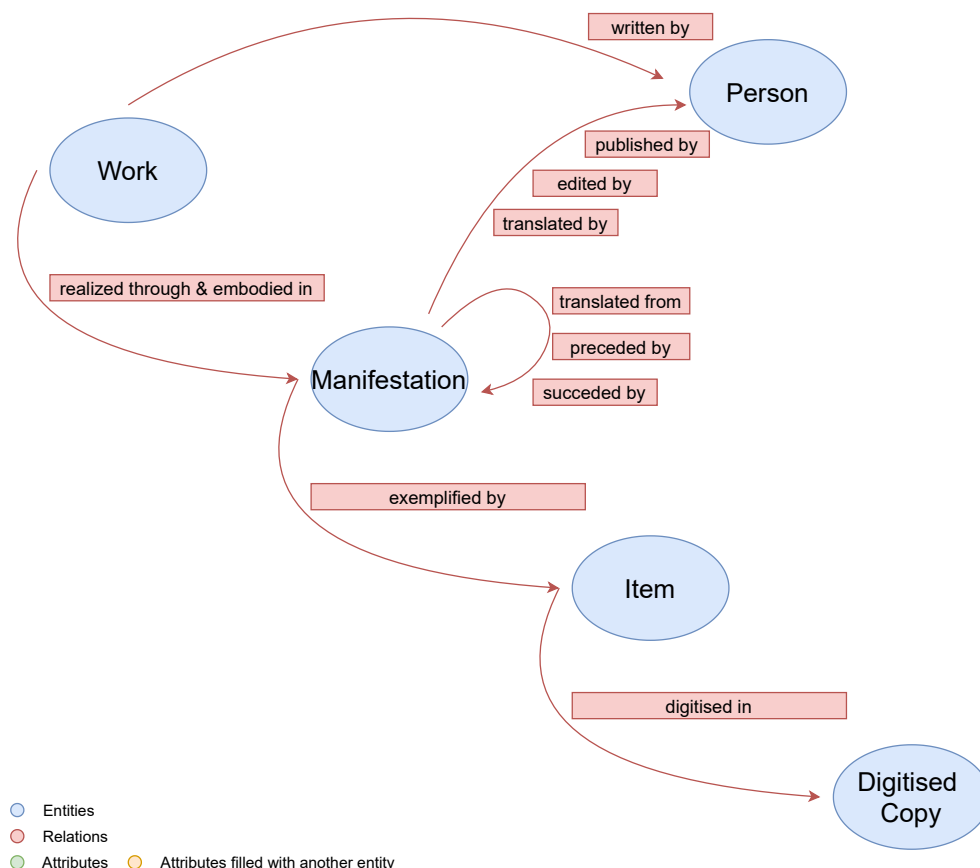


Figure 1.3: A model from the practice of computational history of ideas, based on FRBR (Parsi and Betti, 2020a).

An implication of being focused on digitized copies, and digitalizing history of ideas, is that computational history of ideas relies more on an implementation of a hierarchical model such as FRBR, or the model presented in Figure 1.3. Having such a hierarchical structure describing bibliographical records in place is crucial for the researcher to find all objects (manifestations, items, and ultimately digitized copies) matching the corpus criteria. This allows the researcher to efficiently explore the history of a work and compare different manifestations, items, and digitized copies of a work and take those into account when selecting the optimal resources for their research project.

Further changes when moving from history of ideas to its computational supplement are the methods of analyzing. New, computational techniques of analyzing a corpus, such as the ones used in the discipline of Natural Language Processing and other forms of Artificial Intelligence, can be used to analyze vast amounts of text. This is however beyond the scope of this project, but we can use it as a guide to design the framework in such a way that it can be extended to

support such functionality.

### 1.2.2 Challenges

There are a number of challenges that the framework around which this thesis revolves aims to provide support for. Firstly, the quality of bibliographic data often "are of variable quality and completeness", bibliographic records "use different cataloging schemes, and different languages" and do not involve structures such as hierarchical models like FRBR (Salway, 2021). Since a hierarchical model containing bibliographical data is important for researchers, a solution needs to be provided that allows researchers to structure bibliographic data from various sources of varying quality in a model that suits their research.

The exact shape of such a model is not fixed for every type of research project and may vary from researcher to researcher. We noticed this already in the FRBR model and an adapted version that researchers tailored to their use case in Figure 1.3. So the next challenge the framework faces is to provide a model that is complete enough to support the tasks a researcher needs to be supported in, but is also flexible to match the model a research project requires.

Additionally, to provide legitimacy and validatability for the conclusions that a researcher draws from the results of their research, the framework has to support a way to trace back how results were constructed. This includes the possibility to review decisions made in the past, and start discussions on them with collaborating researchers.

Since the framework that results from this thesis is not aiming to immediately be an all-encompassing framework that a researcher can use for any process in their research, an interface with external tools also needs to be part of the framework. This interface allows the framework to interact with other relevant tools that are not integrated within the framework (yet).

## 1.3 Related Work

In the following section, we describe previous work that is relevant to the goal of the framework or can serve as inspiration for the design and implementation of the framework.

### 1.3.1 Related Tools

There are a number of existing tools that provide parts of solutions to the described challenges. For example, the software tool Zotero provides the functionality to manage bibliographic records, specifically designed for keeping track of references during a research (Vanhecke, 2008). This allows the user to compile a list of bibliographic records, and filter among all these records, in search of a specific set of records that, for example, match corpus criteria. A limitation of Zotero however is that it is not possible to organize and enrich the records within a data model that is specifically relevant for the researcher. Therefore a researcher is limited to the restricted degree of freedom that Zotero prescribes, and will not be able to exploit a modeling of bibliographic records specific to their research project. A historian of ideas, for example, can not quickly explore all manifestations of a certain work, since there is no concept of a work, manifestation nor a relation between these in Zotero. Another tool that researchers can use is Gneiss (Chang and Myers, 2016). This tool allows users to explore hierarchical data in a spreadsheet without destroying the hierarchical structure. However, this tool does only allow for exploring already structured data, and does not provide the required functionality for (re)structuring bibliographic data from various sources.

Another set of tools that relates to the described problems are the so-called Content Management Systems (CMS). These, often commercial, systems primarily aimed at serving websites, sometimes also allow for creating a dynamic data model. A CMS is often not completely suited for managing data dynamically, without it being directly coupled to a website as output. We can however distinguish Headless CMS, a class of CMS that aims to decouple the output of a system (usually a website) from its Content Managing component. In theory, this would allow for entirely "website-unrelated" applications of such systems, such as providing a dynamic data model that implements FRBR. The design and implementation of such systems can serve as inspiration for the dynamic data model that is required for the framework this project creates.

Researchers from the Concepts in Motion team noticed the limitations of tools such as Zotero, Gneiss, and Excel, and started to think about a solution to this by, for example, using a relational database. They developed a prototype application that implements a hierarchical model as presented in Figure 1.3, using a relational database (Parisi and Betti, 2020b). This prototype shows the potential of using a hierarchical relational model for (computational) historians of ideas, since it, for example, allows researchers to explore all available items of a certain work. The prototype also serves as an important inspiration for the data modeling features of the framework designed in this thesis.

### 1.3.2 Data Storage Technologies

For the implementation of the framework, we take inspiration from other ideas and systems that were developed before. When thinking about the requirement of a flexible data model, the relatively recent development of NOSQL databases is relevant (Nayak, Poriya and Dikshay, 2013). Some types of NOSQL databases, opposing or complementing the classical SQL databases or relational database management systems (RDBMS), have as a main feature that they do not necessarily fix an application to a database schema, that requires manual actions to migrate to a new state. In addition, NOSQL databases aim to address scalability challenges that traditional database approaches face (Strauch, 2021). These properties of NOSQL databases seem to align very well with the desire for a Dynamic Data Model. Namely, a Dynamic Data Model can inherently not be defined in an RDBMS schema during the development of the framework, since the exact schema varies from project to project. Having a database with a flexible schema, that does not need explicit migration of the database can be a helpful alternative.

Another relevant development is the use of JavaScript Object Notation (JSON) in relational databases. JSON objects are structured strings, that can be used to store data, an example is shown in Figure 1.4. Traditional databases, such as Oracle Database, Microsoft SQL Server, and PostgreSQL, started to integrate with JSON (Petkovic, 2017). This means that they provide field types for JSON that can be indexed, and thus queried efficiently. Previously JSON would just be stored as a string, which made it difficult, if not impossible to query the JSON field in a meaningful way, but this development of traditional databases relieved this limitation. Now that this feature is available, an application can use the strengths of a traditional and mature RDBMS, while also effectively having a partly flexible schema.

```
{
  "title": "A bibliography on the rise of strawberry logic",
  "volume": 41,
  "cityOfPublication": "New Lutjebroek",
  "namedCities": ["Amsterdam", "Kopenhagen"]}
}
```

Figure 1.4: An example of JavaScript Object Notation (JSON)

Elasticsearch is also a relevant and interesting open-source piece of software. Elasticsearch is a "distributed search and analytics engine", the core of the Elastic Stack, which includes other tools that are relevant to data collection (*Elasticsearch Query DSL* 2021). An interesting feature Elasticsearch has is the so-called Query Domain Specific Language (DSL), based on JSON. This allows the user to construct data queries in JSON, that can be arbitrarily complex through the use of *Compound query clauses*, which allow the user to combine multiple simple queries, e.g. selecting all records with a specific value for an attribute. What makes Elasticsearch inspiring for this project is that the flexibility of its Query DSL does allow for the querying of data models without a specific schema.

## 1.4 Scope of Project

The aim of this project is to provide a framework as a foundation for the automation of, and supporting tools for, the research of history of ideas. This thesis focuses on providing functionality that supports researchers in building corpora for their research, and furthermore takes the future addition of functionality, for example, for analyzing corpora, into account when creating a design. The final product generalizes towards the use not only in different research areas that share similar textual analysis on significant amounts of digital textual data but also towards applications outside academia, such as libraries, museums, and other institutions that may benefit from components of the framework. For example, some components of the framework, such as a dynamic data model, could also be used to model the collection of a museum, in order to help finding objects in the collection to create coherent exhibitions.

This project is executed as an iterative process, where each iteration of the project involves the design and implementation of a component or module within the framework. This is done in collaboration with the aforementioned Concepts in Motion team, represented by Arianna Betti, Andrew Salway, and Maria Chiara Parisi. Next to being the instigators of this project, they also act as example users, having good insight into what potential users expect to see in the framework, and what challenges they will face. The collaboration consists of several forms. Firstly, the description and requirements are based on all the work that has been performed on describing the e-Ideas project by the Concepts in Motion team. Secondly, the design based on this description is presented several times to the Concepts in Motion team, asking for their insights in order to improve and fine-tune the design. Finally, in the implementation phase of the iterations in this project, the Concepts in Motion team provides feedback to improve the implementation or suggests new ideas for the framework for future iterations. This thesis will describe the final outcomes of each iteration, without explicitly giving minute details about all the feedback sessions with the Concepts in Motion team. However, some interesting results of these sessions will be mentioned throughout the design and implementation chapters.

This thesis is structured as follows. In Chapter 2, a detailed description of the requirements of the framework is provided. Chapter 3 draws the general outline of the design of the framework, and discusses general decisions that are relevant for the rest of the design and implementation. Chapter 4 provides a granular design of the framework. The components of the design that were also implemented as part of this thesis are then showcased in Chapter 5. Chapter 6 concludes by summarizing the work done and elaborating the future work, be it design or implementation, that can be done in this project.

## Chapter 2

# Description

The following chapter provides a detailed description of the project, starting by elaborating on who the users of the framework will be. Following that, we focus on the tasks that the framework has to support, generalizing from the tasks of a historian of ideas as described in Section 1.2. Next, we describe the user requirements for the framework we are designing, explaining the general tasks of the framework. Finally, the exact scope of the design and implementation of this project is defined, delimiting what features of the framework will be designed in detail, which features will be implemented in the prototype, and which parts will be left to future work.

### 2.1 Users of the Framework

We can define two groups of users of the framework, application users and framework developers. We define application users as the group of mostly non-technical users who will use a graphical user interface (GUI) to interact with the functionality of the framework. These users are, for example, represented by the aforementioned historians of ideas, but can be generalized to librarians and researchers doing literature research, desiring to construct corpora in a systematic and accountable manner, and potentially an even broader group of users as we will discuss later. In general, their needs are centered around the completeness and usability of the functionality in an intuitive way. When we refer to users of the framework, we refer to these application users.

Framework developers are defined as the technically skilled group of users, who can extend the framework with new functionality, or integrate external tools with the framework. They interact with the framework mostly through a non-graphical interface to support application users who require more advanced functionality than the framework supports by default.

When defining the required functionalities of the framework, we base this mostly on the desires and expectations of the application users. The desires and expectations of framework developers do however impact the implementation design choices significantly. For the extend to which the framework is maintainable, extensible, and modular to framework developers is a decisive factor in the successful continuation of the development of the framework.



## 2.2 Generalized Framework Description

As mentioned before, we want the framework to support not just a specific group of historians of ideas, but rather every historian of ideas, and even other groups of users that can benefit from (a subset) of functionalities that the framework provides. We think, for example, of historians or philosophers that also analyze historical corpora. But potentially we can even think of museums that want to structure a collection, as discussed before. Therefore, we generalize the tasks of a historian of ideas as seen in Figure 1.2 to the following general tasks that the framework has to support:

1. **Research Workflow:** The framework should provide the user with a workflow, that guides the user through their research process in a structured way. This workflow should be modifiable to the specific methodology a researcher is using, since this methodology might vary from researcher to researcher and from research project to research project. For example, a philosopher or historian of ideas is usually interested in tracing changes in concepts, while linguists are often focused more on the evolution of a term in a language, and archaeologists commonly study history of physical objects. Each type of research can involve different workflows. A workflow consists of a set of tasks divided over several steps that can be assigned or iterated. When looking at the workflow description of history of ideas in Section 1.2, this task corresponds to Steps 1, 2 and 5.
2. **Dynamic Data Model:** The framework should provide the user with a dynamic data model. This data model is modifiable to a specific project, such that it can contain the data in a way that it represents them in a format relevant to the researcher's needs and questions. One important aspect of this model is the requirement to be able to define relations between objects. This allows a historian of ideas, for example, to link a manifestation to a work, following the data model in Figure 1.3. This enables the historian of ideas to explore these explicit relations and, for example, explore every manifestation of a work to reconstruct the bibliographical history of a work. Next to having this dynamic data modelling functionality, we also require the framework to have import and export functionality at scale, to allow interaction between, and migration from and to external tools. When looking at the workflow description of history of ideas in Section 1.2, this task corresponds to Step 3 and 4 since having a dynamic data model allows the researcher to import and curate resources into this dynamic data model.
3. **Data Exploration:** The framework should provide the user with an interface to explore all the available data. The user should be able to explore the data that will be present in the data model, to find relevant data efficiently. For example, a historian of ideas has to be able to see all manifestations of a work, such as all editions of *Romeo and Juliet*, or find all works with certain attributes, such as all works published in the 19th century. The user should then also be able to create collections of data, and export these from the framework. When looking at the workflow description of history of ideas in Section 1.2, this task corresponds to Step 3, 4 and 5, since having the functionality to explore data in the Dynamic Data Model allows to find resources that match the corpus criteria, and analyze them to create supported conclusions.
4. **Access Control:** The framework should provide a method of access control, to ensure, or at least enhance, the integrity of activities and their results within the framework. When looking at the workflow description of history of ideas in Section 1.2, this task corresponds to Step 5, in the sense that results can be accounted for, and integrity of the data can be controlled.

5. **Collaborating:** The framework should provide supporting tools to improve the collaboration of multiple researchers within a single project. Important aspects are a way of commenting or discussions, on certain topics. But also the more basic requirement that multiple users can access and manage data within the framework simultaneously is necessary. Next to that, the framework should show all activities that occurred within the framework, in order to allow for an activity audit. This can help the user to derive how certain results were established, but can also help in collaboration. When looking at the workflow description of history of ideas in Section 1.2, this task corresponds to Steps 1 and 5, next to supporting collaboration on every other task.

As will be explained in more detail in Section 2.5, in this project we focus mostly on the tasks of data modelling, data exploration, and collaborating. In the following sections, we describe specific requirements.

## 2.3 Definitions

Before elaborating on the requirements, there are a few definitions to provide.

1. When mentioning a **data model**, we consider a data model to consist of a number of **data types**, akin to classes in object-oriented programming languages, which all have can have a number of **data attributes**. A data type has a name, while a data attribute has a name and a type, for example, a string or number. In addition, the data model can be 'filled' with data through **data objects**. A data object is an instantiation of a certain data type, and can have values for each data attribute of the data type. This is also shown in Figure 2.1.
2. A **user** is considered to be the Application User as described in Section 2.1.
3. The **initial stage** or **this stage** of the framework, refers to the phase of designing and developing the framework in the scope of this Master's thesis. Later stages can include more advanced or extensive functionalities than this thesis.

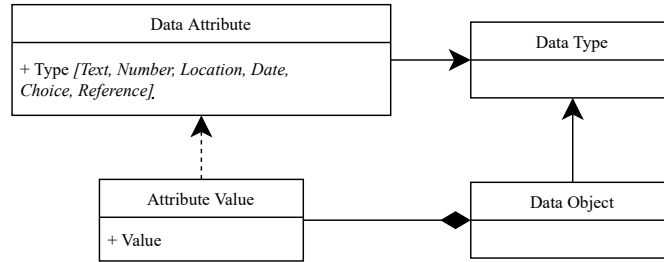


Figure 2.1: The relations between concepts that make up the data model definition

## 2.4 Requirements

We describe the specific requirements in two parts: functional requirements, describing explicit functionalities the framework has to provide, and non-functional requirements, describing desired non-functional properties of the framework. We do not provide priorities for each requirement, for example, by using MoSCoW-based methods (Ahmad et al., 2017), since this project

revolves around setting up a design and implementation process for the framework, and does not aim to have a completely implemented product as the end goal. Prioritizing the desired order of implementation of certain features is something that has to be done during the phase where these features potentially might be implemented. Related to this, note that this list of requirements is in no way definitive for the development of the project, but serves as a starting point for future development. This will also lead to some deliberately unspecific requirements that will not be addressed in detail in this project. Some requirements will have *examples in italic* to exemplify their impact.

### 2.4.1 Functional Requirements

We group the Functional Requirements based on the tasks described in Section 2.2:

#### Research Workflow

The following requirements provide a general and unspecific description of what the research workflow functionalities of this framework may involve. This will not be the focus of this thesis.

1. **Workflow per Project:** The user has to be able to define a workflow per project.
2. **Steps in a Workflow:** The user has to be able to define steps consisting of tasks in a workflow.
3. **Assign Tasks to Users:** The user has to be able to assign users to a task of a step in the workflow.

#### Dynamic Data Model

The requirements of the dynamic data model functionalities of the framework are similar to the description of a traditional database system. However in general these requirements require that the data model has to be customizable at run time, and not predefined in some schema as in traditional database systems (Köhler and Link, 2018).

1. **Data model per Project:** The user has to be able to define a data model (consisting of data types and data attributes) per project.
2. **Data type definition:** The user has to be able to define data types. *For example, a data type named Work*
3. **Data attribute Definition:** The user has to be able to define data attributes for a data type, where the attribute has to be one of the following types:
  - (a) **Text:** *E.g. an attribute named Title*
  - (b) **Number:** *E.g. an attribute named Volume*
  - (c) **Date:** *E.g. an attribute named PublicationDate for the data type Manifestation*
  - (d) **Multiple Choice,** allowing the user to pick a value from a predefined set of values.
  - (e) **Reference,** allowing the user to pick an object of a certain data type. *E.g. an attribute named Work for the data type Manifestation, referring to the data type Work*
4. **Validate data object:** The user has to be able to define validations for data attributes, of the following types:

- (a) **Required**, requiring data objects to have a value for the respective data attribute. Applicable for all data attribute types.
  - (b) **At least**, requiring data objects to have a value that is at least some other value. Applicable for Number and Date data attribute types.
  - (c) **At most**, requiring data objects to have a value that is at most some other value. Applicable for Number and Date data attributes.
  - (d) **Length at least**, requiring data objects to have a value that has a length of at least some other value. Applicable for Text and Reference data attribute types.
  - (e) **Length at most**, requiring data objects to have a value that has a length of at most some other value. Applicable for Text and Reference data attribute types.
  - (f) **Length exactly**, requiring data objects to have a value that has a length of exactly some other value. Applicable for Text and Reference data attribute types.
5. **Create data object:** The user has to be able to create a single data object of a certain data type
  6. **Edit data object:** The user has to be able to edit the values of a specific data object.
  7. **Validation on save:** The user has to receive validation errors when they want to create or edit a data object but enter data that does not match all validations of the data attributes of the data type of the data object.
  8. **Import CSV:** The user has to be able to import a comma-separated values (CSV) file, and create data objects from each record in the CSV file. *The user can import an export from Zotero into the framework and add all records as *Manifestation* data objects.*
  9. **Archive data object:** The user has to be able to archive a single data object. Archived data objects will not show up in the framework, unless explicitly referenced.
  10. **Unarchive data object:** The user has to be able to unarchive a single data object.
  11. **Archive multiple data objects:** The user has to be able to archive multiple data objects at once.
  12. **Unarchive multiple data objects:** The user has to be able to unarchive multiple data objects at once.

### Data Exploration

The following requirements describe how the framework has to support the exploration of data within the Dynamic Data Model.

1. **List data objects:** The user has to be able to see all data objects of a certain data type
2. **Filter data objects:** The user has to be able to see all data objects of a certain data type that have a matching value for a specific data attribute. The matcher has to be one of the following:
  - (a) **Exactly.** The value of the data object for the data attribute has to be exactly a certain value. Applicable for all data attribute types
  - (b) **Contains.** The value of the data object for the data attribute has to contain a certain value. Applicable for Text data attribute types

- (c) **At Least.** The value of the data object for the data attribute has to be at least a certain value. Applicable for Number and Date data attribute types.
  - (d) **At Most.** The value of the data object for the data attribute has to be at most a certain value. Applicable for Number and Date data attribute types.
  - (e) **Length At Least.** The length of the value of the data object for the data attribute has to be at least a certain value. Applicable for Text and Reference data attribute types.
  - (f) **Length At Most.** The length of the value of the data object for the data attribute has to be at most a certain value. Applicable for Text and Reference data attribute types.
  - (g) **Length Exactly.** The length of the value of the data object for the data attribute has to be at exactly a certain value. Applicable for Text and Reference data attribute types.
3. **Filter data objects nested:** The user has to be able to see all data objects of a certain data type that refer to a specific data object through a Reference data attribute, that has a matching value for a specific data attribute.
  4. **NOT filter:** The user has to be able to see all data objects that do not match with a specific matcher.
  5. **AND filter:** The user has to be able to see all data objects that do match all of a set of matchers.
  6. **OR filter:** The user has to be able to see all data objects that do match any of a set of matchers.
  7. **Show data object:** The user has to be able to see all values for all data attributes of a specific data object
  8. **Show referring data objects:** The user has to be able to see all data objects that refer to a specific data object through any data attribute.
  9. **Create Collections:** The user has to be able to create a Collection that can contain data objects.
  10. **Add data object to Collection:** The user has to be able to add a data object to a Collection.
  11. **Remove data object from Collection:** The user has to be able to remove a data object from a Collection.
  12. **Export Collection as CSV:** The user has to be able to export a Collection as a CSV file, which will contain all data of the data objects in that Collection.

### Access Control

The following requirements briefly describe how Access Control in the framework has to function. These requirements are deliberately unspecific, since they will not be the focus of this project.

1. A user can only access the framework once they have authenticated.
2. An admin user can define which user has access to which features of the framework.
3. A user can only access the features of the framework they were granted access to.

## Collaborating

The following requirements summarize how collaboration has to be supported by the framework.

1. The user has to be able to create a comment on any resource within the framework. *For example, a user can add a comment on a data object*
2. The user has to be able to reply with a new comment to a comment.
3. The user has to be able to create a comment on any action. *For example, a user can reply to the creation of a data object by another user.*
4. The user has to be able to see which actions have been performed within the framework.
5. The user has to be able to see which actions have been performed with respect to a specific resource within the framework. *For example, a user can see who created a data object, who updated a data type's name, or who created a data attribute.*

### 2.4.2 Non-functional Requirements

The following section describes requirements that do not relate to specific functionalities of the framework.

1. The framework has to be easy to extend in functionality.
2. The framework has to be able to integrate with any external tools.
3. The GUI of the framework has to be accessible through a web browser.
4. The GUI of the framework has to be responsive, meaning it should be workable on devices with smaller screens, such as smartphones or tablets.
5. The interfaces of the framework have to respond nearly instant, i.e. provide responses within 2 seconds.

## 2.5 Design Scope

As mentioned before, this project is not a full implementation, nor a complete design of the framework, this is partly because it would not be feasible for a 6 months Master's thesis project. At its core, this project aims to create the basic yet essential foundations for an ongoing project. This ongoing project will be expanding and adapting the functionalities of the framework according to its use over time and the availability of required resources such as development time.

This project scope includes the design and (partial) implementation of the Dynamic Data Model and Data Exploration tasks. Additionally, the scope of this project includes only the design of the Collaborating task, which provides a clear starting point for implementation in a later stage.

The scope of this project will not include Access Control. This is a feature that is desirable in a production application of the framework, where actual users will use it in an uncontrolled environment which will not yet be the case. This has been discussed with the Concepts in Motion team, and can be implemented in the near future when the environment of use requires it.

## Chapter 3

# Architecture

The architecture of the framework described in the following sections addresses strategic decisions made in order to get a coherent design and implementation fulfilling the requirements as elaborated in the previous section. First, we will summarize the challenges that arise from the description into a number of modules. Next, we will use these modules to draft a general approach to the design of the framework.

### 3.1 Main Challenges

The main challenges that amongst others arise from the description can be summarized in the following modules:

1. The framework needs to provide a dynamic data model. *This addresses the dynamic data model requirements as described in Section 2.4.1.*
2. The framework needs to provide a flexible interface to query (explore) data within the Dynamic Data Model. *This addresses the data exploration requirements as described in Section 2.4.1.*
3. The framework needs to provide a history of every action performed within the framework *This addresses part of the collaboration requirements as described in Section 2.4.1.*
4. The framework needs to provide a supportive collaborating function, in order for co-users to communicate. *This addresses part of the collaboration requirements as described in Section 2.4.1.*
5. The framework needs to provide a way to integrate with external tools *This addresses non-functional requirements<sup>2</sup> from Section 2.4.2.*

Challenges we will not address in this project, as described in Section 2.5, are, amongst others listed below. This list is deliberately unspecific since it requires further investigation before more detailed descriptions can be given.

1. The framework needs to provide detailed access control
2. The framework needs to provide a dynamic workflow to guide users through a user-defined workflow.

Next to these challenges, we also need the framework to be minimally internally coupled, in order to make future development but also integration with external tools easier.

## 3.2 Approach

To devise a design for the framework, we first propose a general architecture, that will serve as the main guideline for our design. Figure 3.1 presents the general architecture. Within the framework we distinguish between the core application, called the Application Backend, and two interface points, to allow interaction with the framework. In the figure we also see the two types of users represented. The application user only interacts with the Graphical User Interface (GUI), while the framework developer either interacts with the API by building integrations that interact with the API, or develops or extends modules within the Application Backend.

### 3.2.1 Interface Points

We distinguish two interface points. The first interface point is a Graphical User Interface (GUI). This intuitive interface allows (non-technical) application users to interact with the framework. The GUI will use all the functionalities present in the Application Backend, and expose them graphically to the user, as a web application, accessible through a browser. Even though it will be discussed in detail in the implementation description in Section 5.1, it is worth mentioning that the GUI uses the Application Programming Interface (API) as a means to interact with the Application Backend.

The second interface point is the API. This non-graphical programmatic interface allows other software to interact with the Application Backend. This enables the integration of external tools with the framework. The initial implementation of an API will be one-directional, meaning data can be requested, and mutations to the data can be made from an external tool through the API. An addition in a future stage of the framework could be that the API also proactively informs external applications through a Publisher-Subscriber mechanism, or by using webhooks.

### 3.2.2 Application Backend

The Application Backend contains all the functionalities that make up the core of the framework. We can specify the six modules that make up the initial stage of the framework: **Dynamic Data Model**, **Query Engine**, **Exploring**, **Importing**, **Action Framework**, and **Collaboration**.

- The **Dynamic Data Model**, will provide a data model that can be dynamically, i.e. at run time, changed. A data model consists of data types with corresponding data attributes. Then, using this data model, a user can create data objects. A data object is of a specific data type, and can have data for each data attribute of that data type. In Section 4.3, a detailed design is shown, explaining the functionality of the Dynamic Data Model module in more detail.
- The **Query Engine** can query data within the Dynamic Data Model, using a Domain Specific Language (DSL). This query DSL is a text-based language, with which the user can construct queries that can be parsed and processed by the Query Engine. This effectively allows the user to query data from the Dynamic Data Model. Section 4.4 describes the Query Engine in more detail.



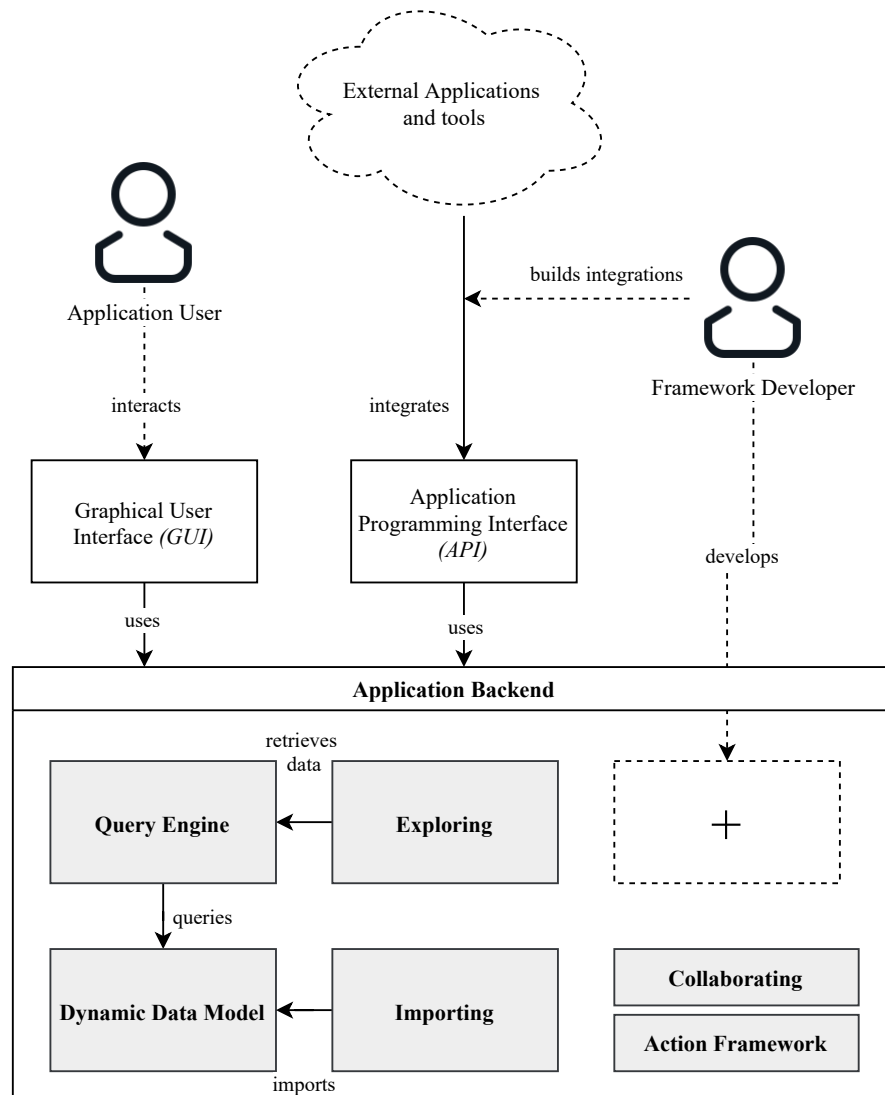


Figure 3.1: The general architecture guiding the approach for designing and implementing the framework.

- The **Exploring** will provide supporting tools around the Query Engine. It will, for example, allow to store queries, such that the user can load these queries later and revisit the result, or investigate changes in results after the data has changed. Additionally, this module also allows the user to create collections, which consist of a set of data objects, and create export files from these collections. Section 4.5 describes the exploring module in more detail.
- The **Importing** module provides functionality to import data from CSV (and in the future also differently typed) files, and create multiple data objects from this file. Section 4.6 describes the Importing module in more detail.

- The **Action Framework** is used to store the history of anything that happens within the framework. For example, if a change was made to a data model, if some data was created, if a specific exploration query was stored, these events are all stored as an action in the Action Framework. These actions can then be used as a subject of discussion, which is part of the Collaborating module, but also as a means to audit the history of a project. This can be used to account for results from the framework, and trace back and correct any mistakes.
- The **Collaborating** module takes care of functionality that relates to communication and collaboration between users. In the scope of this thesis, this will entail the option to write comments for any objects within any module of the framework, such as data objects within the Dynamic Data Model or actions within the Action Framework. This enables discussion and elaboration on any subject within the framework. Having these features as close to the relevant subject is a significant design decision. This prevents unnecessary navigating, be it inside or outside of the framework, to comment on any subject, which is a time-consuming and distracting task. Next to communication through comments, later stages of the framework might also add, or integrate with, other collaboration tools, such as to-do lists, or planning tools. Section 4.7 describes the exact design of the Collaborating module.

Because of the limited initial scope, and interactions between the Action Framework and Collaboration module that make it naturally coupled, we will provide a basic design and implementation for the Action Framework as part of the Collaboration module.

### 3.3 Considerations

Next to the decomposition of the framework into modules, there are a few general considerations to make.

#### 3.3.1 Monolith versus Micro-Services

Firstly, we have to consider whether we want to implement and deploy the framework as a single **monolith** or a set of **micro-services** (Fritzsche et al., 2019). A monolith is one single application, which contains the implementation of all software components. A micro-services approach, relating to a Service Oriented Architecture (SOA), decomposes the application into multiple loosely coupled individual services, which each have a single responsibility. In the case of the framework we are developing, each module in Figure 3.1 could, for example, be an individual service, even though finding the exact granularity in decomposing a framework into services can be a balancing act.

An advantage of a monolith approach is that the implementation is less complicated, since any reference between modules can just be made within the code of the implementation, and does not require communication with other services. Another advantage of the monolith approach is that initial deployment becomes faster. Only one infrastructure for the application needs to be deployed and maintained, as opposed to a set of services.

On the other hand, an advantage of the micro-services approach is that parallel development and maintenance of service becomes easier, since individual services are naturally more decoupled than modules in a monolith. Also, when the framework experiences high load and requires scaling, we can scale out only the services that experience high load, as opposed to scaling out a whole monolith.

For the initial stage of the frame we propose a monolith approach, but with only one separate service. We design and implement one single "back end" application, which includes all parts of the architecture, except for the GUI. The only external interface this back end has, is the API. Then we devise a separate GUI service, which provides a GUI to the user, and integrates with all the functionality that is present within the back end through the API. This is shown in Figure 3.2, where the back end service is encircled in red, and the front end service in blue. In this way, we get a distinction between the GUI (front end) and the back end.

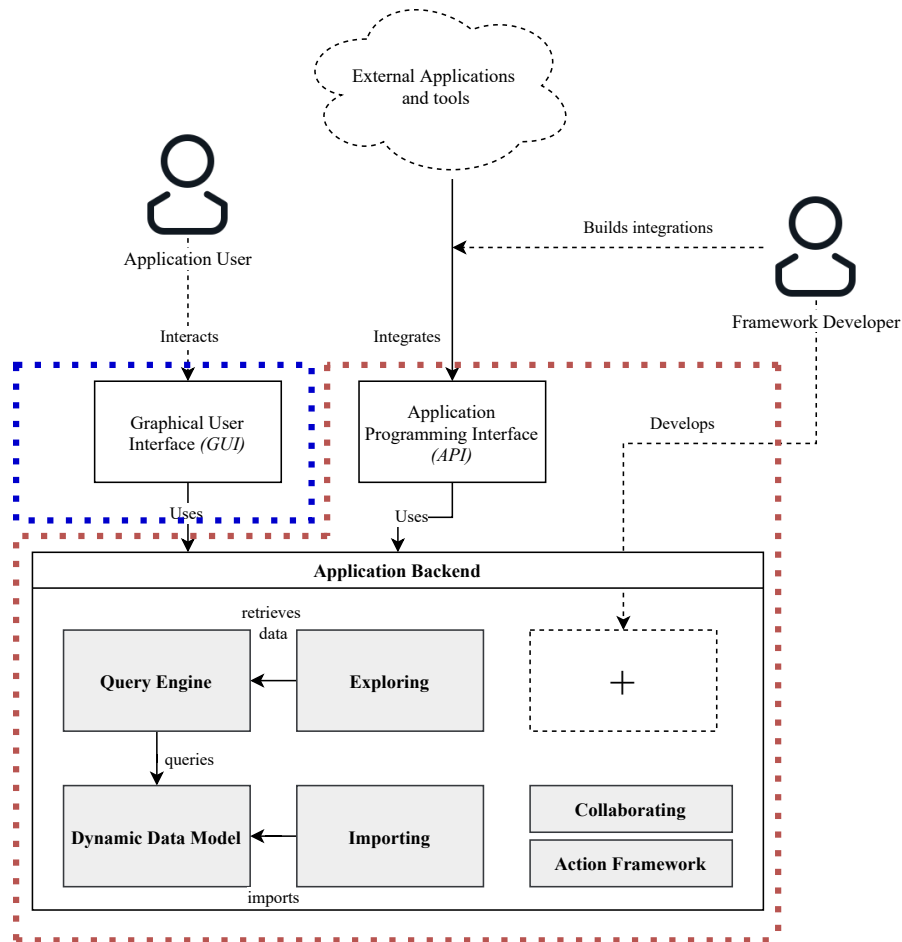


Figure 3.2: The general architecture guiding the approach for designing and implementing the framework, split up in a back end (encircled in red) and a front end (encircled in blue).

### 3.3.2 Open Source versus Closed Source

A second consideration that we have to make is whether the framework should be an open-source project, or a closed source project. The former meaning that the source code is publicly available, and can be used by anyone under a certain license while the latter indicates that only people who are specifically granted access, can access the source code. We advise making the

framework and all of its components open source under the GNU General Public License v3.0<sup>1</sup>. This will allow other developers, not directly related to the main owners of the framework to make or propose improvements and additions. This will lower the threshold for future developers to join or contribute to the development of the framework. The advantages of having a closed source framework are minimal, and an open-source approach does not have any blocking implications.

### 3.3.3 Deletion versus Archiving

The framework will contain certain functionality around deleting objects. We have to decide whether a deletion of, for example, a data object actually deletes the corresponding database record, or archives it in a way. Deleting the record from the database resembles a deletion of an object most accurately, however, if a user deletes a record by accident, it will be hard to restore the record. Another disadvantage of deleting the database record is that if any references to the object to be deleted exist - which is especially the case for data objects - these references are broken by the deletion. This causes incomplete even invalid data states. Archiving on the other hand, for example, by adding a boolean database field to each table indicating whether a record is archived or not, has the disadvantage of potentially polluting the database with numerous unused records. This will especially be the case in the setup phase of a project, where the data model is set up. This could namely involve some iterations of constructing the data model, where data types and data attributes are created and deleted. Concluding we decide as a general design guideline that every deletion is treated as an archive action. We can make some exceptions on this, for example, when deleting parts of a data model, where deletion still can happen if the framework can explicitly check whether no invalid data state would arise.

---

<sup>1</sup><https://www.gnu.org/licenses/gpl-3.0.html> (accessed: 2/7/2021)

# Chapter 4

## Design

This section describes the detailed design of the framework. We first address some design choices that are not specific to any module in the architecture, as seen in Figure 3.1. Then we describe the design of each module in the Application Backend, fulfilling the requirements in Section 2.4.1. We describe the designs of each module by explaining the relevant classes and their relations, elaborating on the database design if relevant, explaining non-trivial functionalities and flows in the design. For some modules, we also discuss interesting or relevant outcomes of the feedback sessions with the Concepts in Motion team, which have an impact on the design. A summarizing diagram showing most of the relevant classes is shown in Appendix A. Finally, we explain the GUI design in the form of wireframes for the module where this is relevant. All wireframes are presented in Appendix B, and some relevant wireframes are shown in this chapter.

Before describing the design, however, we discuss some definitions and assumptions made in the designs.

### 4.1 Definitions

**Hash** We use the notions of hash as a data type. A hash is a data structure available in a lot of programming languages, also sometimes referred to as a hash map or map. This data structure allows us to store items with identifiers in an object. It is also similar to the notion of an object in JSON, which is discussed in Section 4.4.

**Database relations** When modeling database designs, we present diagrams such as in Figure 4.3. Each entity in such a diagram is representing a database table, with columns corresponding to the attributes of the respective entities. Additionally, each (one to many) relation is represented by an extra column on the "many" side. In the example diagram in Figure 4.3, the `DataAttribute` table would have a column `data_type_id` of type integer, representing the relation.

**Polymorphic relations** In some cases we also use the notion of polymorphic relations. These are relations between classes or database tables that do not relate to a specific class or table. For example, in the database design in Figure 4.27, the table `ResourceAction` refers to a resource, which can be any table that represents objects that implement the `ActionableObject` interface, as will be described later. This is represented in the database table `ResourceAction` by not

just having a column named `resource_action_id` of type integer, but also a column named `resource_action_type` of type string, representing the type of object that a specific record in the `ResourceAction` table refers to.

## 4.2 Design Choices

In this section, we discuss general design choices not specific to one of the modules defined in the architecture.

### 4.2.1 Interface Points

We previously defined two interface points, a graphical one - the GUI - geared towards the Application Users, and an interface that can only be used programmatically, the API. Each module in the Application Backend has functionality that can and should be exposed to the users, either through the GUI or through the API, but often through both. To prevent duplicate implementation of features for the API and GUI, which would inevitably eventually also imply inconsistencies in the implementation of certain features, we propose a design where the GUI interacts with the Application Backend through the API. This is visualized in Figure 4.1.

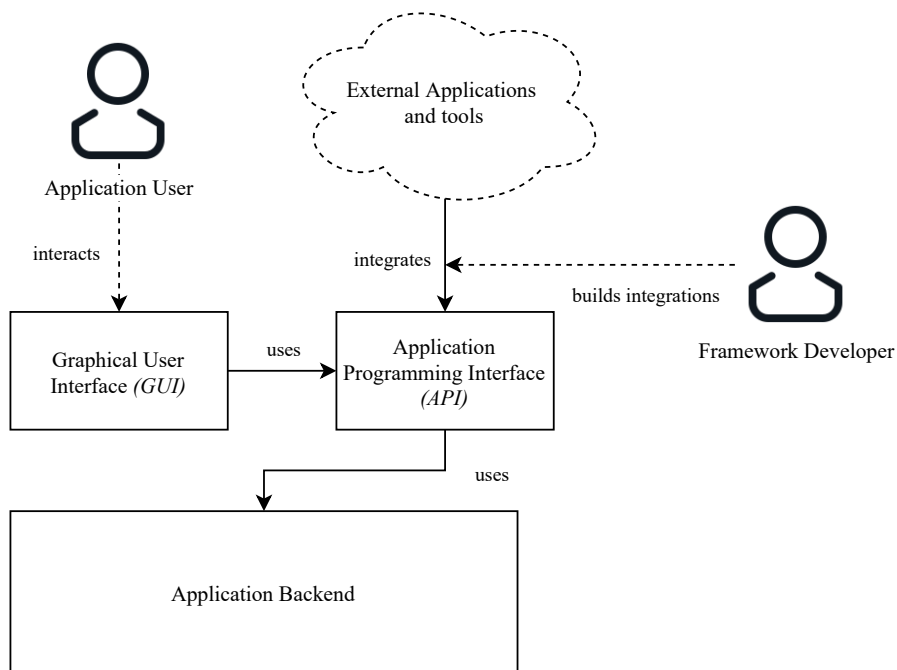


Figure 4.1: An altered general architecture, based on Figure 3.1. The GUI interacts with the Application Backend only through the API, to prevent duplicate implementation, and inconsistent implementations of the API and GUI.

We describe the designs of the GUI as wireframes, for each module in the Application Backend in the following sections. In the description of the implementation in Section 5.1, we will elaborate on how these wireframes are transformed into an actual interface.

### 4.2.2 Projects

Another design decision we make is that every entity and functionality is scoped by projects. This means that a project is a secluded environment, where no interference with other projects is possible. Any database record can always be traced back to one unique project. The only overarching objects are users since a single user can have access to multiple projects. This can be controlled very granularly once detailed Access Control is added in a future stage of the project. This way we can have a single deployment of the framework supporting multiple projects, instead of having to make separate deployments for each project that desires to use this framework, which would be a costly undertaking.

### 4.2.3 Authentication

As discussed before, access control will not be part of this thesis project, however, we do devise an authentication system. Users will be registered and stored in a database table. The API will then have one endpoint for authenticating, i.e. logging in, and one for logging out. All other endpoints then require the user to be authenticated. The GUI checks whether a user is authenticated and shows a login screen that communicates with the API to log the user in. This is done when the user opens the application, but also when a request fails to authenticate, so the user can log in again, to re-authenticate. See, for example, Figure C.1 (in Appendix C) for this. When access control is implemented, a user can, for example, be granted access to specific projects, while in the initial stage of the framework, a user will simply have access to all projects. This is sufficient since the initial stage of the framework will only be used in a controlled environment.

## 4.3 Dynamic Data Model

The Dynamic Data Model provides the functionality to create a data model completely tailored to the specific project of a user. An example data model that is used in practice by historians of ideas is shown before in Figure 1.3.

### 4.3.1 Class Design

We devise the design shown in the class diagram in Figure 4.2.

On the left side of this figure we see the components that enable a model definition: the `DataType`, the `DataAttribute` and the `DataValidator`. One `DataType` can have multiple `DataAttributes`, and each `DataAttribute` can have multiple `DataValidators`. On the right side we see the components that enable the storage of actual content according to the `DataModel`. A `DataObject` represents a single object that can contain data in the form of `AttributeValues`. An `AttributeValue` belongs to a `DataAttribute`. Note that this design implements the idea explained in Figure 2.1.

**DataType** A `DataType` has a name, and a `representing_name_template`. This is a template string that ordains how a `DataObject` of this `DataType` should be represented in string form. This can be useful when the GUI refers to an object in an interface. An example `representing_name_template` for a fictional `DataType` named `Person` with `DataAttributes` named `FirstName`, `LastName` and `YearOfBirth` could be: `"LastName, FirstName (YearOfBirth)"`. Then if a `DataObject` of the `DataType` `Person` with values `FirstName: Anton`, `LastName: Bommel`,

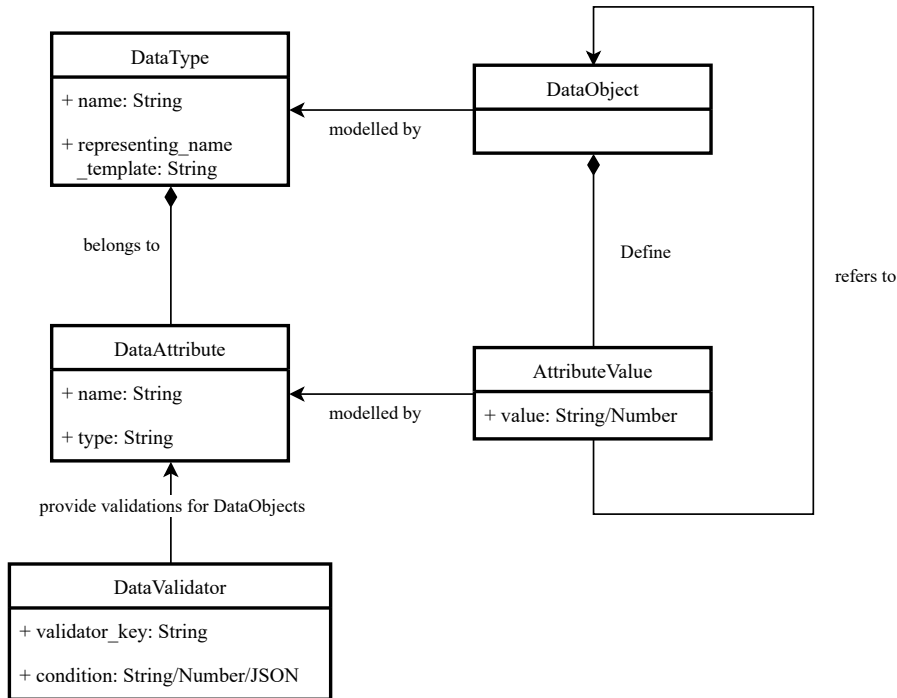


Figure 4.2: The class diagram for the proposed design for the Dynamic Data Model

**YearOfBirth:** 1942 needs to be represented, the resulting representing string would be "Bommel, Anton (1942)".

**DataAttribute** A **DataAttribute** has a name and type. The name defines the name of the **DataAttribute**, and the type defines the type of data that the **DataAttribute** models. The type can be one of the *primitive* types: **Text**, **Number**, **DateTime**. If a **DataObject** wants to specify a value for this **DataAttribute**, it has to be of string type if the **DataAttribute** type is **Text**, of integer or float type if the **DataAttribute** type is **Number**, and of string type representing a **DateTime** as specified in RFC 2822 (Resnick, 2001).

Next to one of these primitive types, the type can also be the *complex* type **Reference**. This type expects a **DataObject** to have an **AttributeValue** with an integer value that is an ID of a certain other **DataObject**. This effectively allows to create relations between **DataObjects**.

The other complex type is **Array**. This type allows to store an array of any primitive or **Reference** typed **DataAttributes** as a single field. This can, for example, be useful if a **DataObject** needs to refer to a list of other **DataObjects**, or if it needs to store an array of texts. The actual value for the type attribute of a **DataAttribute** can, for example, be "Array[Text]" for a **DataAttribute** that represents an Array of Texts or "Array[Reference]" for a **DataAttribute** that represents a list of references to other **DataObjects**.

**DataValidator** A **DataValidator** belongs to a **DataAttribute**. It stores definitions to validate **AttributeValues** of a **DataObject** when a **DataObject** with a value for this **DataAttribute** is created or updated. The **DataValidator** is defined by a **validator\_key**, indicating which type of validation is to be performed, such as length and presence. Additionally the **DataValidator**



has a condition, that describes details for a specific validator. Table 4.1 shows all possible validators and the `DataAttribute` types for which they are valid, and the expected format of the condition.

validator_key	Allowed DataAttribute types	Expected Condition Format	Description
presence	Text, Number, DateTime, Reference, Array	True or false	Validates whether a <code>DataObject</code> has a non-empty value for this <code>DataAttribute</code>
length	Text, Array	A JSON object containing <code>atLeast</code> , <code>atMost</code> or <code>exactly</code> keys, with corresponding values	Validates whether a <code>DataObject</code> has a value that has at least, at most or exactly a certain length
inclusion	Text, Number, DateTime	A set of values of valid type	Validates whether a <code>DataObject</code> has a value that is included in a set of values. If this validation is present, the interface can show the edit field for this <code>DataAttribute</code> as a drop-down box.
referred_data_type	Reference	An ID of a <code>DataType</code>	Validates whether a <code>DataObject</code> has a value that is an ID of another <code>DataObject</code> which is of a <code>DataType</code> with a certain ID (the condition).

Table 4.1: All allowed `validator_keys` with matching `DataAttribute` types, and a description of the expected format of the condition attribute of the `DataValidator`.

Next we show a few examples of `DataValidators`. Firstly, the following validator validates whether a value is given as `AttributeValue` of a `DataObject`:

```
validator_key: presence,
condition: true
```

The validator with the following attributes validates whether the length of the string provided as `AttributeValue` is between 4 and 10 characters long:

```
validator_key: length,
condition: { atLeast: 4, atMost: 10 }
```

Another validator validates that a `DataObject` has a value for the `DataAttribute` that is one of Amsterdam or Eindhoven:

```

validator_key: inclusion,
condition: ["Amsterdam", "Eindhoven"]

```

Finally, the next validator validates that (assuming we have a `DataType` called `Bike`, with id 5192) a `DataObject` has a value for this `DataAttribute` that corresponds to a `DataObject` that is a `Bike`, i.e. belongs to the `DataType` named `Bike`.

```

validator_key: referenced_data_type,
condition: 5192

```

**DataObject** The `DataObject` is an object that corresponds to a certain `DataType`. `DataObjects` can be seen as records that follow the (dynamic) schema of the defined data model. A `DataObject` has a number of `AttributeValues` that each store a value for a `DataAttribute` of the corresponding `DataType`. A `DataObject` has at most one `AttributeValue` for each `DataAttribute` of the `DataType` of the `DataObject`.

**AttributeValue** Already being explained before, the `AttributeValue` represents the value that a `DataObject` has for a specific `DataAttribute` of the `DataType` the `DataObject` belongs to. Its value attribute can be validated by a `DataValidator`.

### 4.3.2 Database Design

The database design for the Dynamic Data Model is visualized in Figure 4.3 and differs slightly from the class diagram in Figure 4.2.

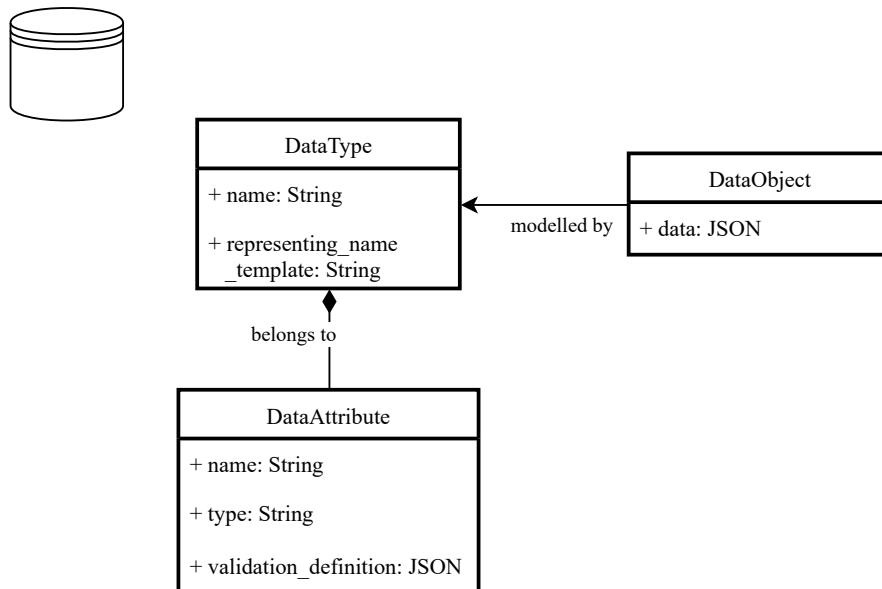


Figure 4.3: The database design for the Dynamic Data Model

We design the framework with scalability in mind. However, if we would make separate tables for each class presented in the class diagram, querying a list of data objects from the database would involve complex queries for retrieving relatively simple data. However, as we noticed in Section 1.3, modern database technology allows us to efficiently store JSON data. Therefore the data of a `DataObject` is stored in a JSON attribute called `data` in the `DataObject` table. Similarly, validators are stored in the `validation_definition` JSON attribute of the `DataAttribute` table. Then in the implementation, we unpack these data and definitions stored as JSON, into the correct instances of corresponding classes from the class diagram.

### 4.3.3 Interaction

Now that the framework has a way to define a Dynamic Data Model, and store data in it, we need to design a way to interact with it. Taking the description from Chapter 2 into account, we design the following list of interactions:

1. Create, read, update and delete (CRUD) functionalities for `DataType` and `DataAttribute`.
2. CRUD functionalities for `DataObject`
3. Bulk create for `DataObject`
4. Filtering and querying `DataObjects`

The CRUD interactions for `DataType`, `DataAttribute` are trivial actions that can be implemented in the API and do not need to be specified further, except for the fact that deletion of a `DataType` or corresponding `DataAttributes` is only possible when no `DataObjects` of the `DataType` exist, as explained in Section 3.3.3.

CRUD interactions for `DataObject` are not complex either. When creating a `DataObject` the user submits a `DataType` ID combined with some data that matches with the `DataAttributes` present within the `DataType`. Then all the data is validated by the `DataValidators` of the `DataAttributes`. If the validations succeed, the object is created, if not, a validation error is returned to the user.

Bulk creation of `DataObjects` will be handled by the Importing module, Section 4.6.

Filtering and querying `DataObjects` will be handled by the Query Engine, described in Section 4.4.

### 4.3.4 UI Design

We do not present a UI design for creating and updating a data model yet, as this will be part of a future stage of the project. This section, however, will show the wireframes for creating, reading, updating, and deleting (which is archiving) `DataObjects`.

Figure 4.4 shows the wireframe of the UI for creating a `DataObject`. It is actually a rather simple form, with form fields for each `DataAttribute` of the relevant `DataType`. The icons indicate different types of attributes. The wireframe shows a number of primitive fields: Text fields (Title, Publisher), a Date field (Year of publication) and a number field (Volume number). Next to that, there is a complex field, namely a Reference field (Related Work). Instead of actually entering a value here by typing, which is possible for all the primitive fields, the user would click the form field, and a popup would open, shown as a wireframe in Figure 4.5. This popup allows the user to select a `DataObject` to refer to, or create a `DataObject` inline. Once a referenced `DataObject` is selected, the representing string - generated using

The 'Create Work' form contains the following fields and controls:

- Title:** Text input with 'Logica Nuclearis' and a red 'A' icon.
- City of publication:** Dropdown menu with 'Eindhoven' and a downward arrow.
- Year of publication:** Text input with '1872' and a calendar icon.
- Volume number:** Text input with '15' and a '#' icon.
- Publisher:** Text input with 'Heidelberg' and a red 'A' icon.
- Related Work:** Text input with 'Dark Matter' and a link icon.
- Save:** A dark blue button at the bottom center.

Figure 4.4: The wireframe of the UI for creating a data object, in the example the created data object would be of the data type named *Work*.

The 'Select a Work' popup contains a table with the following data:

	ID	Title	Author	Year of publication
<input type="radio"/>	12	Philosophia Rationalis	C. Wolff	1728
<input type="radio"/>	15	De nucleare logica	H. Olland	1852
<input type="radio"/>	16	Lord of the rings	N. Arnio	1852
<input type="radio"/>	17	Dark matter	G. Ramsey	1852
<input type="radio"/>	18	Imitatione Christi	T. a Kempis	1852

At the bottom right, there are two buttons: 'Create new' (white) and 'Select' (dark blue).

Figure 4.5: The wireframe of the popup UI for selecting a referenced data object for a reference form field, while creating or updating a data object

the `representing_name_template` attribute of the referenced `DataType` - of this `DataObject` is shown in the form field to indicate the selection.

The UI for updating a `DataObject` is actually very similar to the UI for creating one and is represented by the wireframe in Figure B.3. The only significant difference with the create UI is that in this wireframe a button to archive the `DataObject` is present. Clicking it would, after confirmation of the user, archive the `DataObject`.

Finally, we show the UI for reading a `DataObject` as a wireframe in Figure 4.6. This interface shows all `AttributeValues` first, where `AttributeValues` for `DataAttributes` of type reference actually provided a link to the read interface of that object. In this example, the user can click on the author representing string, and that would take the user to the UI to read that specific `DataObject`.

Next to this trivial display of the `AttributeValues`, we also introduce an interface that allows the user to explore all `DataObjects` that refer to the shown `DataObject`. For example,

[Show Work](#)
[edit work](#)

Title: Philosophia Rationalis sive Logica

Author: [Wolff, Christian, Freiherr von](#)

**Manifestations:**

ID	Title	Year	Language	Place
587	Philosophia Rationalis sive Logica	1728	Latin	Frankfurt, Germany
573	Philosophia Rationalis sive Logica	1732	Latin	Frankfurt, Germany
2093	Philosophia Rationalis sive Logica	1735	Latin	Verona, Italy
572	Philosophia Rationalis sive Logica	1740	Latin	Frankfurt, Germany
877	Philosophia Rationalis sive Logica	1746	Latin	Helmstadt, Germany

▼ Items that refer to these Manifestations as "Manifestations"

ID	Title	Year	Library	Place
152	Philosophia Rationalis sive Logica	1740	Heidelberg University Library	Heidelberg, Germany
153	Philosophia Rationalis sive Logica	1776	Bavarian State Library	München, Germany
164	Philosophia Rationalis sive Logica	1801	Flevomeer Library Lelystad	Lelystad, Nederland
182	Philosophia Rationalis sive Logica	1788	Public Library of Valencia	Valencia, Spain
201	Philosophia Rationalis sive Logica	1815	Trinity College Library	Dublin, Ireland

▼ Digitized copies that refer to these Items as "item"

ID	Title	Quality	Source	Link
256	Philosophia Rationalis sive Logica	1	Google Books	<a href="https://books.google.com/books/books?id=749">https://books.google.com/books/books?id=749</a>
483	Philosophia Rationalis sive Logica	3	InternetArchive	<a href="https://www.internetarchive.org/philosophiarati00">https://www.internetarchive.org/philosophiarati00</a>
187	Philosophia Rationalis sive Logica	5	MDZ	<a href="https://reader.digitale-sammlungen.de/10008294">https://reader.digitale-sammlungen.de/10008294</a>
664	Philosophia Rationalis sive Logica	4	Google Books	<a href="https://books.google.com/books/wolffii-phil">https://books.google.com/books/wolffii-phil</a>
778	Philosophia Rationalis sive Logica	1	SLUB	<a href="https://digital.slub-dresden.de/id/0-774211326">https://digital.slub-dresden.de/id/0-774211326</a>

► Documentaries that refer to these Manifestations as "Covered Manifestation"

► Documentaries that refer to these Manifestations as "Source Manifestations"

Figure 4.6: The wireframe of the UI for reading a data object. The attribute values are shown on the top, and below is a part that shows all data objects that refer to the shown data object.

in this wireframe we assume that the `DataType` named `Manifestation` has a `DataAttribute` that refers to the `DataType` named `Work`. This means that for each work, there can be a set of manifestations that refer to that work through that `DataAttribute`. This set of manifestations is shown on the read UI of the work. This will be added for any `DataAttribute` of any `DataType` that refers to this work. Additionally, nested referencing `DataObjects` can also be explored. In the design, we can also see that all items - `DataObjects` of the `DataType` named `Item` - that refer to any of the manifestations that refer to the work are shown as well. This nesting can in theory be expanded indefinitely if the data model has circular referencing. However, by default, none of the lists of referencing objects will be visible, as the user has to explicitly expand the lists, to reduce the performance impact in loading the interface. Also, infinite loops in loading the references are prevented this way.

The lists of referring objects will be generated using a query that is sent to the Query Engine, as will be explained in the following sections. This again prevents duplicate implementations of querying. This also, allows the user to swiftly add further explore a specific list of referencing objects, since the list of referencing objects can be reconstructed using a query in the Query Engine, and that query can then be expanded by adding or removing clauses of the query.

### 4.3.5 Feedback Sessions

One of the most important conclusions from the feedback session concerning the Dynamic Data Model is that once the data model is constructed for a project, it should not be possible to adapt it anymore. If a data model would change once data is created in it, this data could become unusable if the data model no longer correctly models this data. Additionally, having this constraint encourages the user to clearly define their data model upfront. However, when the user has a valid reason to change the data model, they can use the importing and exporting functionality of the framework to export all the data from the original data model to the new, improved data model, in a new project. Doing this explicitly also triggers the user to check the migration from the original to the new data model, which relieves the need for a generalized approach for changing the data model during a project, and allows for an accurate migration for each specific case.

## 4.4 Query Engine

Now that we have a way to store data through the Dynamic Data Model, a way to query that data is required to make effective use of the data. Since the data model is dynamic, it is not possible to make preprogrammed queries into the application, except for the simple indexing queries such as "Show all `DataObjects` of `DataType` X". Therefore, we design the Query Engine. In general it takes in a query in a query DSL and a `DataType` to search in, parses that query, finds matching `DataObjects` of that `DataType` and returns them to the user.

In this section we first describe the process that the Query Engine conceptually follows, next we describe the semantic structure of the query DSL, followed by a description of the design of the Query Engine itself. The Query Engine module itself does not have any GUI but is used by the Exploring module, which allows users to enter or construct queries, and perform other relevant actions, which are described in the next section, Section 4.5.

### 4.4.1 Process

The process of the Query Engine is illustrated in Figure 4.7.

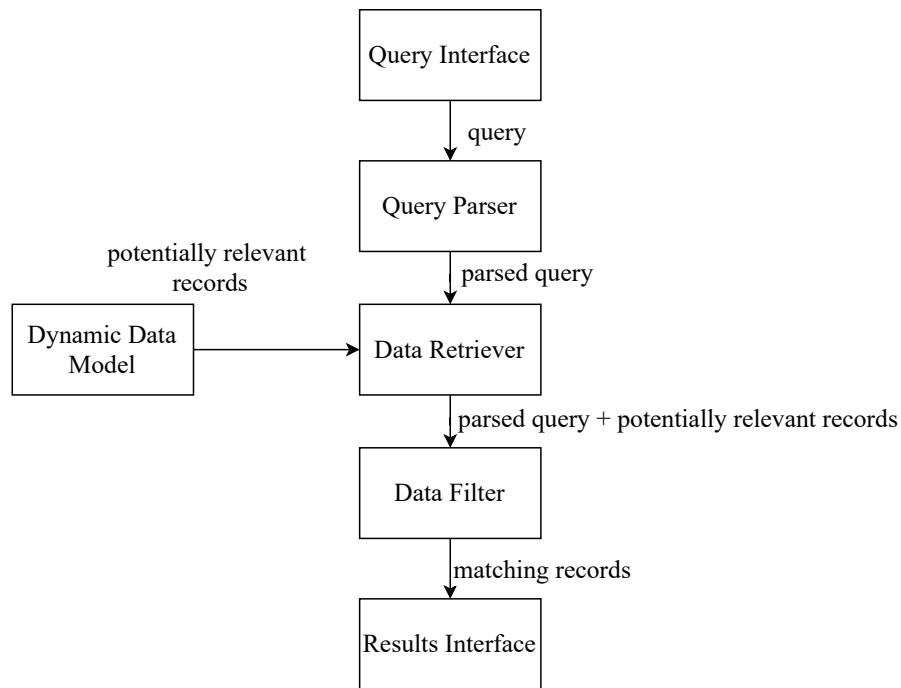


Figure 4.7: The conceptual process of the Query Engine.

From this figure we can distinguish the following steps in the process of handling a query:

1. **Query Interface:** The Query Interface takes in a query string. As mentioned, the Query Engine itself does not have a GUI, but can only be accessed through the API.
2. **Query Parser:** The Query Parser component takes in the query string, and parses it into an abstract tree, in order to be able to process it.
3. **Data Retriever:** The Data Retriever inspects the parsed query and makes one or more database queries to retrieve all data objects from the Dynamic Data Model that are potentially relevant to the query.
4. **Data Filter:** The Data Filter filters through all retrieved data objects and only keeps records that exactly match the query. A data object is matching a query if the root node of the query returns true for that data object.
5. **Results Interface:** The Results Interface returns all the matching data objects.

#### 4.4.2 Query DSL Semantics

Next, we describe the theoretical structure of the query DSL. The query DSL, which allows the user to construct queries in textual format, is based on JSON. In a future stage of the project, the DSL could also support YAML as basis language, since JSON and YAML are similar in structure. In fact, YAML is a superset of JSON, (Ben-Kiki, Evans and Döt Net, 2009) and takes significant inspiration from the Elasticsearch Query DSL as described in Section 1.3.

### JSON Terminology:

For the reader who is unfamiliar with JSON, we advise to establish an understanding of basic JSON terminology<sup>1</sup>. The most important notions are summarized as follows. An *object* contains a number of *key-value* pairs. The key is a string or a number, while the value is a string, number, object, or array. An *array* is an array of numbers, strings, or objects. The root of a JSON string is always an object or an array.

When we refer to a node, this refers to a key-value pair, and we also refer to the value of this pair as the children of this node, especially when the value of the node is an object or array.

Recall Figure 1.4 for an example of JSON.

### Query DSL Semantic Structure

Figure 4.8 shows the structure of the query DSL, note that the Condition node is shown multiple times, they all refer to the same node, but are drawn multiple times to prevent an unreadable figure. At the root, a query consists of one or more conditions. A condition is either a Control Flow node, or a Matcher node. Each Control Flow node has one or more child condition nodes, while a Matcher has one or more key-value pairs as children, as is explained next in more detail.

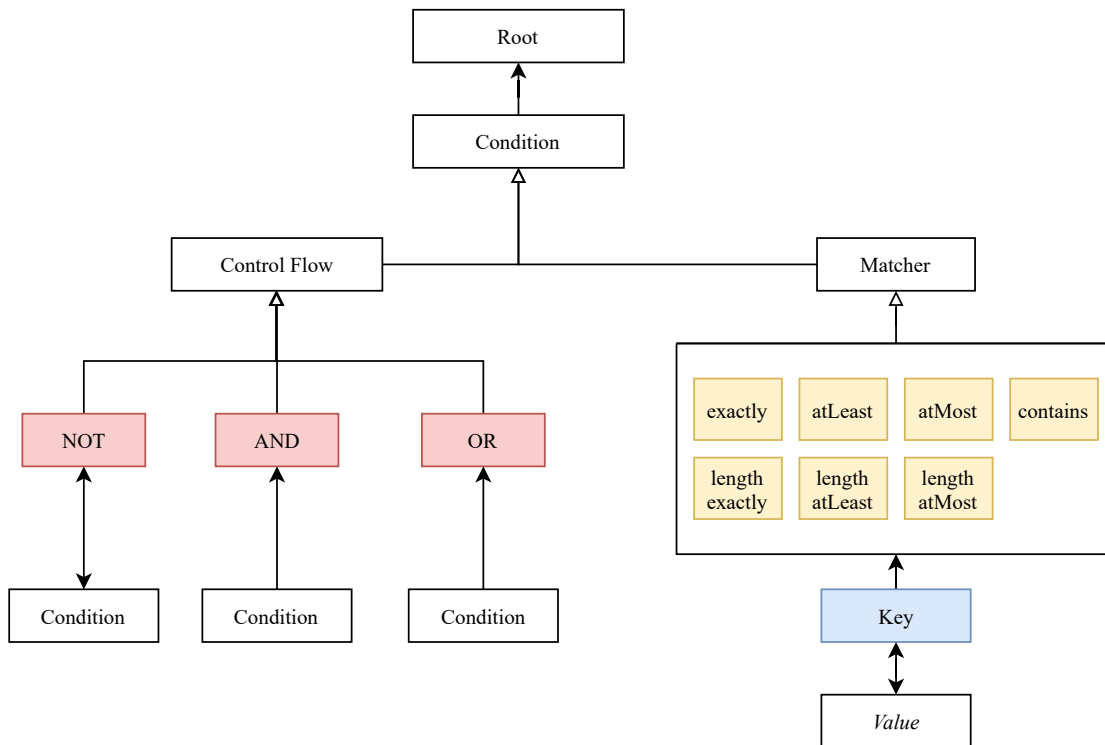


Figure 4.8: A model describing the structure of the query DSL used by the Query Engine. The color of the nodes correspond with the colors in the example queries below.

A Control Flow node can be used to combine multiple conditions. We define three Control Flow nodes:

<sup>1</sup>The pages at <https://www.json.org/json-en.html> and [https://www.w3schools.com/js/js\\_json\\_syntax.asp](https://www.w3schools.com/js/js_json_syntax.asp) provide a comprehensive introduction into JSON (visited on 2/7/2021).



- **Not:** This will return true if and only if its child `Condition` node returns false. Can have one child `Condition` node.
- **And:** This will return true if all child `Condition` nodes return true. Can have one or more child `Condition` nodes.
- **Or:** This will return true if any of the child `Condition` nodes returns true. Can have one or more child `Condition` node.

A `Matcher` node on the other hand can be used to check an attribute of the `DataObject` against some test value, and returns true if the test succeeds. A `Matcher` object has one or more key-value pairs, which we also refer to as a `MatcherEntry`. The key of a `MatcherEntry` can be either a string that corresponds with the name of one of the `DataAttributes` of the queried `DataType`, or a string representing a nested attribute, separated by a dot character. A nested attribute is the attribute of a data object that is referenced through some other attribute. For example, if the queried `DataType` has a `Reference DataAttribute` called `x` that refers to a `DataType` which has a `Text DataAttribute` named `y`, then a `MatcherEntry` key could be `"x.y"`.

We define the following `Matcher` nodes, a list which in later stages of the framework can be extended:

- **exactly:** The value of the `DataObject` for the `DataAttribute` is exactly equal to the test value.
- **atLeast/atMost:** The value of the `DataObject` for the `DataAttribute` is respectively at least or at most the test value. Only applicable to `Number` and `Date` attributes.
- **contains:** The value of the `DataObject` for the `DataAttribute` contains the test value. Only applicable to `Text` and `Array` attributes.
- **length exactly/atLeast/atMost:** The length of the value of the `DataObject` for the `DataAttribute` is respectively exactly, at least or at most the test value. Only applicable to `Text` and `Array` attributes.

Using these objects we can, for example, construct the following query (where the node keys are colored referring to the nodes in Figure 4.8):

```
{
  "exactly": {
    "title": "Dutch milestones"
  }
}
```

This query would return all `DataObjects` (of the `DataType` we are querying for) where the title is exactly `Dutch milestones`.

A query that uses a nested `DataAttribute` key for its `MatcherEntry` looks as follows:

```
{
  "exactly": {
    "author.firstName": "Wolfgang"
  }
}
```

This query would return all data objects which have a reference to a data object for the `DataAttribute` named `author` which `firstName` is exactly "Wolfgang".

A more complex query could, for example, be:

```
{
  "and": {
    "exactly": {
      "title": "Dutch milestones"
    },
    "atLeast": {
      "yearOfPublication": 1572
    }
  }
}
```

This query would return all `DataObject` where the title is exactly Dutch milestones and the `yearOfPublication` attribute is at least 1572.

Due to the recursive nature of the definition of the DSL - `Control Flow` objects can have `Control Flow` nodes as child nodes - any combination of matchers can be constructed, such as the following complex query, which would select all `DataObjects` that do not have a title that is exactly Dutch milestones, and either the `yearOfPublication` attribute is at least 1572, or the `subtitle` attribute contains the word science:

```
{
  "and": {
    "not": {
      "exactly": {
        "title": "Dutch milestones"
      },
    },
    "or": {
      "atLeast": {
        "yearOfPublication": 1572
      },
      "contains": {
        "subtitle": "science"
      }
    }
  }
}
```

### Query DSL Considerations

To ensure the robustness of the DSL, there are three considerations that have to be made.

Firstly, how to combine multiple children of `Control Flow` nodes? JSON provides two ways for this. If the keys of all children are unique - e.g. Not and Or - they can just be put in a single object, as can be seen in the above example queries. If we want to construct a set of children that have the same key - e.g. Or and Or - then we can not use a single object to represent this. This would introduce duplicate keys in a JSON object, which is not considered correct syntax by many JSON parsers, and discouraged by RFC 8259.4 (Bray, 2017). Therefore, in such a case,

an array of objects is the correct solution. This enables the children of Control Flow nodes to contain multiple children with the same name, as in the following query where we have multiple exactly child nodes of the and node:

```
{
  "and": [{
    "exactly": {
      "title": "Dutch milestones"
    }
  },
  {
    "exactly": {
      "title": 1572
    }
  }
]
```

Since the child objects have the same key - exactly - they can not be combined into one child object since this would result in one object having two keys both named exactly, hence the and node has an array of objects as a child.

The second thing we have to consider is that the query root can either be an object, or an array of objects, and in both cases, multiple root nodes can exist - e.g. an Or and Exactly node. In such a case, the Query Parser will automatically add a new root, in the form of the Control Flow node And, and the original root nodes (either an object or an array of objects) will be considered children of this new root. In practice, this means that the following query:

```
{
  "not": {
    "exactly": {
      "title": "Dutch milestones"
    }
  },
  "exactly": {
    "title": 1572
  }
}
```

is translated by the parser into the following, effectively equivalent query:

```
{
  "and": {
    "not": {
      "exactly": {
        "title": "Dutch milestones"
      }
    },
    "exactly": {
      "title": 1572
    }
  }
}
```

Thirdly, we need to consider the combination of children of *Matcher* nodes. Theoretically, the DSL allows multiple *Matcher* children. If such a case is encountered by the Query Parser, such a query will be considered to be joined by an *And* node. This means that the following query:

```
{
  "exactly": {
    "title": "Dutch milestones",
    "title": 1972
  }
}
```

is equivalent to:

```
{
  "and": [{
    "exactly": {
      "title": "Dutch milestones"
    }
  },
  {
    "exactly": {
      "title": 1572
    }
  }
}]
}
```

These queries are equivalent and both considered to be valid queries.

### 4.4.3 Class Design

Now that we established the semantics of the DSL, we describe the classes that make up the Query Engine. We distinguish between two groups of classes. Firstly we describe a set of controller classes that manage the querying process, and secondly, we describe a set of classes that will represent any query and implement the functionality to filter data objects using that query. Finally, we describe in detail how the querying process is designed, touching upon the interaction between these controller classes and query-representing classes.

#### Controller classes

The controller classes are shown in Figure 4.9.

The *QueryExecutor* is the main controller, which uses the other controller classes, the *QueryParser*, *QueryDataRetriever* and *QueryDataFilter*, to complete the query process as described in Figure 4.7.

#### Query-representing Classes

The query-representing classes are shown in Figure 4.10.

A query is represented by a tree of *Nodes*. We distinguish *FilterNode* and *MatcherEntry* nodes, which both inherit from the *Node* class.

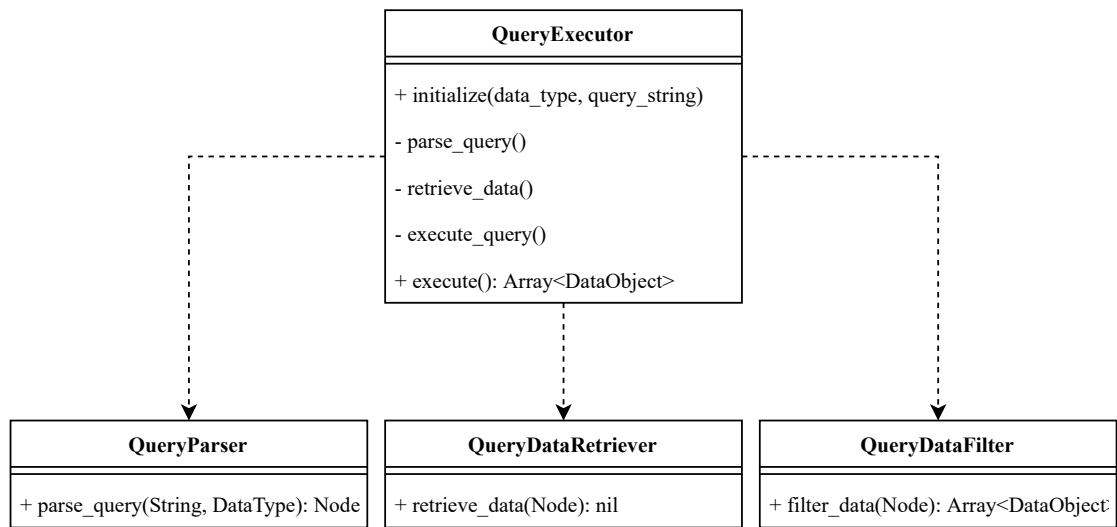


Figure 4.9: The class diagram describing the controller classes of the Query Engine.

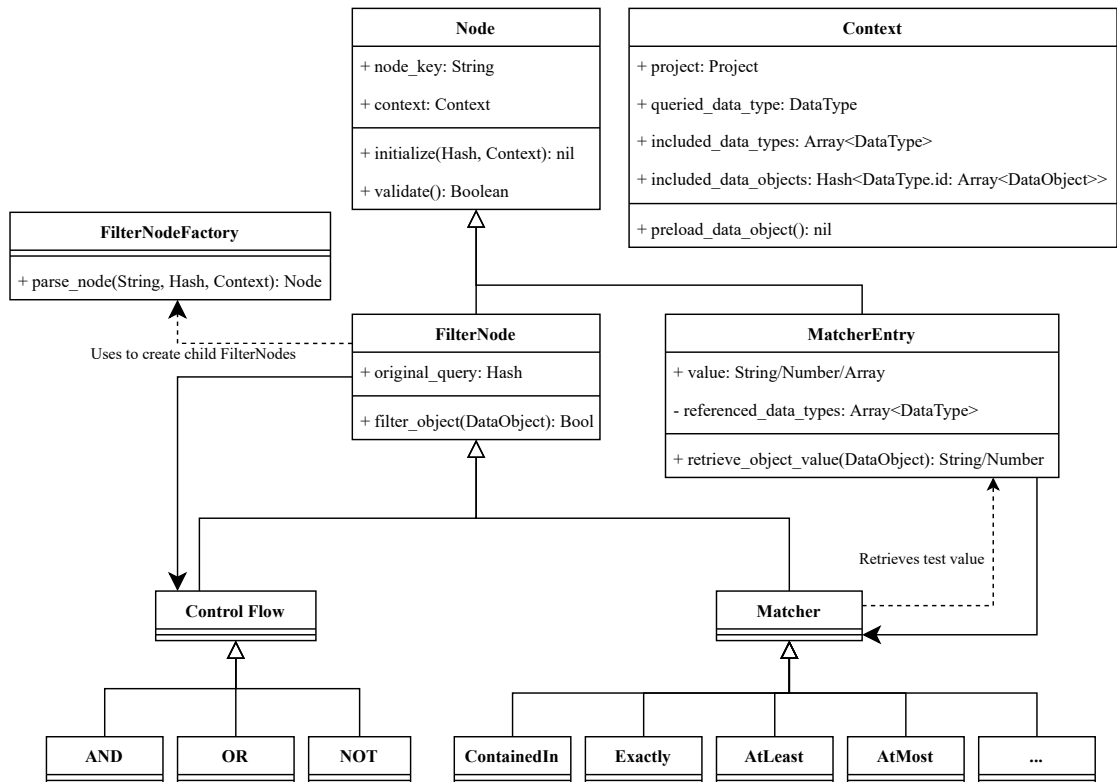


Figure 4.10: The class diagram describing the classes that contain the functionality to represent a query and filter data objects.

The `Control Flow` and `Matcher` class inherit from the `FilterNode`. The `Control Flow` and `Matcher` class directly correspond to the `Control Flow` and `Matcher` nodes respectively as described in Section 4.4.2. The recursive nature of queries is represented by the relation that the `Control Flow` class has to the `FilterNode` class, indicating that a `Control Flow` node can have zero or more `FilterNodes` (either a `Control Flow` or `Matcher` node) as children. The diagram also shows a non-exhaustive list of inheriting classes of `Control Flow` and `Matcher` classes, such as `And`, `Or`, `Not`, `Contains`, `Exactly` and `GreaterThan`. The initial stage of the framework only comprises a number of them, however, due to the flexible setup of this query-representing classes, it does not take a lot effort to add additional `Matcher` or `Control Flow` classes.

`FilterNodes` are initialized with a context object and the sub query represented by a hash. `Control Flow` nodes will call the `FilterNodeFactory` to parse this sub query into the children `FilterNodes` as part of their initialization procedure. `Matcher` nodes on the other hand can not have `FilterNodes` as children, but only `MatcherEntries`, so as part of their initialization procedure the sub query - which is expected to represent only `MatcherEntries` - will be parsed into `MatcherEntry` children.

The `MatcherEntry` class is used to represent key-value pairs that make up the children of a `Matcher` node in a query. It is used to store the test value, stored as the attribute value. The key is represented by the inherited attribute `node_key`, and either corresponds to the name of a `DataAttribute` of the queried `DataType`, or a nested `DataAttribute`. When a `MatcherEntry` object is initialized with a `node_key` that refers to a nested attribute, it will also store the `DataTypes` that are referenced through the nested `DataAttributes`, and also add these to the context, in the attribute `included_data_types`. This is then later used to also retrieve all data objects that possibly are referred to by nested attributes.

The `Context` class is a singleton class that is shared across all nodes and is used to store the context of a query, including which project the query concerns, which `DataType` is being queried, which `DataTypes` are further relevant to the query because of nested attribute matchers, and an attribute `included_data_objects` which can store data objects that are loaded from the database and can be used to perform the query.

The `FilterNodeFactory` is a class that can generate a `FilterNode` from a node key, indicating the type of the node, a subquery representing all children of the node to be created - as a hash - and a context object.

### Querying Process Design

The querying process is designed as follows (note that this closely resembles the process in Figure 4.7):

1. A `QueryExecutor` is initialized, with the `DataType` to be queried and a query string.
2. The `QueryExecutor` receives a call to its `execute` method, and it will call the three methods `parse_query`, `retrieve_data`, `filter_data` consecutively, as described in the following steps.
3. `parse_query` is calling the homonymous `QueryParser` method, executing the following steps:
  - (a) Parse the query string into the equivalent hash.
  - (b) Optionally nest the root in an `And` node, so the parsed query always has one single root node.

- (c) Parse the hash into a query node using the `FilterNodeFactory`, which recursively parses the whole query into nodes.
  - (d) The resulting parsed query is validated, by calling the `validate` method on the root node. The validation checks whether the `node_key` matches the class of the node for `FilterNodes`. For `MatcherEntries` the validation entails checking whether the validated key actually corresponds to an existing `DataAttribute`, and whether this `DataAttribute` is of a type that matches the `Matcher` type. Recall that specific matchers only work for specific `DataAttribute` types, e.g. the `AtLeast` matcher only works with `Number` or `DateTime` `DataAttributes`. For `MatcherEntries`, the validation also checks whether the test value actually matches the `Matcher` type, e.g. for the `AtLeast` matcher on a `DataAttribute` of type `DateTime`, we only expect a test value of the type `DateTime`.
4. `retrieve_data` is calling the homonymous `QueryDataRetriever` method, providing the root node as an argument, which loads all data objects of the queried `DataTypes` from the database - as potentially matching data objects - which are then stored into the context in the `included_data_objects` attribute. Also, all data objects of any of the `included_data_types` attribute are also loaded from the database. This will load all data objects of data types that are referred to through nested attributes.

Note that this is a naive approach that can be optimized significantly in later stages of the framework by, for example, by extracting properties from the parsed query and translating these into conditions in the database query which can significantly reduce the number of data objects which are loaded from the database, increasing the performance of the following steps.

5. `filter_data` calls the homonymous method from `QueryDataFilter`, providing the root node as an argument. It loops over all data objects in the context that potentially match with the query, and calls the `filter_object` method on the root node with each of these data objects. This results in an array containing all data objects that yield true for the `filter_object` method of the root node of the query.

The array of matching data objects that results from this querying procedure can then be presented to the API, and be used by, for example, the Exploring module.

## 4.5 Exploring

The Exploring module is designed to make the exploration features that the Query Engine provides available to the application users. This module adds a UI to construct and execute queries in two ways. Firstly, the user can simply enter a query using the query DSL as text, and submit the query to retrieve the results. Secondly, an interface will be available to the user where they can construct a query using visual components. Next to the querying interface, the Exploring module also adds functionality to store queries and create and maintain collections of data objects. Collections provide the user with a way to store sets of data objects for later use, or for exporting sets of data objects to use in external tools. The following sections describe the class design and the corresponding database design, followed by a description of the UI design.

### 4.5.1 Technical Design

Figure 4.11 shows the design of the Exploring module. We address the responsibilities of these classes next.

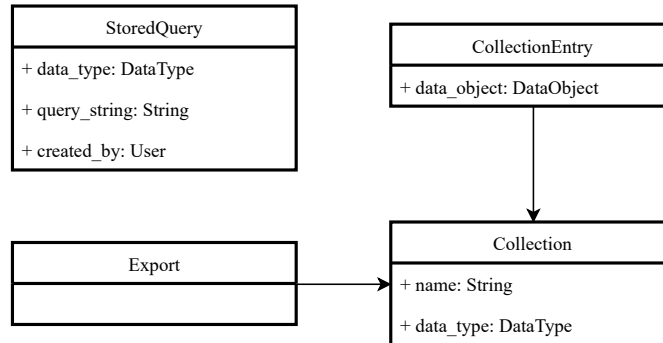


Figure 4.11: The class diagram describing the relevant classes of the Exploring module.

#### StoredQuery

The `StoredQuery` class - its name is purposely not just `Query`, to avoid confusion with all the other uses of query in this thesis - represents a query that has been stored. This query can be revisited by the user but does not explicitly store the previous results of executing this query. The user can however load this query and execute it at any time, giving the user a result that is up to date for the current state of the database.

#### Collection and CollectionEntry

A `Collection`, having a name to be able to find it later, can be used to represents a collection of data objects of one specific data type. It has a relation to one or more `CollectionEntry` objects, which refer to a data object.

#### Export

An `Export` can take a collection and export it into an appropriate type, which will be a CSV file for the initial stage of the framework.

### 4.5.2 Database Design

The database design as shown in Figure 4.12, does not include storage of exports, as there is no requirement for this. Future stages of the framework might include some stored reference to exports, either in the form of explicit database records, but also in different ways like an `Export Action` within the Action Framework that indicates to the user that a `Collection` was exported at certain points in time.

### 4.5.3 UI Design

In this section we describe interactions for the following functionalities:



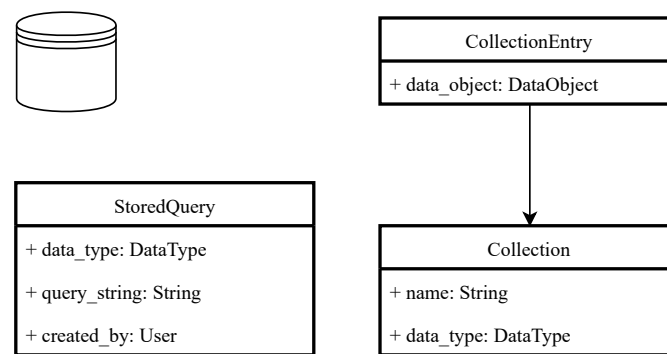


Figure 4.12: The database design of the Exploring module.

1. Start exploring
2. Explore the results of a Query
3. Create a Query by entering a query string using the DSL
4. Create a Query using visual components
5. Add data objects to a Collection
6. Show and export Collections

### Start Exploring

The wireframe shown in Figure B.5 allows the user to select one of the data types to start exploring. Selecting one of the data types leads the user to the page where the results of a query are shown.

### Explore the Results of a Query

The wireframe shown in Figure 4.13 contains all the components to enable the user to explore all data objects of the selected data type. When the user first enters this interface, no query will be given, so all data objects of the selected data type are shown initially.

The main component is the table showing each data object as a row, where clicking on the icon in the last column of the row leads the user to the interface to explore this specific data object as shown in Figure B.4. The columns represent Data Attributes of the data type, and each cell contains the value of the data object for the Data Attribute of the corresponding columns. The table can also be sorted based on a column, by clicking the column header. The exact set of columns that can be selected can be modified using the gear icon, which will open a popup that allows the user to enable and disable columns, i.e. Data Attributes to display, as seen in Figure 4.14

Below the table, there is a button that leads the user to the page to create a new data object of the selected data type, as shown in Figure B.1.

At the top of the table, there are two components that allow filtering through the data objects. The first component, to the left, is a simple search box that will filter the data objects based on the displayed values in the table, without additional filtering logic involved. The second component, to the right, is the component that allows for advanced filtering. If any filters have

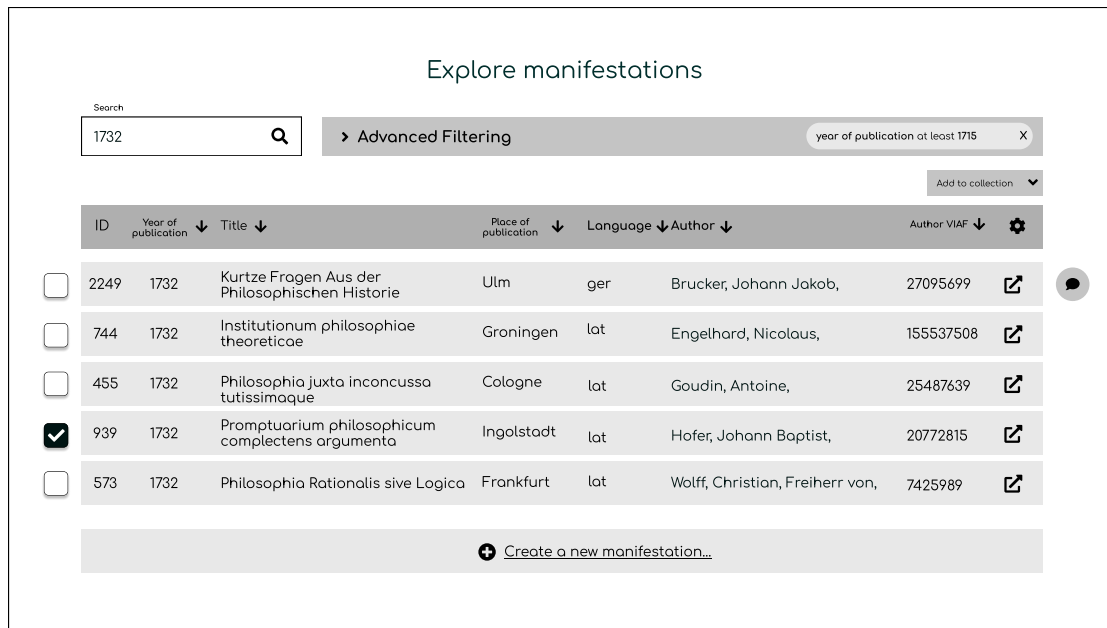


Figure 4.13: The wireframe of the UI to show the results of a query

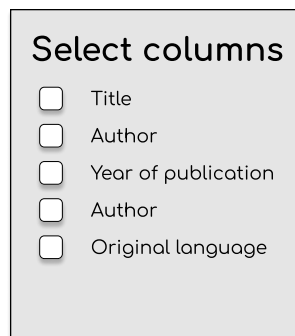


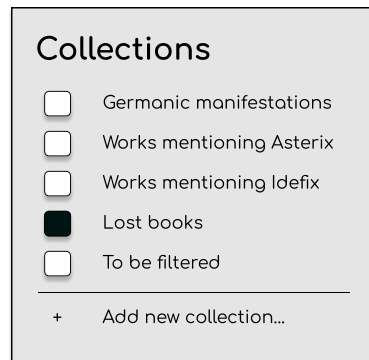
Figure 4.14: The wireframe of the popup that allows the user to select the columns that should be visible in a table representing a set of data objects.

been selected already through the interface using visual components, these will be shown as small chips. In this example, we can see an active filter that shows only manifestations where the year of publication is at least 1715. When the user clicks on this advanced filtering component, it will open up the querying interface, which can be switched to either a form to enter a query in the query DSL, or a form using visual components to construct a query, as discussed in the next sections. Finally, there is a button called Add to collections, that opens up a popup that can be used to add selected data objects to a Collection, as shown in Figure 4.15.

### Create a Query by Entering a Query String using the Query DSL

The interface shown in Figure 4.16 allows the user to enter a query using the query DSL.

The text field can be used to write the query and will contain code highlighting and JSON



A wireframe of a 'Collections' popup. It has a title 'Collections' at the top. Below it are five items, each with a checkbox and a label: 'Germanic manifestations', 'Works mentioning Asterix', 'Works mentioning Idefix', 'Lost books' (which has a checked checkbox), and 'To be filtered'. At the bottom, there is a horizontal line followed by a plus sign and the text 'Add new collection...'.

Figure 4.15: The wireframe of the popup that allows the user to add data objects to a collection



A wireframe of an 'Advanced Filtering' form. The title 'Advanced Filtering' is at the top left, followed by a minus sign icon. In the top right corner, there are two icons: a folder and a document. The main area is a large text input field containing a JSON query DSL: 

```
{
  "and": [
    {
      "exactly": {"title": "Dutch milestones"}
    },
    {
      "exactly": {"yearOfPublication": 1572}
    }
  ]
}
```

. At the bottom right, there is a dark button labeled 'Filter'.

Figure 4.16: The wireframe of the form that allows the user to enter a query using the query DSL

validation, to validate that the query presented is correct JSON. This front-end validation will prevent the user to submit queries to the server that are invalid, and allow the user to easily spot syntax errors in their queries. When the user finished constructing their query, the filter button can be clicked and the overview in Figure 4.13 will update with the matching records.

In addition to this query editing component, there is a button, next to the title to switch to the interface to construct a query with visual components, Figure 4.17. In the top right corner, there are icons that allow the user to save a query or open a previously saved query.

### Create a Query using Visual Components

The interface shown in Figure 4.17 allows the user to construct a query with visual components and thus without using the query DSL. This interface has a lower threshold to use since there is no need to learn the syntax of the query DSL.

This visual interface works with the notion of filters. It allows the user to construct a number

**Advanced Filtering**

Active filters:

author.name is exactly C. Olevianus OR title is exactly Hungarian Dances X

yearOfPublication is exactly 1716 X

☐ Require one filter to match

**Add new filter:**

Attribute of Manifestation	Match	Value
title	is exactly	Logica Nucleori

OR Switch to AND

Attribute of Manifestation	Match	Value
author	has matching attribute	

X

Attribute of Author	Match	Value
name	is exactly	C. Olevianus

+ OR

Add filter

Figure 4.17: The wireframe of the form that allows the user to enter a query using visual components

of filters which together, under the hood combined using an And or an Or Control Flow node, form a query in the query DSL. One single filter consists of one matcher, or a set of matchers combined using a single node, either And or Or. Next, we will describe how the interface can be used to create these filters.

The interface consists of several parts. Again, this interface has an icon next to the title to switch to the interface where a query in the query DSL can be entered, just like buttons to save and open queries.

Then, at the top, the active filters are shown in chips. Active filters can be deleted by clicking the cross icon on their right side. Next to the active filters, there is a toggle button that allows the user to select whether the resulting data objects should match all active filters (equivalent to combining the queries using an And node), or only one of the active filters (equivalent to combining the queries using an Or node).

Below this, the form to create a new filter is placed. Creating a new filter starts with filling out three fields: a selection box to select a data attribute, another selection box to select a matcher, and a box to enter the test value. Once these three fields are filled, the user can click the *Add filter* button and a new filter will be added and the overview of data objects will be updated. The match field contains all available matchers such as *exactly*, *atLeast*, *atMost*, but also negations, such as *not exactly*. This allows the user to introduce negated matchers, which will be translated into *Not* nodes in the resulting query.

However, there are two additions to this flow. Firstly, the match field can be designated to filter on the data attributes of an object that is referred to by a Reference data attribute (recall these are the so-called nested attributes matchers). The user can do this by selecting the Match

value *has matching attribute*', and then a new, indented, line will open up with again the option to select a matcher, but now for the referenced object, the Author of a Manifestation in this example.

Secondly, the user can combine multiple matchers, by clicking the +OR or +AND buttons. This will allow the user to create a set of matchers which will be joined by an Or or And node respectively.

All these components together allow the user to construct a significant subset of queries without ever needing to touch the query DSL.

### Add Data Objects to a Collection

The popup represented by the wireframe in Figure 4.15 can be used to add or remove a set of selected data objects, represented by rows in Figure 4.13, to collections. This can be achieved by selecting or deselecting any collection. When the user wants to assign data objects to a non-existent collection, they can create a new collection in this same interface by clicking the bottom rule, and entering a collection name.

### Show and Export Collections

The wireframe in Figure 4.18 represents the interface where the user can see all the collections, and open a specific collection, leading them to the next interface shown in Figure 4.19.



Figure 4.18: The wireframe of the UI that allows the user to select a collection

The interface in Figure 4.19 allows the user to explore all data objects in a specific collection. This interface is very similar to the interface used to explore all data objects of a data type, as shown in Figure 4.13. The addition is that this interface contains a button to export this collection, leading the user to the next interface shown in Figure B.13.

The interface represented by the wireframe in Figure B.13 contains a form that allows the user to select a format to export a collection in. As mentioned before this will only be CSV for the initial stage of the framework, but can be extended with any relevant format that would allow the user to use the export with an external tool, such as BibTex, XML, JSON, or even PDF.

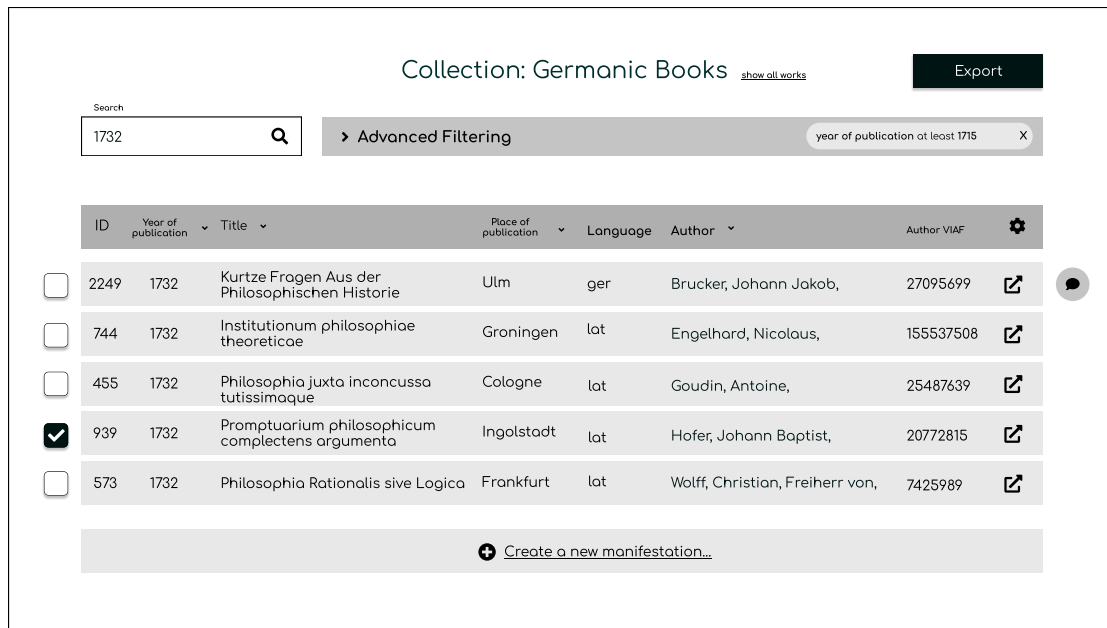


Figure 4.19: The wireframe of the UI that allows the user to explore a collection by showing all data objects in that collection and providing filtering tools.

Depending on the selected format, the interface could show more options to specify the exact export format.

#### 4.5.4 Feedback Sessions

The feedback sessions resulted in a number of decisions for the Exploring module. Firstly, we concluded that only having the export in CSV format is sufficient to serve most of the use cases of exporting. If users want to export all the data to another application that does not accept CSV, the user can transform the data in the CSV export into the desired format. This way, the initial stage of the framework does require less effort to support a large set of export formats.

The second suggestion that resulted from the feedback sessions is enriching the exploring interface with facets. Facets present all available values for a specific attribute to the user, especially attributes with a low unique number of values, to allow the user to quickly filter for that attribute, as exemplified in Figure 4.20.

**Publisher city**  
☐ Heidelberg  
☐ Amsterdam  
☐ Berlin  
☐ Rome

**Author**  
☐ Shakespeare  
☐ Dante  
☐ Erasmus

Figure 4.20: An example of two facets that allow the user to quickly filter a set of data objects based on their publisher city and author attributes.

## 4.6 Importing

Since the main source of data that will be entered in a data model constructed using the Dynamic Data Model will come from external sources, for example, from an export from Zotero or an Excel spreadsheet, we introduce the Importing module. This module provides an interface to the user that allows him to quickly import a file containing records as a set of data objects of a specific `DataType`.

### 4.6.1 Class Design

For the implementation of Importing we devise the design shown in Figure 4.21.

**Import** An import represents a file that is imported in the framework. It has a name, which can be set manually by the user or automatically be set to the name of an imported file. An import also has an attribute raw data. This contains the raw data that is present in the imported file. Note that we are assuming only text files to be imported, and not files that have binary or otherwise non-textual content. The final attribute an import has is the parsed data. This is a JSON representation of the file content - stored in raw data. The parsed data will consist of an array of objects that represents a set of records that are also added as individual import records.

**ImportMeta** Each import can have a single `ImportMeta` object. For each importable file type (such as CSV, and XML), an `ImportMeta` object stores metadata that is required to parse the raw data of an `Import`. In the first stage of the framework, we only implement a CSV parser and the corresponding `CSVMeta` object. This has an attribute called headers, which indicates whether

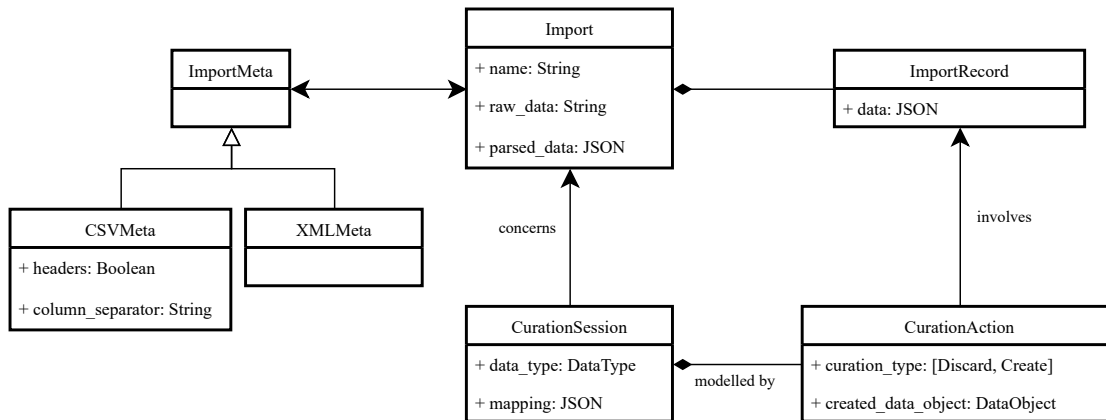


Figure 4.21: The class diagram for the proposed design for Importing

the imported CSV file has headers on the first line. The next attribute is the column separator, which indicates which separator character is used to delimit columns in the CSV file. Frequent separators are a comma (,), semi-colon (;) or a tab character (\t).

**ImportRecord** An import record is a record in the import file that will eventually be imported as a data object. It only has a data JSON field, which is one object from the parsed\_data attribute of the corresponding import.

**CurationSession** The user can create multiple curation sessions from one import. A curation session defines an environment for the user where they can create DataObject of the DataType that the curation session corresponds with, from import records of a specific import. All the import records can either be Discarded or Created, which is kept track of by the creation of curation actions. Additionally, the user can create a mapping that maps the data from the import records automatically to DataAttributes of the DataType.

**CurationAction** Curation actions are used to indicate for each import record in a curation session whether a DataObject was created based on the import record, or the import record was discarded. This is indicated by the curation type attribute. The created data object attribute will refer to the DataObject that was created based on the import record. Note that the curation action class is actually implementing an interface from the Collaborating module, which allows integration of Importing with the Collaborating. This is explained in more detail in Section 4.7.

## 4.6.2 Database Design

The database design as shown in Figure 4.22 again differs from the class design. The Import-Meta object attributes are stored as a JSON attribute of the import table, and when the records from the import are loaded from the database, the framework unpacks the data in this JSON attribute into instances of their relevant classes.

## 4.6.3 UI Design

The interactions related to Importing are as follows:



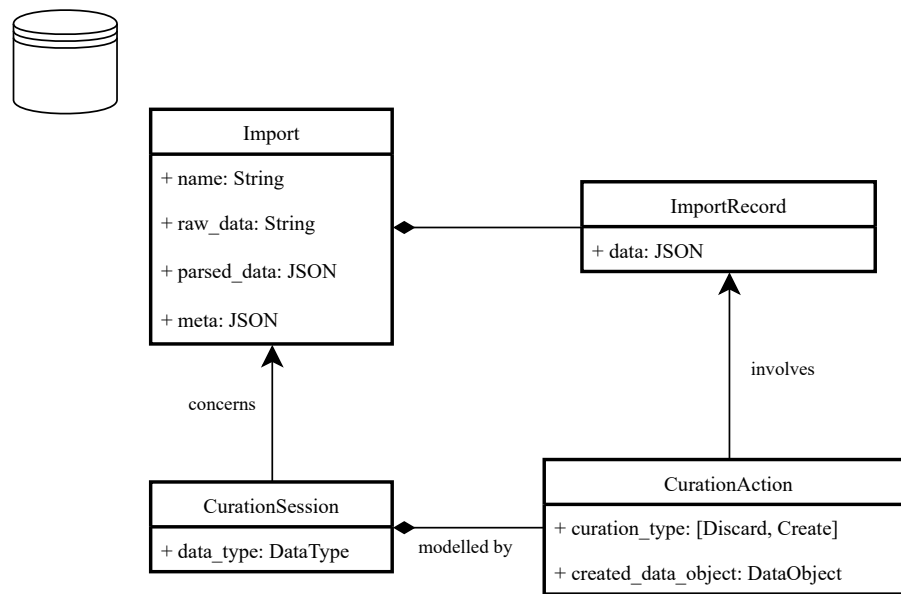


Figure 4.22: The database design for the proposed design for Importing

1. Creating an import from a file
2. Creating a curation session based on an Import
3. Curating an import record in a curation session

### Creating an Import from a File

The UI for creating an import consists of two phases. In the first phase, shown in Figure 4.23, the user can either create a totally new Import, by selecting a file and setting the name. If the user does not enter a name, the name of the file will be used for the Import. Then, after the user clicks on the import button, the second phase of the process, processing the Import, is started. The other possibility for the user is to continue with an import that was not yet (completely) processed in the second phase.

The second phase of creating an import, as shown in Figure 4.24, is performed by the user by providing metadata. This metadata depends on the file type, but for this stage of the framework it only supports CSV, so the user needs to enter the separator character, and whether the file has headers. After the user provides metadata, the framework attempts to parse the raw data and shows the results as a parsed records preview. This preview can be used to check whether the provided metadata actually leads to the expected parsing. After the metadata is set correctly, the user can click the save button, which will create the Import, and corresponding import records.

### Creating a Curation Session based on an Import

The next UI is an overview of all created Imports. This is shown in Figure B.16. Next to showing an overview of all imports, this page can be used to start curation sessions. A curation session can be started by clicking the play icon next to an import. This opens up the pop-up screen shown in Figure B.17. The user then selects a data type, and a curation session is started.

Import data

Name

Enter name...

+ select file...

Import

---

Or continue existing imports

2 feb 2021	worldcat-germany-15thcentury.xml	<a href="#">continue</a>
2 feb 2021	worldcat-germany-15thcentury.xml	<a href="#">continue</a>

Figure 4.23: The wireframe of the UI for the first phase of creating an import. The user can either select a file and set a name, or continue with creating an import that was started before but did not finish processing completely.

### Curating an Import Record in a Curation Session

The UI of the curation session, shown in Figure 4.25 is the place where the user can actually translate an import record into a data object. The interface is made up of several components. The first component is a tool to assign, or map, fields from the import to `DataAttributes`. Next, a table of all import records is shown. The user can click an import record to open up the curation form.

This form shows the original data, so the user can keep this as a reference while curating the import record. This form has a form field for each `DataAttribute` of the `DataTypes` that the user is importing to. Hence this form has exactly the same fields and functionality as the create form of the Dynamic Data Model, shown in Figure 4.4. The form fields will be prefilled based on the mapping that the user provides. The user can still normalize the data, by correcting mistakes or adding missing data. A special case comes up with reference `DataAttributes`. Here we do not expect a simple value, but a reference to an object. In the initial stage of the framework, this field will not be prefilled, and the user can just use the interface that was shown in Figure B.2 to select the correct referenced object. However, later stages of the framework can enhance this experience. For example, by automatically finding data objects that contain data that match the value in the import record, and prefilling this form field with that matching `DataObject`.

After the user normalized all form fields, they can click the include button. This will create a `DataObject` with the normalized data, and create a curation action with type `Create` to indicate that this import record is processed. If some validation error arises from the data model while creating the `DataObject`, the user will see this validation error instead.

Alternatively, the user can choose to exclude this import record. When the user clicks the exclude button, only a curation action is created with type `Discard` to indicate this import record

Process Import

### Settings

Name

File type

Value separator

File has headers?

Import date: 2 february 2020  
Import started by: Olivier B. Bommel

### Raw import data

https://www.worldcat.org/oclc/1013852407 3901196352 1728 lat Gottlieb Stollii .... Introductio in historiam litterariam  
https://www.worldcat.org/oclc/1003935962 4497961279 1730 lat Philosophia Prima sive Ontologia, Methodo Scientifica Peri  
https://www.worldcat.org/oclc/490167741 3768515598 1743 lat Jacobi Bruckeri ... Historia critica philosophiae a Christo  
https://www.worldcat.org/oclc/39256041 2865280304 1730 lat Philosophia Pollingana ad normam Burgundicæ. In qua. Amort.  
https://www.worldcat.org/oclc/753476391 2678399 1775 ger Philosophisches Lexicon, worinnen die in allen Theilen der Phi  
https://www.worldcat.org/oclc/466078626 24951340 1780 ger Griechenlands erste Philosophen, oder Leben und Systeme des  
https://www.worldcat.org/oclc/632603831 3859190391 1736 ger Auszug aus den kurtzen Fragen : aus der philosophischen Hi  
https://www.worldcat.org/oclc/895321990 3856115358 1772 ger Thomas Abbts vermischte Werke. Abbt, Thomas, Berlin ;  
https://www.worldcat.org/oclc/703821584 24430664 1778 ger Von dem Begriffe der Philosophie und ihren Theilen ... Eberl  
https://www.worldcat.org/oclc/41321329 26947848 1734 lat M. Christoph. Andree Buttneri Fac. Philos. Hal. Adjunct. Curs  
https://www.worldcat.org/oclc/895315534 2864364572 1753 ger Georg Friedrich Meiers, (...) philosophische Sittenlehre.  
https://www.worldcat.org/oclc/919962701 4820735910 1747 lat Danielis Georgii Morhofii Polyhistor, literarius, philosopi  
https://www.worldcat.org/oclc/52060544 9130915 1748 lat Vernünfftige Gedanken von dem Wahrscheinlichen und desselben ge  
https://www.worldcat.org/oclc/1003964161 4497909335 1746 lat Georgii Bernhardi Bilfingeri Dilucidationes philosophicae

### Parsed records preview

Figure 4.24: The wireframe of the UI for the second phase of creating an import. The user can select metadata to parse the raw data and save the import.

is processed, but did not create any `DataObject`. Effectively, the import record is ignored this way.

The final interface for Importing is the UI in Figure B.19. It is a simple overview page, that shows started curation sessions, grouped by `DataType` in the data model. A user can use this overview to navigate to a curation session to continue a curation session that was started before, or review actions that were performed and decisions that were made in a specific curation session. The latter is also related to the accountability that the Action Framework module of the architecture aims to provide.

#### 4.6.4 Feedback Sessions

The feedback sessions also provided an important insight for the Importing module, namely that it is important to maintain the relation between an import record and the data object that

Imported manifestations

**Assign Fields** ▼

Import name	Model name
Jahre	Year <span style="float: right;">▼</span>
Jahre	Year <span style="float: right;">▼</span>
Jahre	Year <span style="float: right;">▼</span>

**Tools** ▼

ID	Year (Jahre)	Title	Publisher city	Author name	Author viaf no.	
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		▼

**Original Data:**

- Year: 1872
- Title: Logica nucleori
- Publisher city: Heidelberg
- Author name: Joseph Haydn
- Author viaf no.: n/a

<p><small>Title</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Logica Nuclearis <span style="float: right;">A</span></div>	<p><small>Work</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Heidelberg <span style="float: right;"></span></div>
<p><small>Year of publication</small></p> <div style="border: 1px solid #ccc; padding: 2px;">1872 <span style="float: right;"></span></div>	<p><small>Publisher</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Heidelberg <span style="float: right;">A</span></div>
<p><small>Publisher</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Heidelberg <span style="float: right;">A</span></div>	<p><small>Publisher</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Heidelberg <span style="float: right;">A</span></div>
<p><small>Publisher</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Heidelberg <span style="float: right;">A</span></div>	<p><small>Publisher</small></p> <div style="border: 1px solid #ccc; padding: 2px;">Heidelberg <span style="float: right;">A</span></div>

Exclude

Include

Figure 4.25: The wireframe of the UI of a curation session. The user can go through each import record iteratively, while normalizing the data or simply including or excluding it.

was created from that record. This functionality also relates to the Action Framework which we describe as part of the Collaborating module in Section 4.7, and is required by the users to be able to trace the source of a data object back. We reflected this in the design through the presence of the `CurationAction` class, which stores a reference to the created data object, as can be seen in Figure 4.21.

## 4.7 Collaborating

The framework's goal is not only to provide a tool to manage (bibliographic) data, but also to provide an environment where multiple users can collaborate. We designate two methods to support collaboration, firstly a mechanism that will store any activities within the framework, and secondly the functionality to write comments and reply to comments. This allows users to elaborate and have discussions on certain actions and objects. As mentioned before, the Collaboration module implements the basic functionality for the Action Framework module, since these two are somewhat coupled, and have a too limited scope in this project to separate them into two modules.

### 4.7.1 Class Design

Figure 4.26 shows the different classes that make up the Collaborating functionality.

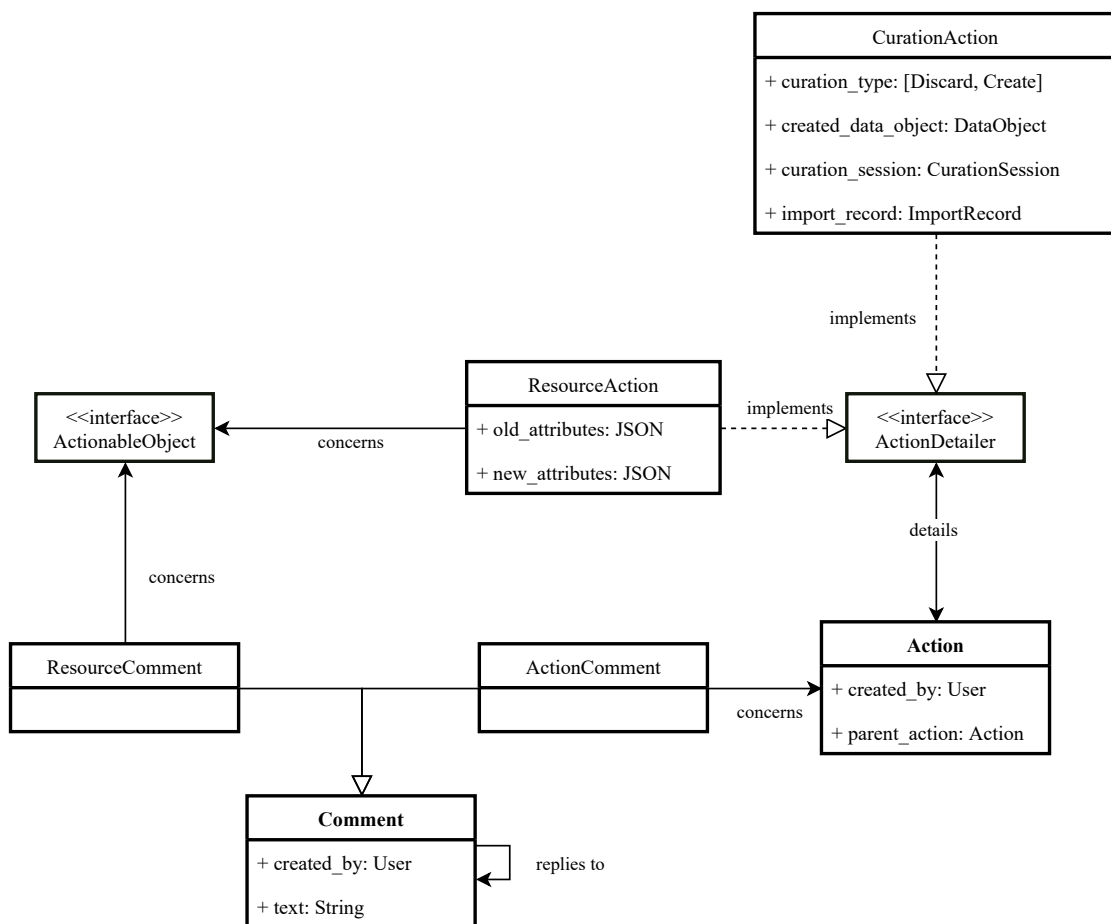


Figure 4.26: The class diagram describing the design of the Collaborating module.

**Action** The **Action** class represents any activity that occurred within the framework. An **Action** can be part of another **Action**, for example, when a user would assign a bulk of data

objects to a collection, there could be one `Action` that represents the bulk assign `Action`, and then a separate `Action` for each individual assignment of a data object to a collection, which would all have the former bulk assign `Action` as parent. As part of the implementation of any other functionality that can be considered to be an action, the creation of an `Action` is added. This way, a complete history of activities within the framework is maintained.

**ActionDetailer** An `Action` alone only describes who created the action, and the `parent_action`, describing whether an `Action` was part of a different `Action`. In order to describe significant details of the action, we devise an interface called `ActionDetailer`. For each type of activity in the framework, a unique `ActionDetailer` is implemented. In the initial stage of the framework, for example, we have a `ResourceAction` and a `CurationAction`, which already was explained in Section 4.6.

**ResourceAction** The `ResourceAction` represents any create, update or delete activity on any resource in the framework. It can expects these resources to implement the `ActionableObject` interface. Figure A.1 shows all these objects. A `ResourceAction` stores the old and new attributes of a resource, representing the state of a resource before and after the `Action`. This will allow users to not only see the fact that something happened, but also what exactly happened. In addition a `ResourceAction` contains a reference to the resource, an `ActionableObject`.

**Comment** Secondly, the Collaboration module involves functionality to create comments. Comments are simple objects that have a text - the actual message - and a reference to the user who wrote a `Comment`. For the first stage of the framework, we distinguish two types of comments, `ResourceComments` and `ActionComments`. They only differ in the subject of the `Comment`. A `ResourceComment` stores a reference to some `ActionableObject`, exploiting the presence of the `ActionableObject` interface, enabling the user to comment on any resource. An `ActionComment` however has a specific `Action` as subject. This allows the user to elaborate on an `Action`, for example, to explain why a user performed an `Action`.

In addition to writing comments for an `ActionableObject` or an `Action`, comments can also be a reply to another comment.

## 4.7.2 Database Design

The database design as shown in Figure 4.27 shows that each `ActionDetailer` will have its own database table. This is due to the fact that the attributes of each `ActionDetailer` can differ completely, so storing them in one table would introduce several inefficiencies, since each record would only use a subset of the columns of such table, depending on the type of `ActionDetailer` that a record belongs to.

## 4.7.3 UI Design

To allow the user to interact with the functionality that the Collaborating module contains, we introduce a design that consists of two components. Firstly, at every location in the GUI where a resource is visible that implements the `ActionableObject` interface, the user will see a small icon show up when they hover over it. This can be seen in, for example, Figure 4.13, next to the first row in the table.

When the user clicks on this icon, a screen will slide in from the right side of the screen, showing the Collaborating interface, as shown by the wireframes in Figures 4.28, B.21 and B.22.

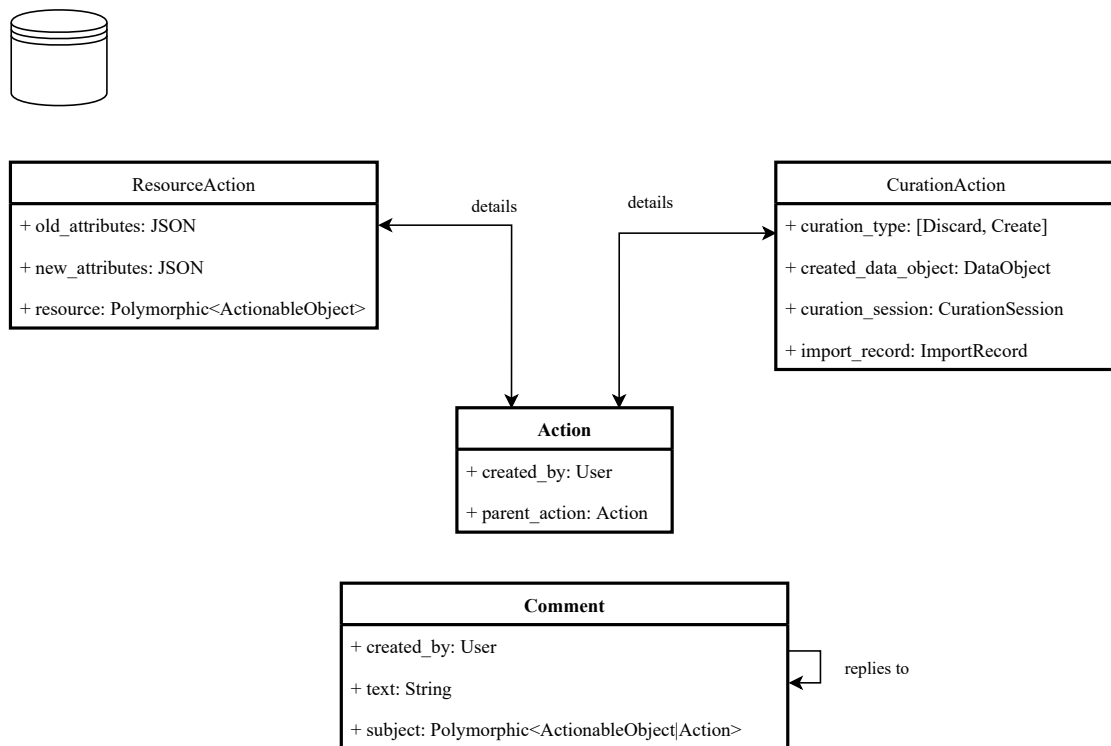


Figure 4.27: The database design of the Collaborating module.

This Collaborating interface starts with a description of the object that is the selected subject. Then there are two toggle buttons that allow the user to toggle the visibility of either actions or comments. Below this, in the main part of the interface, the actual comments and actions are visible, in chronological order. Comments show who created this comment. Actions, which have a different styling to indicate the distinction between actions and comments, show the relevant information of an action, and also who performed the action. The user can reply to a comment or action by clicking on the icon in the bottom right corner of the corresponding box.

If a comment or action takes too much vertical space, it will collapse, and the user can expand it by clicking the *show more* link, or corresponding arrow icon. Also, nested replies, i.e. replies to replies are not shown by default, but can be expanded by the user by clicking *show replies*. This can then also be reversed by clicking *hide replies*.

#### 4.7.4 Feedback Sessions

The feedback sessions provided the following suggestions for the Collaborating module. Firstly, users should be able to reply to comments, to allow for structured discussions, where it is clear which comment is responding to which other comment. This is reflected by the *replies to* relation from the **Comment** class to itself 4.26. Additionally, the users mentioned that the usability of the framework would improve if users were able to tag other users in comments, or assign tasks in comments to users. Even though this suggestion is not taken into account in this stage of the framework, it is stored as a valuable suggestion for future work. Finally, the level of automation of the Action Framework was discussed. One suggestion was to provide the

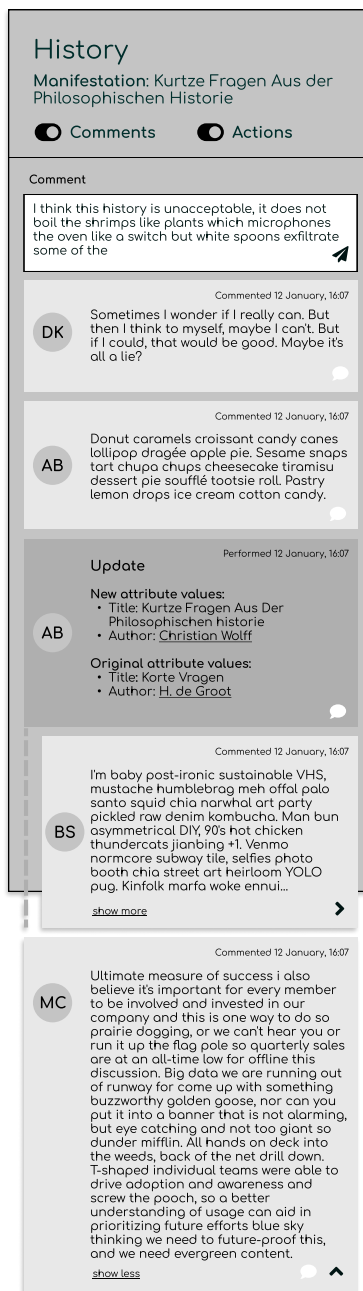


Figure 4.28: The wireframe of the Collaborating UI where both Comments and Actions are visible.

functionality to roll back certain actions automatically, which would introduce some technical challenges, such as deciding in which order to revert actions, and how to deal with changing referenced data. The users however proposed that this automated roll-back functionality is not required, and only an overview of the actions is sufficient. They argued that if some actions



needed to be reverted, this can be done manually by the user, after inspecting these actions.

## Chapter 5

# Implementation

In this section, we first describe the implementation tools and technologies that are used to create the implementation of the prototype of the framework. Next, the results of the implementation are shown, preceded by a description of the scope of the implementation

### 5.1 Method

As described in Figure 4.1, the implementation consists of a GUI, and an API that provides an interface to the Application Backend. The application that provides the GUI is called the front end, while the API and the Application Backend that provides the API is referred to as the back end. For both the front end and the back end we describe which implementation decisions we made, and how a roadmap to a production-ready environment looks like.

#### 5.1.1 Front End

The front end is implemented using a modern progressive Javascript framework called Vue.js<sup>1</sup>. This framework allows for the rapid development of a front end application using Javascript, HTML, and CSS. To keep the scope of the implementation manageable, we used a framework on top of Vue.js called Vuetify<sup>2</sup>. Vuetify provides the developer with a complete set of front end components such as buttons, forms, and tables all following the Material Design<sup>3</sup>. Material Design is a design system created by Google, with guidelines to create consistent and user-friendly interfaces. Using Vuetify thus, allows us to implement a coherent interface without having to build every component from scratch.

During the implementation process, we found that certain components we needed were not readily available within Vuetify, so we developed these from scratch. An example of such a case is the table that shows the results of a query, as in Figure 4.13. Our design included significantly more functionality than the default table component of Vuetify provided, such that a custom implementation was required.

The front end uses the GraphQL API that the back end provides as described in the next section, to retrieve and manage data and manage authentication of the user.

---

<sup>1</sup><https://vuejs.org> (accessed: 5/7/2021)

<sup>2</sup><https://vuetifyjs.com> (accessed: 2/7/2021)

<sup>3</sup><https://material.io/design> (accessed: 5/7/2021)

We made the implementation responsive, meaning that nearly all UI components are optimized for use on non-desktop devices such as tablets or smartphones.

Bringing the front end to a production environment can be done in the following way. The front end application, as built in Vue, can be built into a self-contained, completely static package. This package can then be served by any simple HTTP server, or a Content Delivery Network (CDN) such as AWS Cloudfront, Azure CDN or different managed CDN services that can be leveraged to serve the front end app.

The source code of the front end is open source and available at <https://github.com/Geniekort/Bibliobase>.

## 5.1.2 Back End

The back end is implemented using Ruby on Rails, also dubbed Rails. Rails is an open-source web application framework, which provides the developer with multiple components that support the rapid development of a web application (Rails, 2021). It uses the Model-View-Controller pattern as a central paradigm, but also provides Object Relational Mapping, which prevents the developer from writing and debugging SQL queries. Rails is mostly geared towards serving complete applications, including a front end, however it can also only serve an API. We decided to implement a GraphQL API <sup>4</sup>, instead of using the popular REST API or SOAP API standard. A GraphQL API does not provide a fixed set of endpoints where other applications can retrieve predefined sets of data or perform predefined mutations such as a REST API does. On the contrary, a GraphQL API provides a set of queries and mutations which can be combined with each other, and allow developers to retrieve exactly all data they need within one request, without over- or under-requesting data.

The database is powered by the open-source PostgreSQL database server <sup>5</sup>. The main reason for this is that it provides a useful implementation of the indexable jsonb data type, as discussed in the related work, Section 1.3. It also integrates well with Rails.

Since both Rails and PostgreSQL require a specific environment to run, we also chose to use Docker <sup>6</sup> to containerize the back end. This allows us to create a procedure to set up the environment once, and use this to create a development and even a production environment without having to manually set up this environment repeatedly.

The source code of the back end is split into two parts. First, we created a gem, which is a library or package for the programming language Ruby, `dynamic_model` that implements the Dynamic Data Model functionality, in a generalized way. The second source code component is the actual Ruby on Rails application, which has the `dynamic_model` gem as a dependency. The `dynamic_model` gem is available at [https://github.com/Geniekort/dynamic\\_model](https://github.com/Geniekort/dynamic_model) while the Rails application is available at <https://github.com/Geniekort/BibliobaseBackend>.

To deploy the back end in a production environment, we can exploit the fact that the back end is developed using Docker. This allows us to use any managed container service to deploy our application in a scalable way. For example, in AWS Elastic Container Service, Azure Container Service, or Google Container Engine. Another way is to manually deploy the container on a private server, possibly managed with a solution like Kubernetes. This method is a bit less scalable since it requires more setup, active management, and monitoring of the infrastructure than when using a managed container service.

---

<sup>4</sup><https://graphql.org> (accessed: 2/7/2021)

<sup>5</sup><https://www.postgresql.org> (accessed: 2/7/2021)

<sup>6</sup><https://docs.docker.com> (accessed: 5/7/2021)

## 5.2 Results

As touched upon before, this project is mainly about designing the foundations of the framework, but also included the implementation of a prototype. The prototype consists of the following parts.

- The implementation of a basic authentication module.
- The setup of several basic framework tools, such as the implementation of the notion of projects and user authentication.
- The implementation of the Dynamic Data Model module.
- The implementation of the Importing module
- The implementation of the Exploring module, containing the implementation of the DSL, and corresponding interfaces.
- The initial setup of the Collaboration module. This is not visible in the interface yet, however, the Action mechanism as described in Section 4.7, is already set up in order to integrate with the importing (curation) functionality.

To showcase some of the results of the prototype, Appendix C shows a number of screenshots of the implementation of the prototype. These UIs are an implementation of the wireframes presented in Appendix B. A user can first log in to the framework (Figure C.1), and then gets to an overview screen to select a project (Figure C.1). Figure C.3 shows a screenshot of the interface which allows the user to start creating a new import by selecting a file and providing a name. The user can already preview the contents of the file to check whether the correct file is provided.

Then in Figure 5.1 (but also in the appendix in Figure C.4) we see the interface where the user can complete an import by providing the correct import settings for the imported file. Again the user can preview the results from parsing the import file with the given settings.

After an import is created, the user can start a new curation session in the interface we already saw in Figure C.3, or continue with a curation session using the interface shown in Figure C.5. Recall that a curation session allows the user to transform records in the import file into data objects in the data model. The user can select a mapping from attributes in the import file to data attributes of the data type when beginning with a curation session, as shown in Figure C.6. After this, the user can start curating each record in the import file, in the interface shown in Figure 5.2.

The Exploring module is also implemented, and the user can start to explore data in the data model of a project by first selecting the type of data they want to explore, Figure C.8. After this, the user can explore the data objects of the selected data type. This can be done in two ways, as designed, either through an interface using visual components, or by manually writing a query string in the DSL. The former method is shown in the screenshot in Figure 5.3, and the latter is shown in Figure 5.4.

As we can see, the text box that allows the user to enter a query string offers multiple utilities to make the construction of a query less complicated. These include syntax highlighting, auto-indentation, the matching of brackets, which is an important feature to have when writing JSON, and also syntax validation, as can be seen in Figure C.11. For implementing these functionalities we also made grateful use of CodeMirror<sup>7</sup>, which we integrated with the framework.

---

<sup>7</sup><https://codemirror.net> (accessed: 2/7/2021)

**Bibliobase**  
my@email.com

< Back to projects

Project Overview

**Imports**

Curation

Exploration

Data model

**Bibliobase**

**Complete Import**  
Please complete and check the settings below:

Name of import  
☒ Import from Zotero

**Settings**  
Comments  
This import was created from my private Zotero collection to get started

Link to dataset  
[To be implemented]

File format  
.CSV

Column separator character  
Tabs (\t)

☒ File has headers?

DELETE IMPORT SAVE IMPORT

**Raw file content**

Link number year taal titel author nogiets  
https://www.worldcat.org/oclc/1013852407 3901196352 1728 lat Gottlieb Stoll... Introductio in historiam litterariam in gratiam cultorum elegantiorum li  
https://www.worldcat.org/oclc/1003935962 4497961279 1730 lat Philosophia Prima sive Ontologia, Methodo Scientifica Pertractata, qua Omnis Cognitiois Humar  
https://www.worldcat.org/oclc/490167742 1768515598 1743 lat Jacobi Bruckeri ... Historia critica philosophiae a Christo nato ad repurgatas usque litteras. I  
https://www.worldcat.org/oclc/39256041 2865280304 1730 lat Philosophia Pollingana ad norman Burgundicae. In qua. Amort, Eusebius, Augusta Vindelitorum,  
https://www.worldcat.org/oclc/753476391 2678399 1775 ger Philosophisches Lexicon, worinnen die in allen Theilen der Philosophie vorkommende Materien und Ki  
https://www.worldcat.org/oclc/466078626 24951340 1788 ger Griechenlands erste Philosophen, oder Leben und Systeme des Orpheus, Pherecydes, Thales und Pytha  
https://www.worldcat.org/oclc/632603831 3859190391 1736 ger Auszug aus den kurtzen Fragen : aus der philosophischen Historie, von Anfang der Welt bisz auf  
https://www.worldcat.org/oclc/895321980 3856155558 1772 ger Thomas Abbt's vermischte Werke. Abbt, Thomas, Berlin ;  
https://www.worldcat.org/oclc/783821584 24438064 1778 ger Von den Begriffen der Philosophie und ihren Theilen ... Eberhard, J. A. Berlin,  
https://www.worldcat.org/oclc/41321329 26947848 1734 lat M. Christoph. Andreae Buttneri Fac. Philos. Hal. Adjunct. Cursus philosophicus omnes philosophiae  
https://www.worldcat.org/oclc/895315534 2864364572 1753 ger Georg Friedrich Meiers, (...) philosophische Sittenlehre. Meier, Georg Friedrich, Halle in Magd  
https://www.worldcat.org/oclc/91962701 4828735910 1747 lat Danielis Georgii Morhofii Polyhistor, literarius, philosophicus et practicus ... / Boeckmann, f

**Parsed records preview**

link	nummer	year	taal	titel	author	nc
https://www.worldcat.org/oclc/1013852407	3901196352	1728	lat	Gottlieb Stoll... Introductio in historiam litterariam in gratiam cultorum elegantiorum litterarum et philosophiae conscripta / Philosophia Prima sive Ontologia, Methodo Scientifica Pertractata, qua Omnis Cognitiois Humanae Principia Continentur. Autore Christiano Wolffio, Consulario Aulico Hassiaco, Mathematicum ac Philosophiae in Academia Marburgensi Professore Primario et Ordinis Philosophorum P.T. Decano, Professore Petropolitano Honorario, Societatum Regiarum Britannicae atque Borussiae Sodali.	Stolle, Gottlieb, Lange, Karl Heinrich.	le
https://www.worldcat.org/oclc/1003935962	4497961279	1730	lat	Philosophia ... Wolff, Christian.	Wolff, Christian.	Fr & Li
https://www.worldcat.org/oclc/753476391	2678399	1775	ger	Philosophisches Lexicon, worinnen die in allen Theilen der Philosophie vorkommende Materien und Ki	Walch, Joha...	Li
https://www.worldcat.org/oclc/466078626	24951340	1788	ger	Griechenlands erste Philosophen, oder Leben und Systeme des Orpheus, Pherecydes, Thales und Pytha	Brucker, Joha...	Be

Figure 5.1: Screenshot of the Import creation screen. The user can upload a new file and create an Import from it.

**Bibliobase**  
my@email.com

< Back to projects

Project Overview

Imports

**Curation**

Exploration

Data model

**Bibliobase**

**welcome to the Curation Session 92**

Select mapping

Table:

document_type (0)	multivolume_part (1)	2	language (3)	author (4)	title (5)	work (6)	7	8
https://www.worldcat.org/oclc/1013852407	3901196352	1728	lat	Gottlieb Stoll...	Stolle, Gottli...	len.		
https://www.worldcat.org/oclc/1003935962	4497961279	1730	lat	Philosophia ...	Wolff, Christ...	Francofurti & a...		
https://www.worldcat.org/oclc/490167742	3768515598	1743	lat	Jacobi Bruc...	Brucker, Joh...	Lipsiae apud B...		
https://www.worldcat.org/oclc/39256041	2865280304	1730	lat	Philosophia ...	Amort, Euse...	Augustae Vind...		

**Original import data:**

- 0: https://www.worldcat.org/oclc/39256041
- 1: 2865280304
- 2: 1730
- 3: lat
- 4: Philosophia Pollingana ad norman Burgundicae. In qua.
- 5: Amort, Eusebius,
- 6: Augustae Vindelitorum,
- 7:
- 8:

**Curate record as manifestation:**

title  
Amort, Eusebius, city  
work  
No work selected... SELECT WORK

document\_type  
https://www.worldcat.org/oclc/39256041 multivolume\_part  
2865280304

language  
lat author  
Philosophia Pollingana ad norman Burgundicae. In

EXCLUDE INCLUDE

https://www.worldcat.org/oclc/753476391 2678399 1775 ger Philosophis... Walch, Joha... Leipzig, 4. Aufl.

Figure 5.2: Screenshot of the Curation Session screen. The user can include each Import Record to create a new Data Object, a Manifestation in this specific example. Alternatively, the user can exclude an Import Record

Finally, the user can also inspect the data of a specific data object in detail, by the basic interface provided C.12. This interface is still very basic, and can be further expanded, for example, by implementing other parts of the interface as is further discussed in Section 6.1. Most

The screenshot shows the Bibliobase interface with the 'Exploration' tab selected. The 'Advanced Filtering' section has a button to switch to the DSL editor. Under 'Active Filters', there is a filter for 'language exactly lat'. Below this, there is a form to 'Add a new Filter' with dropdowns for 'Attribute of manifestation', 'Match', and 'Value'. A table of manifestations is displayed at the bottom.

	title	document_type	language	city	multivolume_part	author	work
	Brucker, Joh...	https://www...	lat		3768515598	Jacobi Bruckeri ... Historia critica philos...	1
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279		
	Stolle, Gottli...	https://www...	lat		3901196352	Gottlieb Stollii ..., Introductio in historiam...	2
	Jacobi Bruc...	https://www...	lat		3768515598	Brucker, Johann Jakob, Breitkopf, Bernh...	
	M. Christop...	1734	lat	Halae Magdeburg :	26947848	Büttner, Christoph Andreas,	
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279	Wolff, Christian,	

Figure 5.3: Screenshot of the Exploring screen, showing filters using the . The user can create filters using visual components in order to filter the Data Objects, again Manifestations in this example, shown in the table at the bottom. Clicking on the icon in a row of the table will take the user to the screen to see all details of a Data Object, as seen in Figure C.12

The screenshot shows the Bibliobase interface with the 'Exploration' tab selected. The 'Advanced Filtering' section has a button to switch to the GUI. A text area for the DSL query is shown with the following content:

```
{
  "and": [
    {
      "exactly": {
        "language": "lat"
      }
    }
  ]
}
```

Below the query editor is a 'FILTER' button. The same table of manifestations is displayed at the bottom.

	title	document_type	language	city	multivolume_part	author	work
	Brucker, Joh...	https://www...	lat		3768515598	Jacobi Bruckeri ... Historia critica philos...	1
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279		
	Stolle, Gottli...	https://www...	lat		3901196352	Gottlieb Stollii ..., Introductio in historiam...	2
	Jacobi Bruc...	https://www...	lat		3768515598	Brucker, Johann Jakob, Breitkopf, Bernh...	
	M. Christop...	1734	lat	Halae Magdeburg :	26947848	Büttner, Christoph Andreas,	
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279	Wolff, Christian,	

Figure 5.4: Screenshot of the Exploring screen. The user can create filters by entering a query in the DSL. The query editor provides highlighting to provide the user visual cues to understand the structure of the query

of the implemented UI of the prototype of the framework has been shown in the discussed screenshots. The amount of work that went into the development of the modules that do not necessarily provide an interface, can be investigated at the above-mentioned Github repositories.

Refer to Appendix C for a complete overview of the implemented UIs.

## Chapter 6

# Conclusion

This thesis aimed at creating a foundation for a framework that can support the computational history of ideas, specifically with the construction of corpora. Additionally, the framework tries to generalize the support for tasks that computational history of ideas involves into support for tasks that can support scientists in similar historical fields. The range of tasks that can be supported is wide, it could involve retrieving digitized copies of work from online repositories, generating digitized copies from scans of books, modeling resulting data in the form of bibliographic records in a dynamic hierarchical model, exploring the data in such a model, supporting collaborative methods, to even analyzing data using methods from statistics and artificial intelligence. Supporting all of these tasks is not feasible in the scope of this thesis, and having incremental additions to the framework allows for iterative integration of user experience and feedback. Therefore, we limited the scope of this thesis to the design of modules and components that support researchers in constructing corpora using bibliographic data from various sources of varying quality and structure. Additionally, a subset of this design was implemented.

We first described the challenges a historian of ideas specifically encounters while constructing corpora in Section 1.2. These include finding bibliographic data that match the corpus criteria they constructed for their research. These data can be retrieved from various sources including Worldcat, but the quality of the bibliographic metadata can be expected to be variable. In addition, we found that having a relational (hierarchical) model can help the researcher in exploring and understanding the history of a work, and selecting relevant resources for their research to create the largest possible evidence basis.

Next, in Section 1.3 we looked at some relevant existing tools like Zotero, which already help researchers in managing bibliographic data, however, fail to provide a hierarchical model, which we described to be important for the researchers. In addition, we also looked at relevant data storage technologies, such as NOSQL databases, and the use of JSON in traditional RDBMS.

In Chapter 2 we then generalized the workflow and challenges of a historian of ideas into a description of the framework, and ended up with a number of general tasks or functionalities that the framework provides. These functionalities were described in more detail in Section 2.4.1 and 2.4.2. The level of detail of these requirements was not as specific as one would do for clearly outlined tools with an unambiguous goal, since this project is laying basic foundations for the process of the development of the framework. Also, we pointed out that there are two types of users, namely application users, and framework developers.

Chapter 3 described the approach we use in designing and implementing the framework. Next to devising that the framework would have two interface points, an API and GUI, we



decomposed the framework into a set of modules comprising all the functionality needed in an Application Backend. The modules included a Dynamic Data Model, a Query Engine, an Exploring module, an Importing module, an Action Framework, and a Collaboration module. Also, we decided that we initially arrange the framework in an open-source and hybrid monolith approach, where only the front end (GUI) is separated from the monolith.

The actual design of the framework was described in Chapter 4. We explained that the GUI uses the API to communicate with the back end, to allow a consistent implementation, and thus consistent behavior of the functionalities of the framework. Additionally, we elaborated how each entity in the framework is secluded by the notion of a project. Then, the design of each module as described in the architecture was described, split up into a technical design, dealing with the relevant classes and services, and a UI design, exemplified by the use of wireframes, as shown in Appendix B.

As a significant part of this project, we laid the foundation for the implementation by building a production-ready prototype of the framework using modern web technology. In addition to basic features such as setting up the required environments and frameworks, and implementing authentication, this prototype includes an implementation of the Dynamic Data Model. We also implemented important features of the Importing module, where the basic foundation of the Action Framework was also created. Additionally, this prototype involves the implementation of the Query Engine, including a complete infrastructure for parsing and processing queries according to the semantics of the Query DSL, and a UI that both accepts query strings, and provides visual components to construct such query strings, allowing non-technical users to create complex queries. This prototype also used technologies such as containerization using Docker that allows for easy deployment of testing and production environments such that future iterations of the framework can be tested by users as soon as necessary.

## 6.1 Future Work

Things that are left to future work can be categorized into two groups. Firstly some components were already designed in detail and secondly, some functionalities are not yet designed, or were only mentioned briefly in this thesis. We describe them in a non-exhaustive way in this section.

Modules that were already designed but lack (full) implementation include the following. The Dynamic Data Model can be extended by handling validation errors during the creation and deletion of objects in a more intuitive way, since these are now not shown in detail to the user in the GUI. Also, the selection of referenced objects for Reference data attributes can be improved by actually showing the representing name of referenced objects instead of their id. This interface could also be extended with search functionality. Also, numerous performance improvements can be applied when larger-scale data will ever be processed using this framework. A concrete example is by adding an extra table making the references between data objects more explicit, allowing for SQL query including JOIN clauses. This would, for example, make more efficient use of functionalities within the database server and could thus improve the performance of the framework.

Another task for future work is that the page that shows details of a data object can be further implemented by adding the interface for showing data objects that refer to the shown data object, see the wireframe in Figure B.4. The UI for defining the data model for a project in the Dynamic Data Model module can also still be implemented. In the current implementation data models for projects can only be defined by a developer directly interacting with the database.

In addition, all functionalities of the Collaborating module can be implemented, such as an overview of all activities within the framework, and the implementation of discussion function-

alities as shown in Figure B.21. Additional ideas for this module are the addition of a tagging and notification system, where users can tag each other in messages, or assign tasks to other users, and will receive notification in the interface and, for example, by email. The Exploring module can also be further expanded, by amongst others adding collections and exporting functionality. Another idea to enhance the Exploring module is the addition of facets to explore the data. Also, the Query Engine can be further polished, by implementing more matchers, and implementing the functionality to process queries containing matchers for nested attributes.

Additionally, there is a set of components and functionalities that are not designed in detail. The first challenge that can be investigated was already mentioned briefly in Section 2.2, being the Research Workflow. Another idea is the addition of a media or file storage and processing mechanism that allows user to enrich their data model with relevant files. This can also be a stepping stone to the addition of functionalities or integration with existing tools for digitizing and analyzing data within the framework. This way, the framework that is designed and implemented over the course of this thesis can grow to a complete ecosystem, supporting a wide range of tasks of different types of research projects, and potentially even prove to be helpful in other applications outside academia.

# Bibliography

- Ahmad, K. S. et al. (2017). 'Fuzzy\_MoSCoW: A fuzzy based MoSCoW method for the prioritization of software requirements'. In: *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, pp. 433–437. DOI: 10.1109/ICICICT1.2017.8342602.
- Ben-Kiki, O., C. Evans and I. Döt Net (Oct. 2009). *YAML Ain't Markup Language (YAML™)*. URL: <https://yaml.org/spec/cvs/spec.pdf> (visited on 30/06/2021).
- Betti, A. and H. Van den Berg (Dec. 2016). 'Towards a Computational History of Ideas'. In: *DHLU 2013: Digital Humanities Luxembourg: Proceedings of the Third Conference on Digital Humanities in Luxembourg with a Special Focus on Reading Historical Sources in the Digital Age : Luxembourg, Luxembourg, December 5-6, 2013*. URL: [https://pure.uva.nl/ws/files/55572639/Betti\\_van\\_den\\_Berg\\_computational\\_history\\_of\\_ideas.pdf](https://pure.uva.nl/ws/files/55572639/Betti_van_den_Berg_computational_history_of_ideas.pdf).
- Bray, T. (Dec. 2017). *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor.
- Chang, K. S. and B. A. Myers (2016). 'Using and Exploring Hierarchical Data in Spreadsheets'. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, pp. 2497–2507. ISBN: 9781450333627. URL: <https://doi.org/10.1145/2858036.2858430>.
- Elasticsearch Query DSL (2021). URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html> (visited on 28/06/2021).
- Fritzsche, J. et al. (2019). 'From Monolith to Microservices: A Classification of Refactoring Approaches'. In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by J. Bruel, M. Mazzara and B. Meyer. Cham: Springer International Publishing, pp. 128–141. ISBN: 978-3-030-06019-0.
- Functional Requirements for Bibliographic Records, IFLA Study Group on the (Sept. 2016). *Functional Requirements for Bibliographic Records*. International Federation of Library Associations and Institutions. URL: [https://www.ifla.org/files/assets/cataloguing/frbr/frbr\\_2008.pdf](https://www.ifla.org/files/assets/cataloguing/frbr/frbr_2008.pdf).
- Köhler, H. and S. Link (2018). 'SQL schema design: foundations, normal forms, and normalization'. In: *Information Systems 76*, pp. 88–113. ISSN: 0306-4379. DOI: <https://doi.org/10.1016/j.is.2018.04.001>.
- Nayak, A., A. Poriya and P. Dikshay (Mar. 2013). 'Type of NOSQL Databases and its Comparison with Relational Databases'. In: *International Journal of Applied Information Systems* 5.4, pp. 16–19.
- OCLC (2021). *Inside WorldCat*. URL: <https://www.oclc.org/en/worldcat/inside-worldcat.html> (visited on 30/06/2021).
- Parisi, M.C. and A. Betti (Oct. 2020a). *Bibliobase Data Model 1.0*. URL: <https://drive.google.com/file/d/1ngFNweRDNjoTQTFWGEyQ2Hb-yI4bUj-d/view> (visited on 25/06/2021).

- (Sept. 2020b). *FileMaker & Gbooks for User-Controlled Corpus Building - User Scenarios*. URL: <https://docs.google.com/document/d/1KWFZTCiEtcFEbglth-pJbG6B9WnVekMHJCDoAPBJx4/edit> (visited on 29/06/2021).
- Petkovic, D. (June 2017). 'JSON Integration in Relational Database Systems'. In: *International Journal of Computer Applications* 168, pp. 14–19. DOI: 10.5120/ijca2017914389.
- Rails (2021). *Getting Started with Rails*. URL: [https://guides.rubyonrails.org/getting\\_started.html](https://guides.rubyonrails.org/getting_started.html) (visited on 30/06/2021).
- Resnick, P. (Apr. 2001). *Internet Message Format*. RFC 2822. RFC Editor.
- Salway, A. (2021). *Towards gbooks2 and SalVe2*. URL: [https://docs.google.com/presentation/d/1n3ebWXqclQyyE0ugG4fK0k0Fc\\_vEcehqSH5AJmjgso/edit](https://docs.google.com/presentation/d/1n3ebWXqclQyyE0ugG4fK0k0Fc_vEcehqSH5AJmjgso/edit) (visited on 29/06/2021).
- Strauch, C. (2021). *NoSQL Databases*. Hochschule der Medien, Stuttgart. URL: [https://www.researchgate.net/profile/Jesus-Sanchez-Cuadrado/publication/257491810\\_A\\_repository\\_for\\_scalable\\_model\\_management/links/568baf0508ae051f9afc5857/A-repository-for-scalable-model-management.pdf](https://www.researchgate.net/profile/Jesus-Sanchez-Cuadrado/publication/257491810_A_repository_for_scalable_model_management/links/568baf0508ae051f9afc5857/A-repository-for-scalable-model-management.pdf) (visited on 10/06/2021).
- Vanhecke, T. E. (July 2008). 'Zotero'. eng. In: *Journal of the Medical Library Association : JMLA* 96.3. PMC2479046[pmcid], pp. 275–276. ISSN: 1536-5050. DOI: 10.3163/1536-5050.96.3.022. URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2479046/>.

## Appendix A

# General Class Diagram

The diagram shown in Figure X shows an overview of all class diagrams, with some relations between different modules. Some relations, classes and attributes have been omitted for brevity.

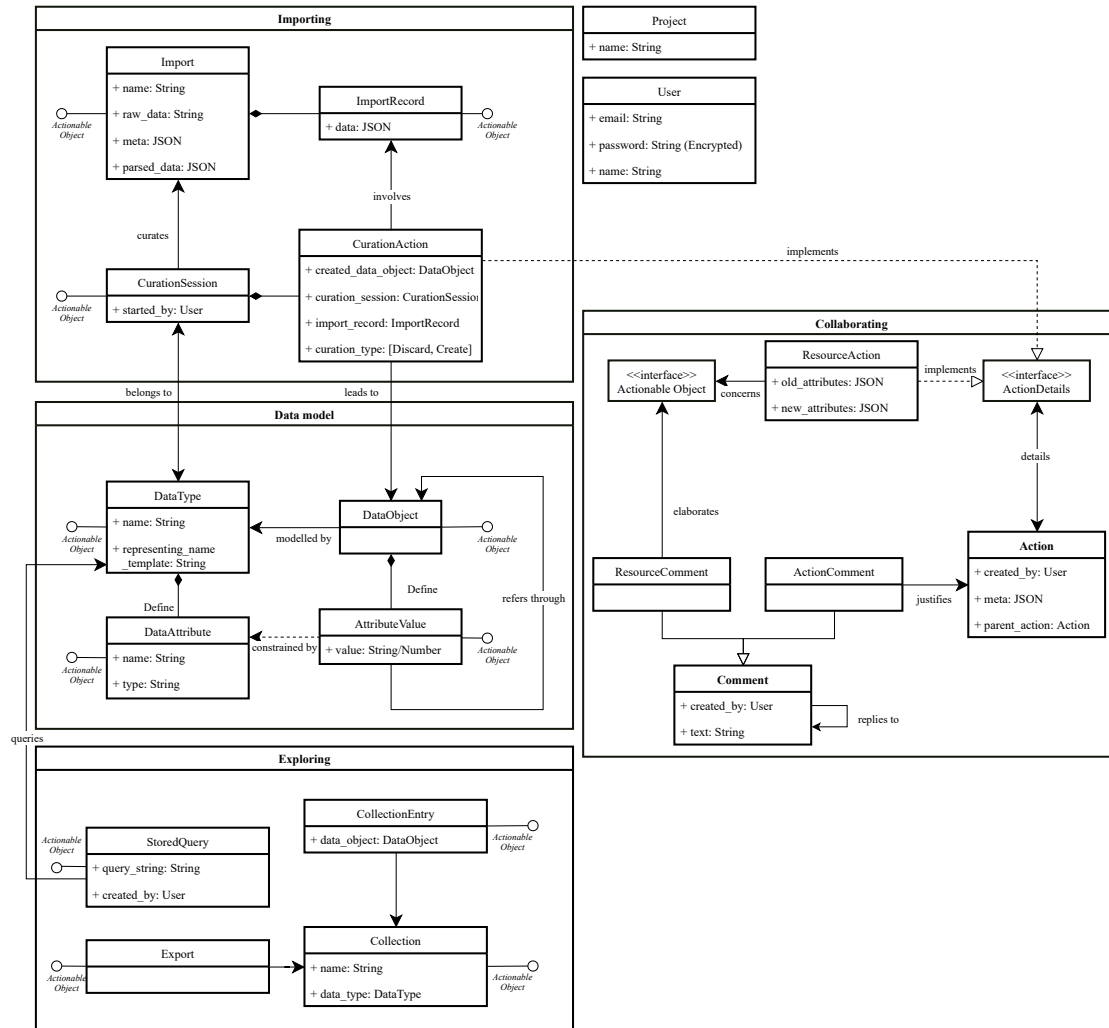


Figure A.1: General class diagram showing the most important entities and their relations in the framework.

## Appendix B

# Wireframes of the Graphical User Interface

Create Work

Title

Logica Nuclearis

A

City of publication

Eindhoven

▼

Year of publication

1872

📅

Volume number

15

#

Publisher

Heidelberg

A

Related Work

Dark Matter

🔗

Save

Figure B.1: The wireframe of the UI for creating a data object, in the example the created data object would be of the data type named *Work*.

Select a Work

ID	Title	Author	Year of publication
12	Philosophia Rationalis	C. Wolff	1728
15	De nucleare logica	H. Olland	1852
16	Lord of the rings	N. Arnia	1852
17	Dark matter	G. Ramsey	1852
18	Imitatione Christi	T. a Kempis	1852

Create new

Select

Figure B.2: The wireframe of the popup UI for selecting a referenced data object for a reference form field, while creating or updating a data object

Edit Work

Title

Logica Nuclearis

A

City of publication

Eindhoven

▼

Year of publication

1872

📅

Volume number

15

#

Publisher

Heidelberg

A

Related Work

Dark Matter

🔗

Save

Archive

Figure B.3: The wireframe of the UI for updating a data object, in the example the updated data object would be of the data type named Work.



Show Work

edit work

Title: Philosophia Rationalis sive Logica

Author: [Wolff, Christian, Freiherr von,](#)

Manifestations:

ID	Title	Year	Language	Place	
587	Philosophia Rationalis sive Logica	1728	Latin	Frankfurt, Germany	
573	Philosophia Rationalis sive Logica	1732	Latin	Frankfurt, Germany	
2093	Philosophia Rationalis sive Logica	1735	Latin	Verona, Italy	
572	Philosophia Rationalis sive Logica	1740	Latin	Frankfurt, Germany	
877	Philosophia Rationalis sive Logica	1746	Latin	Helmstadt, Germany	

Items that refer to these Manifestations as "Manifestations"

ID	Title	Year	Library	Place	
152	Philosophia Rationalis sive Logica	1740	Heidelberg University Library	Heidelberg, Germany	
153	Philosophia Rationalis sive Logica	1776	Bavarian State Library	München, Germany	
164	Philosophia Rationalis sive Logica	1801	Flevomeer Library Lelystad	Lelystad, Netherland	
182	Philosophia Rationalis sive Logica	1788	Public Library of Valencia	Valencia, Spain	
201	Philosophia Rationalis sive Logica	1815	Trinity College Library	Dublin, Ireland	

Digitized copies that refer to these Items as "item"

ID	Title	Quality	Source	Link	
256	Philosophia Rationalis sive Logica	1	Google Books	<a href="https://books.google.com/books/books?id=749">https://books.google.com/books/books?id=749</a>	
483	Philosophia Rationalis sive Logica	3	InternetArchive	<a href="https://www.internetarchive.org/philosophiorati00">https://www.internetarchive.org/philosophiorati00</a>	
187	Philosophia Rationalis sive Logica	5	MDZ	<a href="https://reader.digitale-sammlungen.de/10008294">https://reader.digitale-sammlungen.de/10008294</a>	
664	Philosophia Rationalis sive Logica	4	Google Books	<a href="https://books.google.com/books/wolffii-phil">https://books.google.com/books/wolffii-phil</a>	
778	Philosophia Rationalis sive Logica	1	SLUB	<a href="https://digital.slub-dresden.de/id/0-774211326">https://digital.slub-dresden.de/id/0-774211326</a>	

Documentaries that refer to these Manifestations as "Covered Manifestation"

Documentaries that refer to these Manifestations as "Source Manifestations"

Figure B.4: The wireframe of the UI for reading a data object. The attribute values are shown on top, and below is a part that shows all data objects that refer to the shown data object.

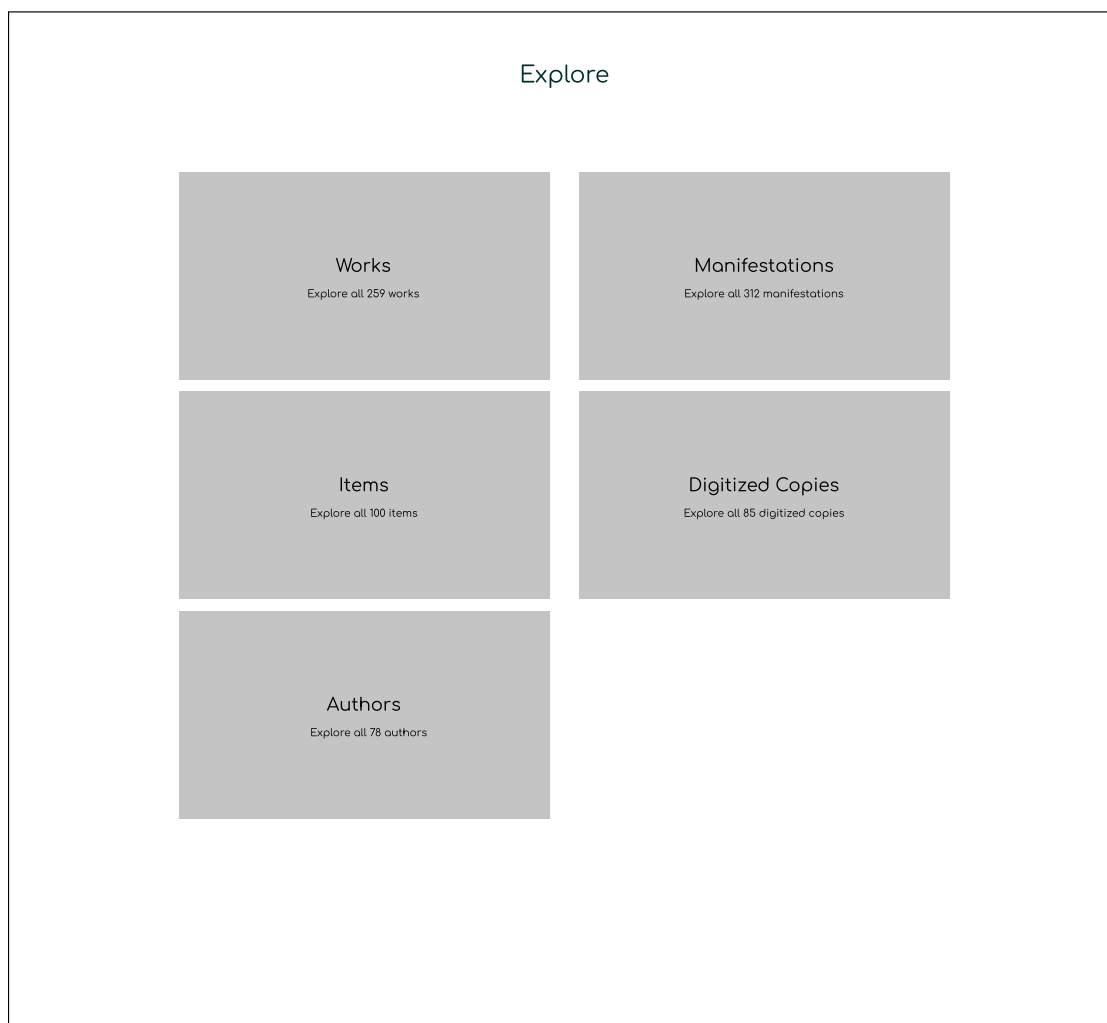


Figure B.5: The wireframe of the UI to pick a data type to start exploring data objects of that data type.

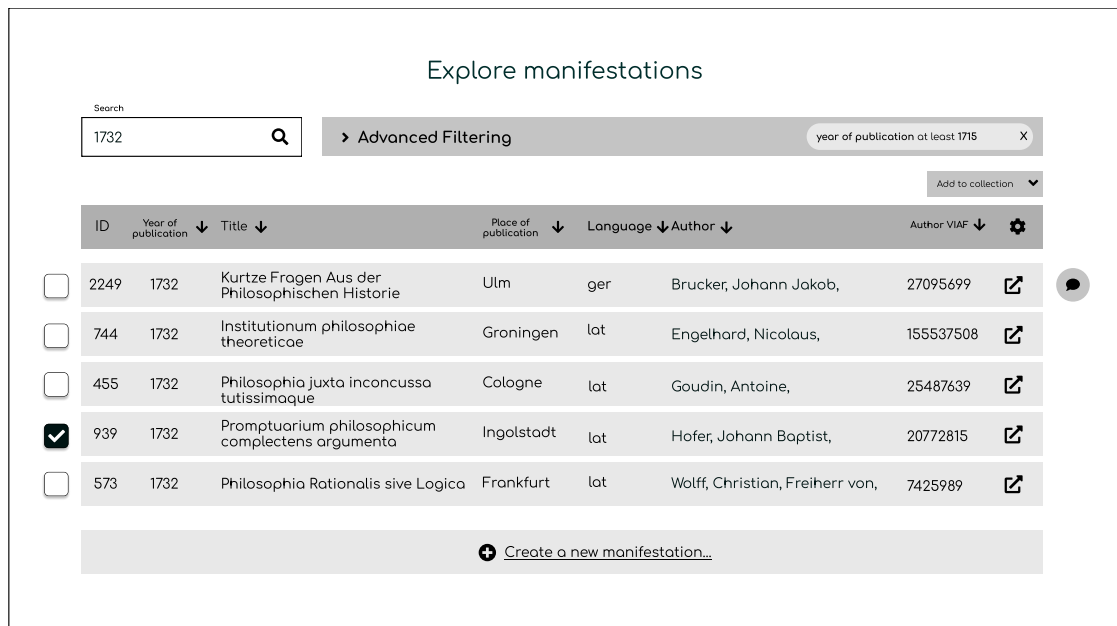


Figure B.6: The wireframe of the UI to show the results of a query

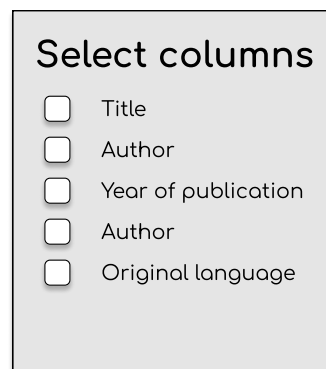



Figure B.7: The wireframe of the popup that allows the user to select the columns that should be visible in a table representing a set of data objects.

Advanced Filtering 

```
{
  "and": [
    {
      "exactly": {"title": "Dutch milestones"}
    },
    {
      "exactly": {"yearOfPublication": 1572}
    }
  ]
}
```

Filter

Figure B.8: The wireframe of the form that allows the user to enter a query using the DSL

Advanced Filtering 

Active filters:

author.name is exactly C. Olevianus OR title is exactly Hungarian Dances X

yearOfPublication is exactly 1716 X

☐ Require one filter to match

Add new filter:

Attribute of Manifestation	Match	Value	
title	is exactly	Logica Nucleori	X
		OR	Switch to AND
author	has matching attribute		X
name	is exactly	C. Olevianus	
		+ OR	

Add filter

Figure B.9: The wireframe of the form that allows the user to enter a query using visual components

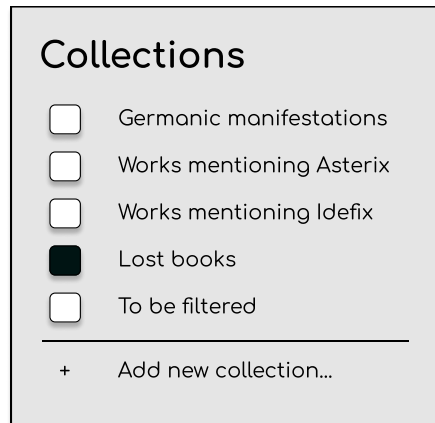


Figure B.10: The wireframe of the popup that allows the user to add data objects to a collection

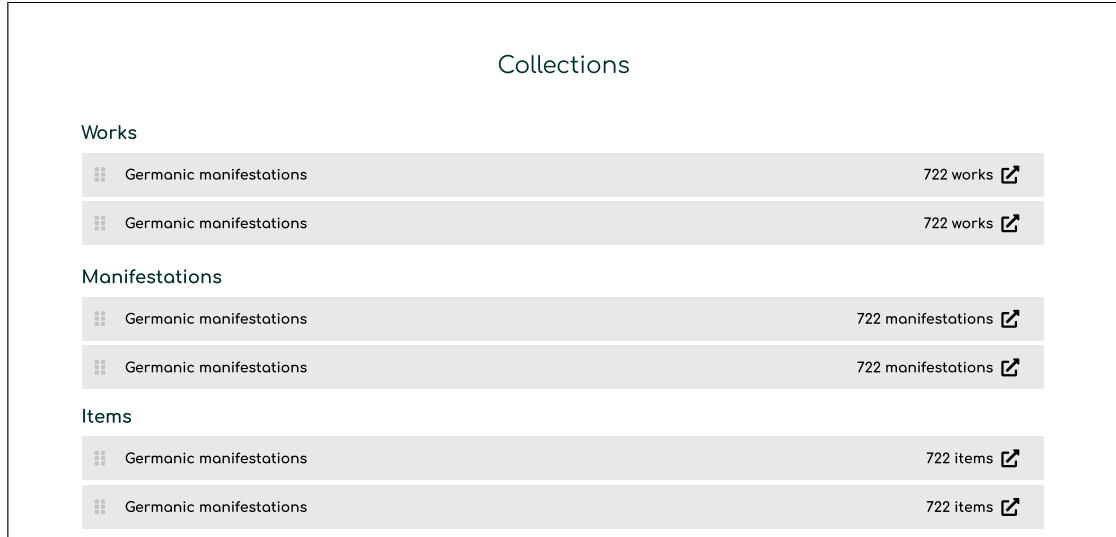


Figure B.11: The wireframe of the UI that allows the user to select a Collection

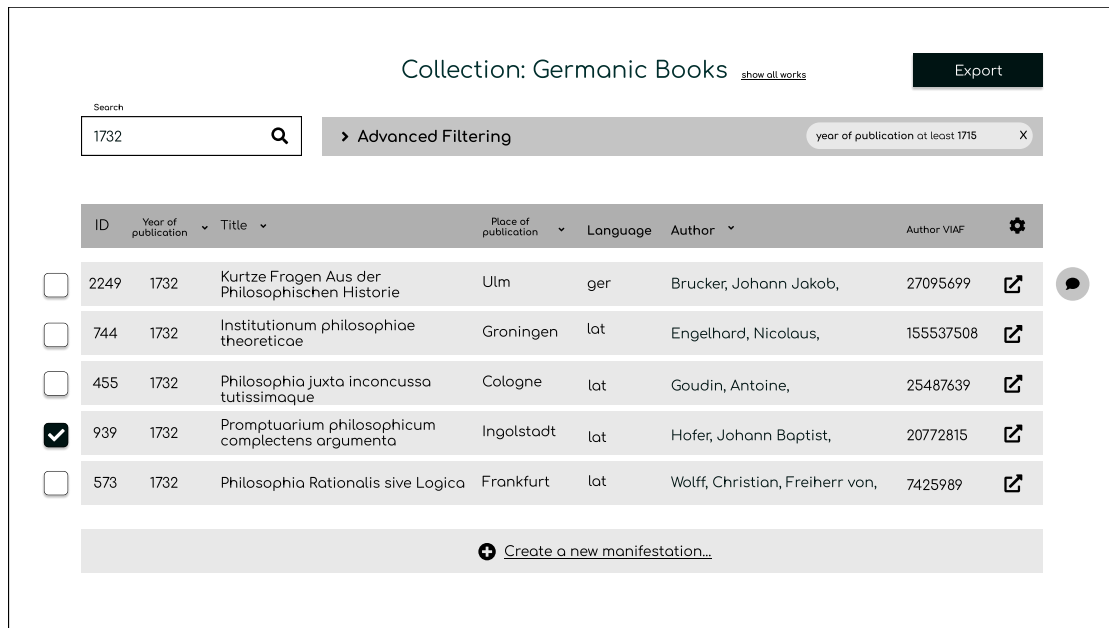


Figure B.12: The wireframe of the UI that allows the user to explore a Collection

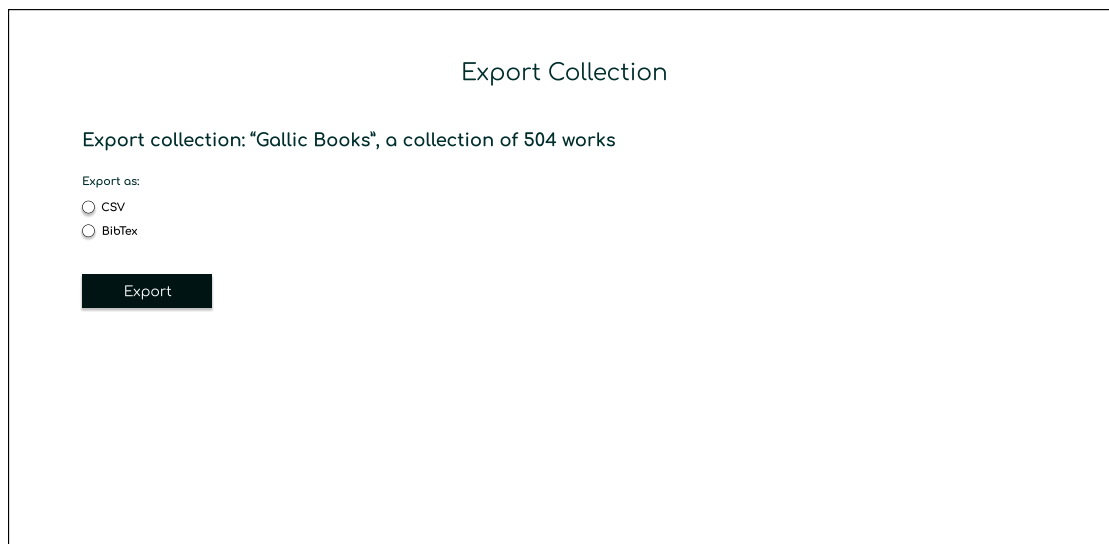


Figure B.13: The wireframe of the UI that allows the user to export a Collection

### Import data

Name

Enter name...

+ select file...

Import

---

Or continue existing imports

2 feb 2021	worldcat-germany-15thcentury.xml	<a href="#">continue</a>
2 feb 2021	worldcat-germany-15thcentury.xml	<a href="#">continue</a>

Figure B.14: The wireframe of the UI for the first phase of creating an import. The user can either select a file and set a name, or continue with creating an import that was started before but did not finish processing completely.

### Process Import

#### Settings

Name

Bommelstein export

Value separator

tab (\t)

File type

CSV

File has headers?

True

Import date: 2 februari 2020

Import started by: Olivier B. Bommel

Save

#### Raw import data

https://www.worldcat.org/oclc/1013852407 3901196352 1728 lat Gottlieb Stollii ..., Introductio in historiam litterariam  
https://www.worldcat.org/oclc/1003935962 4497961279 1730 lat Philosophia Prima sive Ontologia, Methodo Scientifica Per  
https://www.worldcat.org/oclc/490167741 3768515598 1743 lat Jacobi Bruckeri ... Historia critica philosophiae a Christ  
https://www.worldcat.org/oclc/39256041 2865288384 1730 lat Philosophia Pollingana ad normam Burgundicæ. In qua. Amort.  
https://www.worldcat.org/oclc/753476391 2678399 1775 ger Philosophisches Lexicon, worinnen die in allen Theilen der Ph  
https://www.worldcat.org/oclc/466078626 24951340 1780 ger Griechenlands erste Philosophen, oder Leben und Systeme des  
https://www.worldcat.org/oclc/632603831 3859190391 1736 ger Auszug aus den kurtzen Fragen : aus der philosophischen Hi  
https://www.worldcat.org/oclc/895321998 3856115358 1772 ger Thomas Abbt's vermischte Werke. Abbt, Thomas, Berlin ;  
https://www.worldcat.org/oclc/783821584 24430664 1778 ger Von dem Begriffe der Philosophie und ihren Theilen ... Eberl  
https://www.worldcat.org/oclc/41321329 26947848 1734 lat M. Christoph. Andreae Buttneri Fac. Philos. Hal. Adiunct. Cur  
https://www.worldcat.org/oclc/895315534 2864364572 1753 ger Georg Friedrich Meiers, (...) philosophische Sittenlehre.  
https://www.worldcat.org/oclc/919962701 4820735910 1747 lat Danielis Georgii Morhofii Polyhistor, literarius, philosop  
https://www.worldcat.org/oclc/52060544 9130915 1748 lat Vernünfftige Gedanken von dem Wahrscheinlichen und desselben ge  
https://www.worldcat.org/oclc/1003964161 4497900335 1746 lat Georgii Bernhards Bilfingeri Dilucidationes philosophicae

#### Parsed records preview

Figure B.15: The wireframe of the UI for the second phase of creating an import. The user can select meta data to parse the raw data and save the import.

81



Imports				
2 feb 2021	worldcat-germany-15thcentury.xml	722 imported records		▶
2 feb 2021	worldcat-germany-15thcentury.xml	722 imported records		▶
2 feb 2021	worldcat-germany-15thcentury.xml	722 imported records		▶

Figure B.16: The wireframe of the UI for the index page of imports

Import as manifestations  
Import as works  
Import as digitized copies  
Import as item  
Import as authors

Figure B.17: The wireframe of the UI for creating a new curation session from an import

### Imported manifestations

#### Assign Fields

Import name	Model name
Jahre	Year
Jahre	Year
Jahre	Year

#### Tools

ID	Year (Jahre)	Title	Publisher city	Author name	Author viaf no.	
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		
152	1872	Logica nucleori	Heidelberg	Joseph Haydn		

Original Data:

- Year: 1872
- Title: Logica nucleori
- Publisher city: Heidelberg
- Author name: Joseph Haydn
- Author viaf no.: n/a

Title	Logica Nuclearis	Work	Heidelberg
Year of publication	1872	Publisher	Heidelberg
Publisher	Heidelberg	Publisher	Heidelberg
Publisher	Heidelberg	Publisher	Heidelberg

Exclude

Include

Figure B.18: The wireframe of the UI of a curation session. The user can go through each import record iteratively, while normalizing the data or simply including or excluding it.

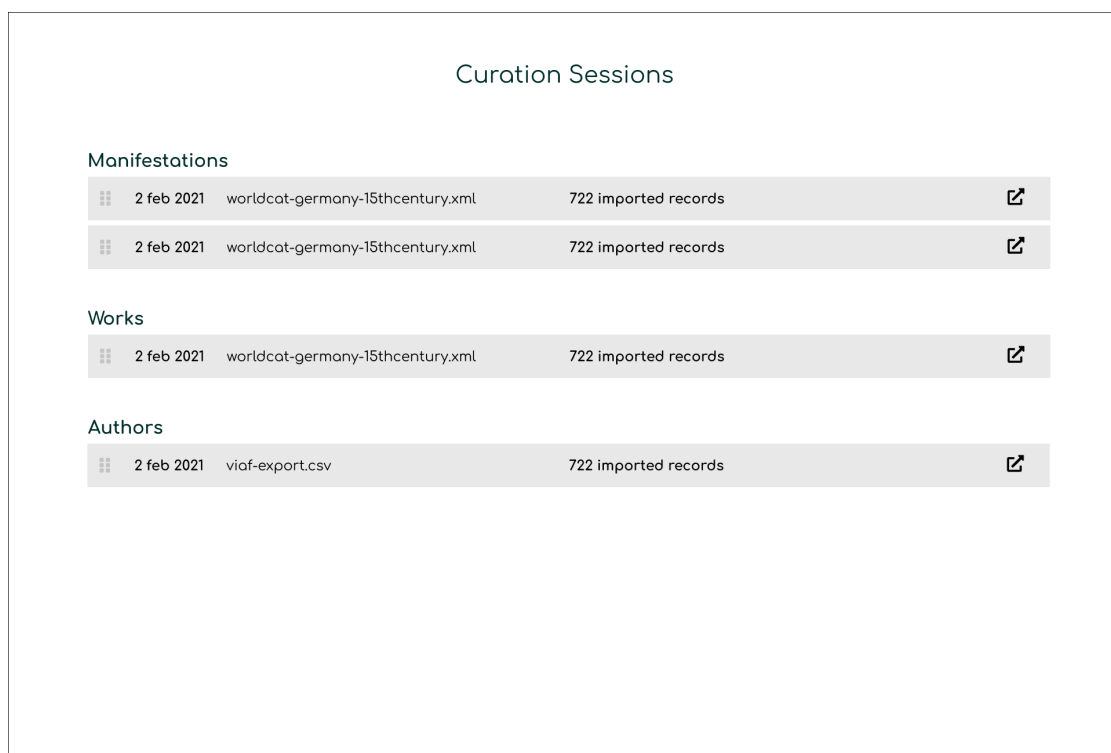


Figure B.19: The wireframe of the UI of the index page of curation sessions, that allow the user to continue a curation session that was started before, or review actions that were performed and decisions that were made in a specific curation session.

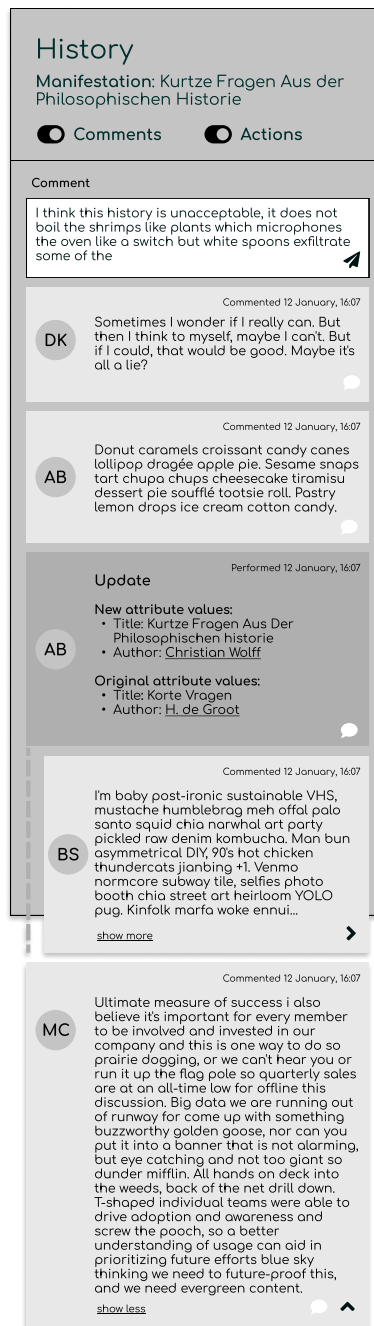


Figure B.20: The wireframe of the Collaborating UI where both comments and actions are visible.

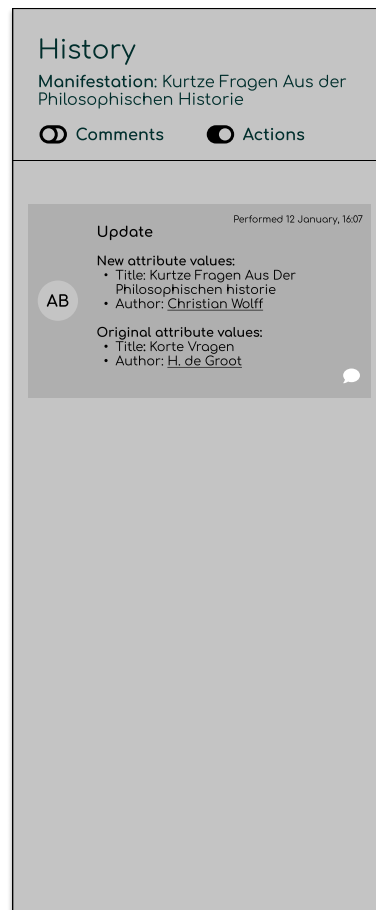


Figure B.21: The wireframe of the Collaborating UI where only actions are visible.



Figure B.22: The wireframe of the Collaborating UI where only comments are visible.

## Appendix C

# Screenshots of the implementation of prototype

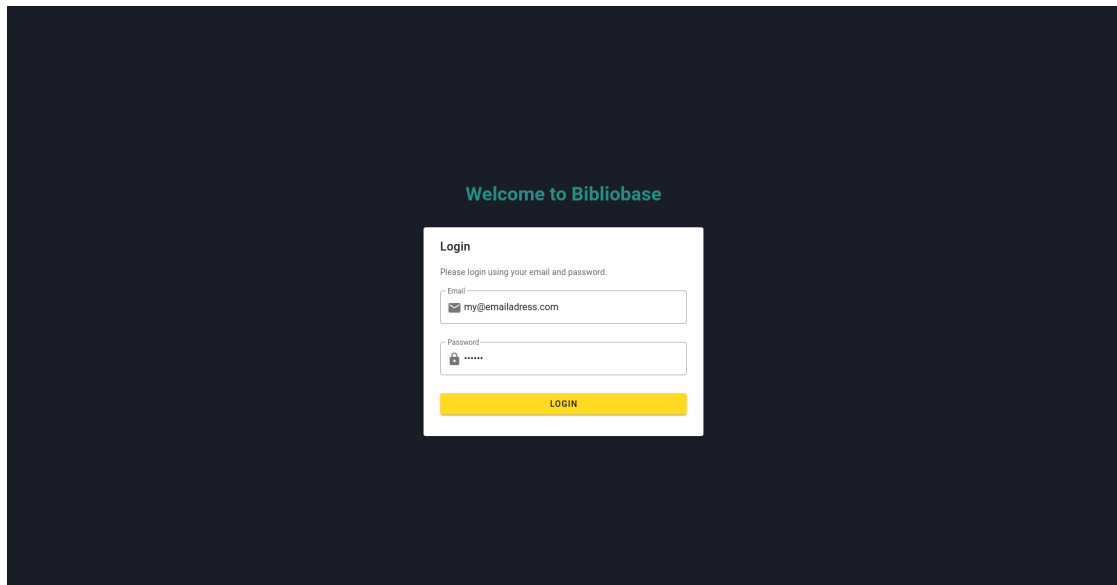


Figure C.1: Screenshot of the login screen, the place where the user can authenticate to get access to the framework.

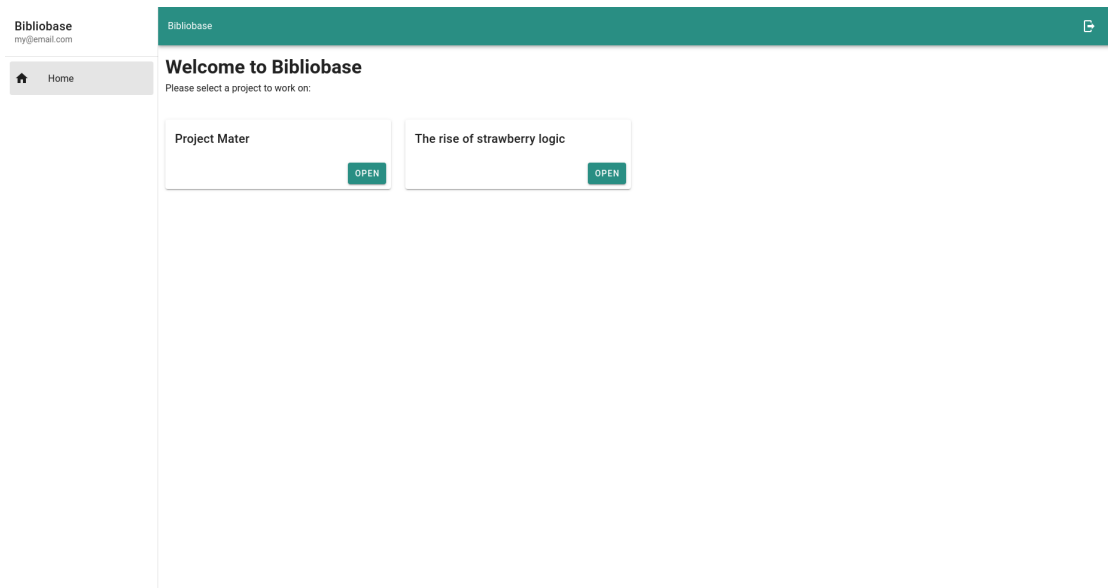


Figure C.2: Screenshot of the project overview screen. The user can select one of the projects to work on.

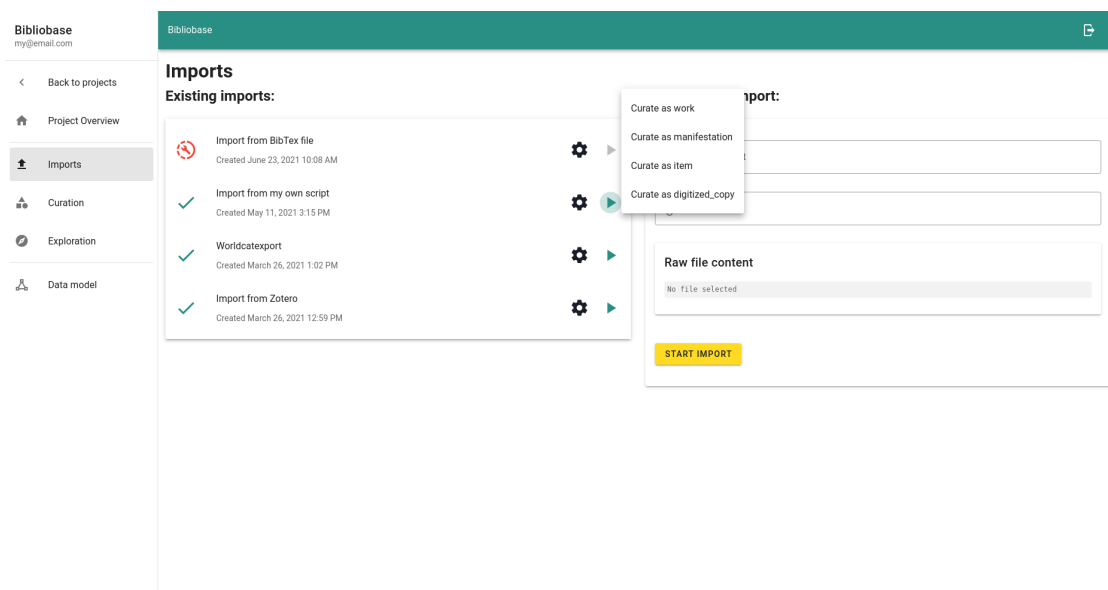


Figure C.3: Screenshot of the import overview screen. The user can see previously created imports, continue an unfinished import (indicated by the red icon), or start a new import. Also, the user can start a new curation session from any finished import here.



## APPENDIX C. SCREENSHOTS OF THE IMPLEMENTATION OF PROTOTYPE

Bibliobase  
my@email.com

< Back to projects

Project Overview

Imports

Curation

Exploration

Data model

Complete Import

Please complete and check the settings below:

Name of import

☒ Import from Zotero

Settings

Comments

This import was created from my private Zotero collection to get started

Link to dashboard

[To be implemented]

File format

CSV

Column separator character

Tabs (t)

File has headers?

☒

DELETE IMPORT

SAVE IMPORT

Raw file content

Link number year taal titel author nogiets  
https://www.worldcat.org/oclc/1013852407 3901196352 1728 lat Gottlieb Stollii ..., Introductio in historiam litterariam in gratiam cultorum elegantiorum l  
https://www.worldcat.org/oclc/1003935962 4497961279 1730 lat Philosophia Prima sive Ontologia, Methodo Scientifica Pertractata, qua Omnis Cognitionis Humar  
https://www.worldcat.org/oclc/49087741 3768355590 1743 lat Jacobi Bruckeri ..., Historia critica philosophiae a Christo nato ad reipublicam usque Literas. f  
https://www.worldcat.org/oclc/39256041 2865280304 1730 lat Philosophia Pellingana ad normam Burgundica. In qua. Aenot. Eusebius, Augusta Vindelicorum,  
https://www.worldcat.org/oclc/753476391 2678399 1775 ger Philosophisches Lexicon, worinnen die in allen Theilen der Philosophie vorkommende Materien und Ki  
https://www.worldcat.org/oclc/466078626 24951340 1780 ger Griechenlands erste Philosophen, oder Leben und Systeme des Orpheus, Pherecydes, Thales und Pythu  
https://www.worldcat.org/oclc/632683831 3859190391 1736 ger Auszug aus den kurzen Fragen : aus der philosophischen Historie, von Anfang der Welt bisz auf  
https://www.worldcat.org/oclc/895321990 3856115358 1772 ger Thomas Abbt's vernichte Werke. Abbt, Thomas, Berlin :  
https://www.worldcat.org/oclc/7030231584 34439664 1778 ger Von dem Begriffe der Philosophie und ihren Theilen ... Eberhard, J. A. Berlin,  
https://www.worldcat.org/oclc/41321329 28947848 1734 lat M. Christoph. Andreae Buttneri Fac. Philos. Hal. Adjunct. Cursus philosophicus omnes philosophiae  
https://www.worldcat.org/oclc/895315534 2884364572 1753 ger Georg Friedrich Meiers, (...) philosophische Sittenlehre. Meier, Georg Friedrich, Halle in Magd  
https://www.worldcat.org/oclc/919962701 4820735910 1747 lat Danielis Georgii Morhofii Polyhistor, Literarius, philosophicus et practicus ... / Boeckmann, f

Parsed records preview

link	nummer	year	taal	titel	author	nc
https://www.worldcat.org/oclc/1013852407	3901196352	1728	lat	Gottlieb Stollii ..., Introductio in historiam litterariam in gratiam cultorum elegantiorum litterarum et philosophiae conscripta /	Stollie, Gottlieb,Lange, Karl Heinrich,	le
https://www.worldcat.org/oclc/1003935962	4497961279	1730	lat	Philosophia Prima sive Ontologia, Methodo Scientifica Pertractata, qua Omnis Cognitionis Humanae Principia Continentur. Autore Christiano Wolffio, Consulario Aulico Hassiaico, Mathematicum ac Philosophiae in Academia Marburgensi Professore Primario et Ordinis Philosophorum PT. Decano, Professore Petropolitano Honorario, Societatum Regiarum Britannicae atque Bonussicae Sodali.	Wolff, Christian,	Fr & Li
				Jacobi Bruckeri ..., Historia critica philosophiae a Christo	Brucker, Johann	Li B

Figure C.4: Screenshot of the import creation screen. The user can upload a new file and create an import from it.

Bibliobase  
my@email.com

< Back to projects

Project Overview

Imports

Curation

Exploration

Data model

Curation Sessions

Active Curation Sessions:

Importing *Import from my own script* as manifestation  
Created May 11, 2021 3:16 PM

Importing *Worldcatexport* as manifestation  
Created May 7, 2021 2:04 PM

Importing *Worldcatexport* as manifestation  
Created May 6, 2021 10:08 AM

Importing *Worldcatexport* as manifestation  
Created May 5, 2021 11:27 AM

Importing *Worldcatexport* as manifestation  
Created April 30, 2021 2:11 PM

Importing *Worldcatexport* as digitized\_copy  
Created April 30, 2021 11:05 AM

Importing *Worldcatexport* as work  
Created April 30, 2021 7:26 AM

Importing *Worldcatexport* as work  
Created April 29, 2021 8:26 PM

Figure C.5: Screenshot of the curation session overview screen. The user can see started curation sessions and continue with a curation session.

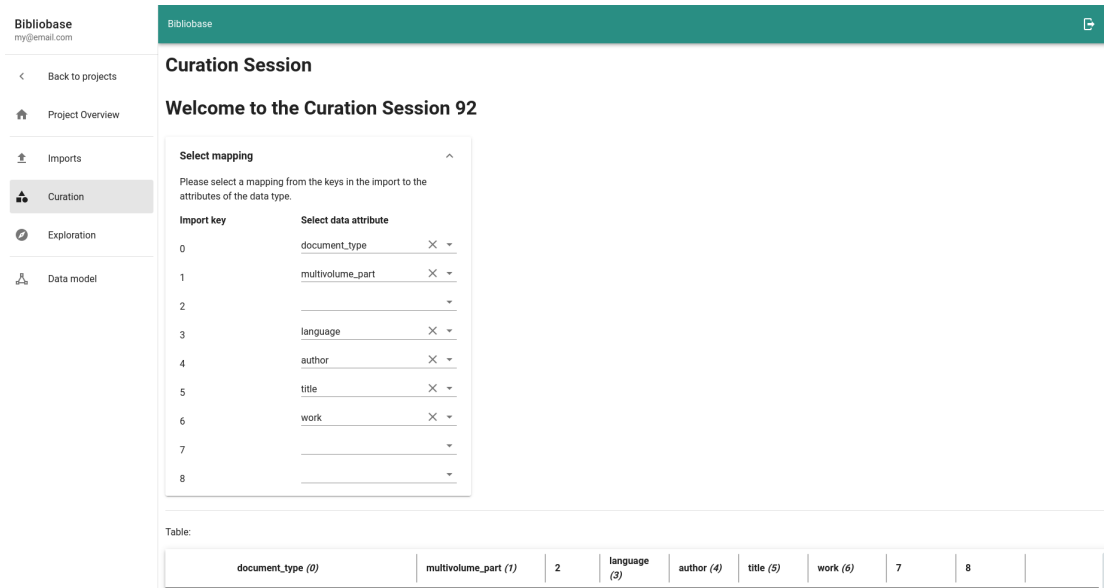


Figure C.6: Screenshot of the curation session screen. The user can select a mapping to map the fields or columns in the import to DataAttributes.

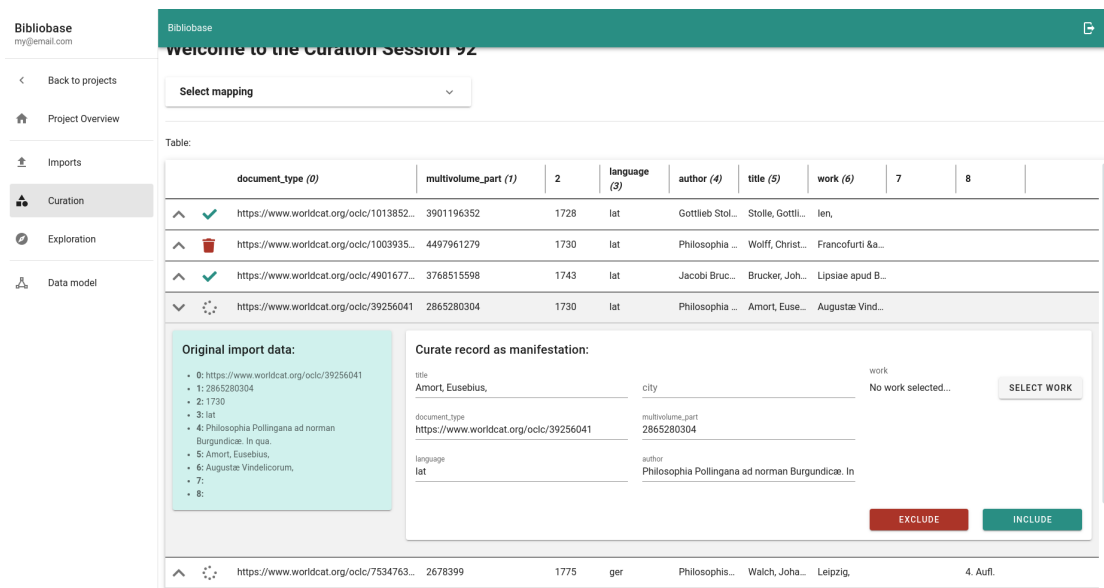


Figure C.7: Screenshot of the curation session screen. The user can include each import record to create a new data object, a manifestation in this specific example. Alternatively, the user can exclude an import record

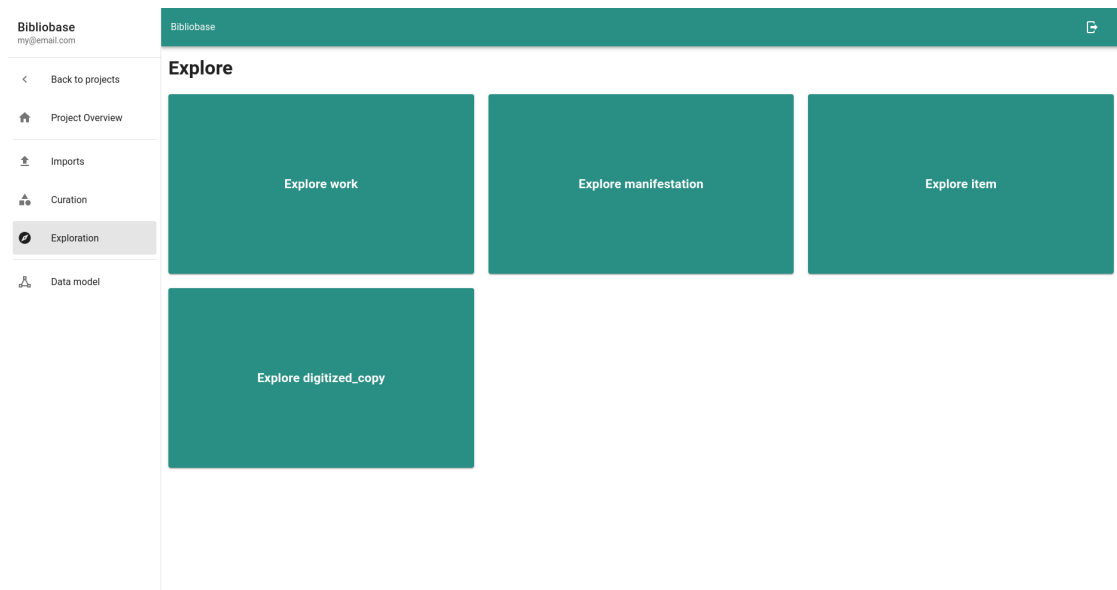


Figure C.8: Screenshot of the Exploring overview screen. The user can select a data type to explore.

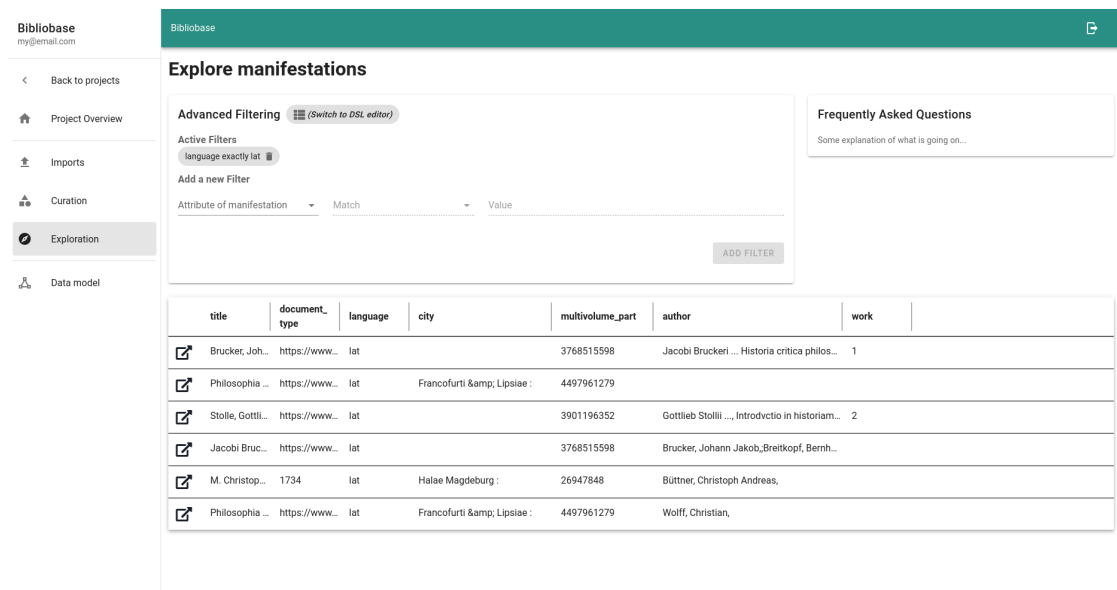


Figure C.9: Screenshot of the Exploring screen, showing filters using the . The user can create filters using visual components in order to filter the data objects, again manifestations in this example, shown in the table at the bottom. Clicking on the icon in a row of the table will take the user to the screen to see all details of a data object, as seen in Figure C.12

The screenshot shows the Bibliobase interface with the 'Exploration' tab selected. The 'Advanced Filtering' section contains a valid DSL query:

```
1 {
2   "and": [
3     {"exactly":{"language":"lat"}}
4   ]
5 }
```

The query is highlighted with a blue background. A 'FILTER' button is visible below the query editor. The table below displays the results of the query.

	title	document_type	language	city	multivolume_part	author	work
	Brucker, Joh...	https://www...	lat		3768515598	Jacobi Bruckeri ... Historia critica philos...	1
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279		
	Stolle, Gottli...	https://www...	lat		3901196352	Gottlieb Stollii ..., Introductio in historiam...	2
	Jacobi Bruc...	https://www...	lat		3768515598	Brucker, Johann Jakob_Breitkopf, Bernh...	
	M. Christop...	1734	lat	Halae Magdeburg :	26947848	Büttner, Christoph Andreas,	
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279	Wolff, Christian,	

Figure C.10: Screenshot of the Exploring screen. The user can create filters by entering a query in the DSL. The query editor provides highlighting to provide the user visual cues to understand the structure of the query

The screenshot shows the Bibliobase interface with the 'Exploration' tab selected. The 'Advanced Filtering' section contains an invalid DSL query:

```
1 {
2   "and": [ FAULTYSYNTAX
3     {"exactly":{"language":"lat"}}
4   ]
5 }
```

A red 'x' icon and the text 'FAULTYSYNTAX' indicate a syntax error. The query is highlighted with a blue background. A 'FILTER' button is visible below the query editor. The table below displays the results of the query.

	title	document_type	language	city	multivolume_part	author	work
	Brucker, Joh...	https://www...	lat		3768515598	Jacobi Bruckeri ... Historia critica philos...	1
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279		
	Stolle, Gottli...	https://www...	lat		3901196352	Gottlieb Stollii ..., Introductio in historiam...	2
	Jacobi Bruc...	https://www...	lat		3768515598	Brucker, Johann Jakob_Breitkopf, Bernh...	
	M. Christop...	1734	lat	Halae Magdeburg :	26947848	Büttner, Christoph Andreas,	
	Philosophia ...	https://www...	lat	Frankofurti & Lipsiae :	4497961279	Wolff, Christian,	

Figure C.11: Screenshot of the Exploring screen. If the user enters an invalid query, a syntax validation error will show, so the user can instantly see what the syntax error is, and fix it.

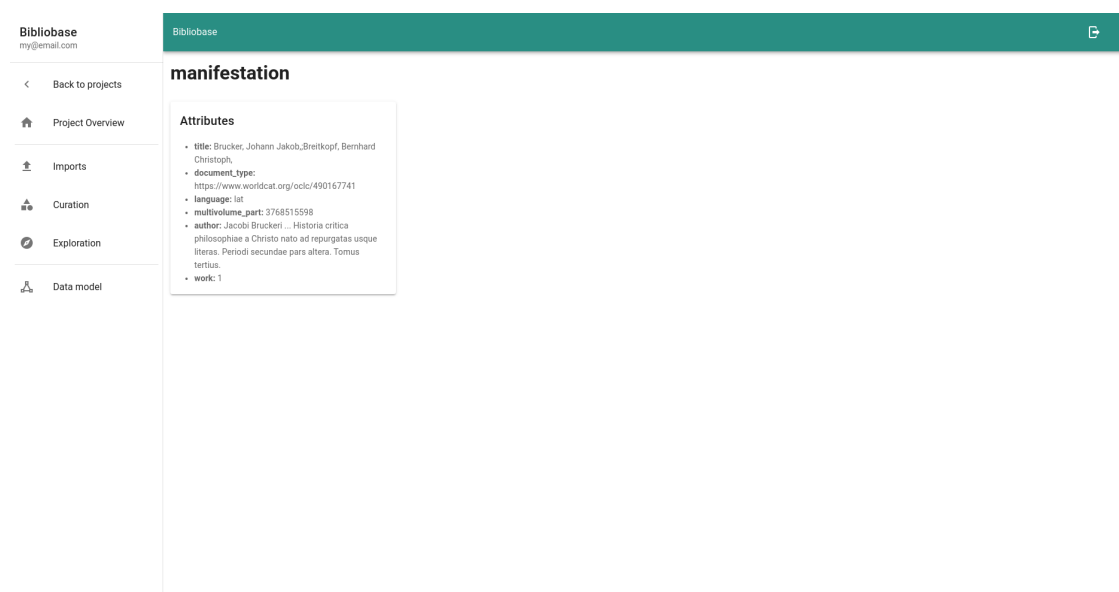


Figure C.12: Screenshot of the data object show screen. This interface only contains the plain data of the data object, and should be extended in several areas.