CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Interactive Ontology Dashboard

Bachelor Thesis
Software Engineering and Technology

Supervisor: Petr Křemen
Author: Avetis Mkrtchian

Prague 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Mkrtchian** Jméno: **Avetis** Osobní číslo: **492731**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Interaktivní ontologický dashboard

Název bakalářské práce anglicky:

Interactive Ontology Dashboard

Pokyny pro vypracování:

Seznam doporučené literatury:

- OWL2 Web Ontology Language Primer, https://www.w3.org/TR/owl2-primer/
- Jackson, R.C., Balhoff, J.P., Douglass, E. et al. ROBOT: A Tool for Automating Ontology Workflows. BMC Bioinformatics 20, 407 (2019). https://doi.org/10.1186/s12859-019-3002-3
- Smith, B., Ashburner, M., Rosse, C. et al. The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. Nat Biotechnol 25, 1251–1255 (2007). https://doi.org/10.1038/nbt1346

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Petr Křemen, Ph.D. skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **24.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **16.02.2025**

Ing. Petr Křemen, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Author statement for undergraduate thesis

I declare that the presented wort was developed independently and that I have listed all sources of information used within it in accordance with the methodical instruction for observing the ethical principles in the preparation of university theses.

Prague, 26 May 2023

Abstract

OBO Foundry contributes to standardizing biomedical terminologies, taxonomies and ontologies. To monitor ontology quality they produce statistics and reports. However, understanding cross-ontology problems, problems of individual ontology concepts, or other more complex queries, the tools are missing. The goal of this work is to create a solution for dashboards over ontology catalogues and test it on the OBO Foundry dashboard. As a prominent test case, we aim at the biomedical terminologies, taxonomies, and ontologies maintained by the OBO Foundry. This involves designing a shared machine-readable representation of ontology quality metrics and quality constraints. Thus, to test the solution, proprietary ontology quality metrics and ontology validation reports delivered by one of OBO Foundry tools - ROBOT - will need to be standardized as a part of the work.

Keywords: RDF, Dashboard, SPARQL, Kibana, Elasticsearch, ROBOT, OBO Foundry, SHACL, DQV

Abstrakt

OBO Foundry přispívá ke standardizaci biomedicínských terminologií, taxonomií a ontologií. Pro sledování kvality ontologií vytváří statistiky a reporty. Pro pochopení problémů napříč ontologiemi, problémů jednotlivých ontologických konceptů nebo jiných složitějších dotazů však nástroje chybí. Cílem této práce je vytvořit řešení pro dashboardy nad katalogy ontologií a otestovat je na dashboardu OBO Foundry. Jako významný testovací případ se zaměřujeme na biomedicínské terminologie, taxonomie a ontologie spravované v OBO Foundry. To zahrnuje návrh sdílené strojově-čitelné reprezentace metrik kvality ontologií a omezení kvality. Pro testování řešení bude tedy třeba v rámci práce standardizovat proprietární metriky kvality ontologií a reporty o validaci ontologií dodávané jedním z nástrojů OBO Foundry - ROBOT.

Klíčová slova: RDF, Dashboard, SPARQL, Kibana, Elasticsearch, ROBOT, OBO Foundry, SHACL, DQV

Contents

1. Introduction

OBO Foundry (Open Biological and Biomedical Ontologies Foundry) is a collaborative effort among scientists and researchers in the life sciences domain. Its goal is to develop and maintain a collection of high-quality ontologies that help organize and represent knowledge in biology and medicine.

The OBO Foundry library refers to a collection of ontologies created by different projects in the life sciences. These ontologies follow certain rules established by the OBO Foundry to ensure they work well together and are of high quality.

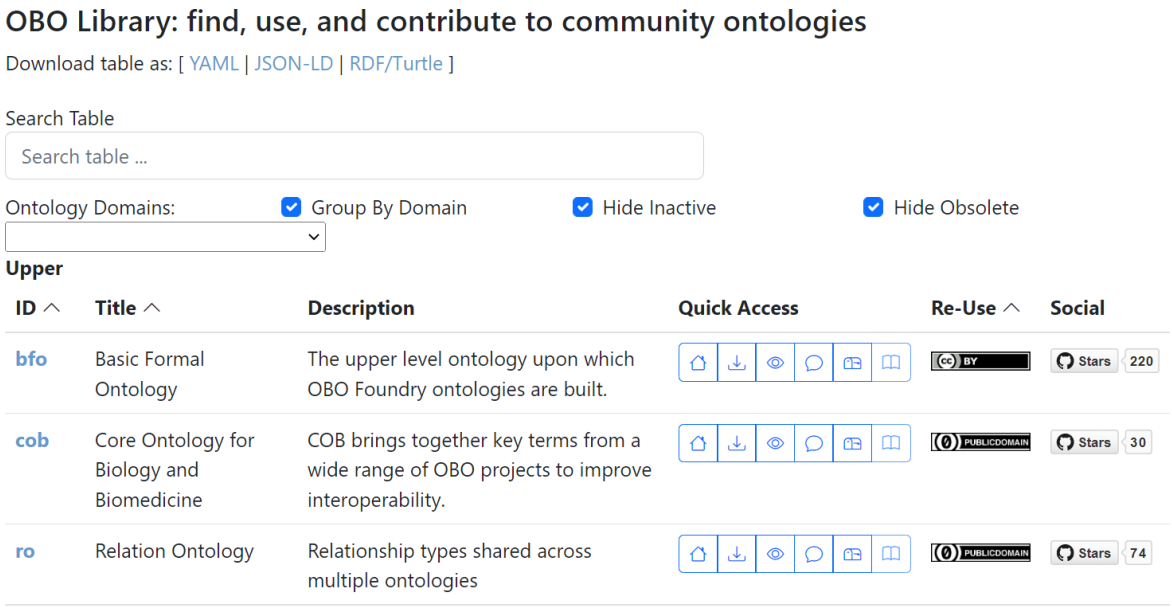

Figure 1.1.: OBO Foundry library

OBO Foundry has its own ROBOT tool [18] for computing number of metrics about an ontology, such as entity and axiom counts, qualitative information and more complex metrics aimed at informing ontology developers, or this tool can report on various issues that may provide problems for users such as classes with multiple labels in the same languages, multiple definitions, missing definitions.

The OBO Foundry also has an OBO Dashboard based on the results of the ROBOT tool, which is a prominent test case for this work. This dashboard checks ontologies for compliance with OBO Foundry principles, providing this information with details of the problems.

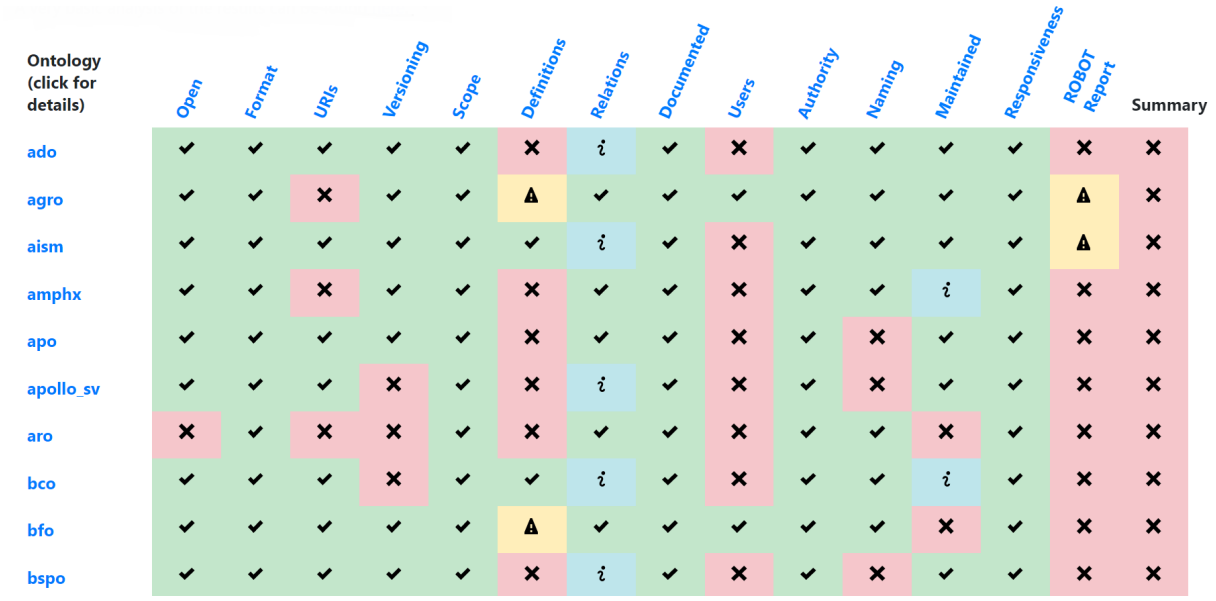

Figure 1.2.: OBO Dashboard

Dashboard solutions over ontologies are proprietary and are generally not built on top of standards. This work aims to design and implement a rich interactive dashboard backed by standardized vocabularies. As a prominent test case, we aim at the biomedical terminologies, taxonomies, and ontologies maintained by the OBO Foundry. Since I was not familiar with ontologies before, one of the first tasks for me was to familiarize myself with the RDF, OWL language, basics of ontologies, OBO Foundry, ROBOT tool. In the following sections, I will describe technologies that I used, explain why I used them. Next I will describe the architecture, you will find out which way I chose to create the dashboard and how it works. At the end, I will describe the implementation and testing.

2. What is an Ontology?

Ontology is a term used in philosophy, computer science, and other fields to refer to the study and representation of knowledge about the world. In computer science it provides a formal and structured way to describe entities, concepts, and the relationships between them in a specific domain of knowledge.

An ontology defines and describes various concepts or entities and their properties. These concepts can be anything from animals and plants to people, places, or even abstract ideas. For each concept, the ontology specifies its characteristics, such as its name, attributes, and relationships with other concepts.

Think of an ontology as a way to organize information in a logical and structured manner. It's like a framework or a set of rules that helps computers understand the meaning and relationships between different concepts. It's a bit like building blocks that fit together to create a bigger picture.

Let's take the example of a library, an ontology for a library might include concepts like books, authors. Each concept would have its own properties and relationships. For instance, a book concept might have properties like title, author, and publication date, while the relationship between a book and an author would indicate that an author can write multiple books.

By defining these concepts, properties, and relationships in ontology, computers can understand and reason about the information more effectively. They can perform tasks like searching for books by a specific author, identifying related genres, or even suggesting similar books based on a user's preferences.

Ontologies are used in various fields, including artificial intelligence, data integration, and knowledge management. They provide a common language for computers and humans to communicate and share information, making it easier to organize, search, and analyze data.

3. Background

In this chapter I will introduce you to the basic technologies, tools, languages, standards I needed to accomplish my assignment.

3.1. Languages and standards

In this section I will present the languages and standards that were used. I will start with RDF which is a semantic web language for graphs, OWL builds ontologies over these graphs, SHACL allows validate these ontologies and DQV is a concrete ontology defined for description of metrics.

3.2. Resource Description Framework

The Resource Description Framework [23] (RDF) is a framework for expressing information about resources. In other words, RDF is a standard way to make statements about resources. Resources can be documents, people, physical objects, and abstract concepts. An example of an RDF statement is:

```
<John> <knows> <Alice>
```

We can visualize this statement as a connected graph.

Figure 3.1.: Informal graph of the simple statement

3.2.1. Components of an RDF Statement (Triple)

RDF is intended for situations in which information on the Web needs to be processed by applications, rather than being only displayed to people. RDF provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning. Since it is a common framework, application designers can leverage the availability of common RDF parsers and processing tools. The ability to exchange information between different applications means that the information may be made available to applications other than those for which it was originally created.

An RDF statement consists of three components, referred to as a triple:

1. **Subject**: It represents the resource or entity being described. The subject is represented by a unique identifier, such as a Uniform Resource Identifier (URI) or a blank node. The subject can be any concept, object, or entity for which information is being conveyed.

2. **Predicate**: Also known as the property or attribute, the predicate describes the specific aspect or characteristic of the subject. It represents a relationship between the subject and the object. Predicates are typically represented by URIs and often come from predefined vocabularies or ontologies.

3. **Object**: It represents the value or target of the predicate. The object can be a literal value, such as a string, number, or date, or it can be another resource identified by a URI or a blank node. The object provides the actual information associated with the subject-predicate relationship.

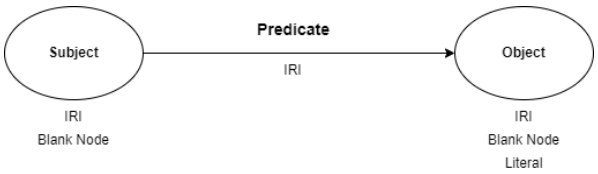

Figure 3.2.: RDF statements consist of a subject, a predicate and an object.

3.2.2. IRIs

International Resource Identifier (IRI) [24] is a protocol standard which builds on Uniform Resource Identifier(URI) [28]. An IRI identifies a resource. As mentioned, IRIs are used to identify resources such as documents, people, physical objects, and abstract concepts. IRIs can appear in all three positions of above mentioned triple. It can be said that IRIs are used as "names" in RDF. They enable the integration and linking of information across different datasets, systems, and domains, forming the foundation for the semantic web.

In RDF, a namespace is a logical grouping of IRIs that share a common prefix. By defining a prefix for a namespace, we can use that prefix as a shorthand notation for the corresponding IRIs within an RDF document or query. But prefixes are not standardized in anyway, anyone can pick any in his/her applications.What is quite commonly used however is `https://prefix.cc/` as a helpful service for reusing prefixes.

For example, here is the following RDF triple:

```
<http://example.org/books/book1>
    <http://purl.org/dc/elements/1.1/title> "War and Peace" .
```

We have book represented by `<http://example.org/books/book1>` and title represented by `<http://purl.org/dc/elements/1.1/title>`. Using prefixes "dc:" for `<http://purl.org/dc/elements/1.1/>` and "ex:" for `<http://example.org/books/>` we can improve readability and obtain the following result:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ex: <http://example.org/books/> .

ex:book1 dc:title "War and Peace" .
```

3.2.3. Literals

Literals are basic values that are not IRIs. Literals are used for values such as strings, numbers, and dates.

In RDF, literals are composed of three components:

- **Value**: The actual data value being represented. It can be a string, number, date, boolean, or any other data type.
- **Datatype**: Specifies the type of the literal value. RDF provides a range of predefined datatypes, such as xsd:string, xsd:integer, xsd:boolean, xsd:date, etc. The datatype helps in interpreting and processing the literal value correctly.
- **Language tag** (optional): When representing textual values, a language tag can be added to indicate the language of the literal value. Language tags follow the syntax defined in the BCP 47 standard [21], such as en for English, fr for French, de for German, etc. The language tag allows for multilingual support in RDF data.

  Here are a few examples of RDF literals:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

"Hello, World!"^^xsd:string
"77"^^xsd:integer
"true"^^xsd:boolean
"2023-05-07"^^xsd:date
```

3.2.4. Blank Nodes

Blank node is node in a RDF graph representing a resource for which a URI or literal is not given. Blank nodes are like simple variables in algebra; they represent some thing without saying what their value is. Blank Node, also known as an anonymous node or a bnode, is a type of node that represents a resource without a specific identifier or URI. Blank Nodes are local to a specific RDF graph and are used to denote anonymous or unidentifiable resources. Blank nodes can appear in the subject and object position of a triple.

Here is an example of Blank Node visualization:

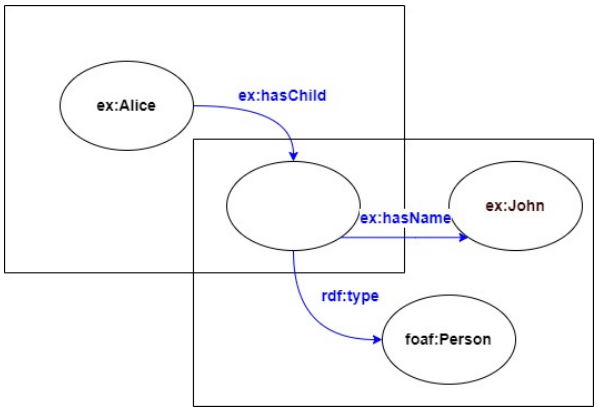

Figure 3.3.: Example of a blank node in a RDF graph

It can also be represented as an RDF triple:

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/>

ex:Alice ex:hasChild _:bnode1 .

_:bnode1 rdf:type foaf:Person .
_:bnode1 ex:hasName ex:John .
```

3.3. Web Ontology Language

Web Ontology Language (OWL) is a language to be used in the Semantic Web, so names in OWL are international resource identifiers (IRIs). As IRIs are long, we will often make use of abbreviation mechanisms for writing them in OWL. The way in which such abbreviations work is specific to each syntactic format that can be used to encode OWL ontologies, but the examples in this document can generally be understood without knowing these details.[17]

There are various syntaxes available for OWL which serve various purposes. The most commonly used is RDF/XML syntax.

OWL 2 is a supercharged version of OWL, which is a special language that helps computers understand and describe things in a detailed and organized way. OWL 2 brings even more capabilities and features to the table, making it even more powerful for creating and working with ontologies.

With OWL 2, you can create more complex and sophisticated ontologies that represent knowledge in a richer and more expressive manner. It provides additional features like more advanced relationships, constraints, and rules that allow for even deeper and more precise descriptions of concepts and their relationships.

You can not only define basic concepts like "car," "engine," and "wheel," but also specify more complex relationships as "hasModel," "hasPart." You can define constraints to ensure that certain conditions are met, such as specifying that a car must have at least four wheels.

OWL 2 also supports more advanced reasoning capabilities, allowing computers to infer new knowledge based on the ontological descriptions. It can help answer complex queries, make intelligent suggestions etc.

OWL 2 is declarative, i.e. it describes a state of affairs in a logical way. Appropriate tools can then be used to infer further information about that state of affairs. How these inferences are realized algorithmically is not part of the OWL document but depends on the specific implementations.

OWL 2 provides more modularization options, allowing ontologies to be organized into smaller, reusable modules. This makes it easier to manage large-scale ontologies and promotes collaboration and interoperability among different ontologies.

In the context of OWL, axioms and entity expressions are key concepts used to describe relationships and constraints within an ontology:

- **Axioms:** the basic statements that an OWL ontology expresses
- **Entities:** elements used to refer to real-world objects
- **Expressions:** combinations of entities to form complex descriptions from basic ones

I suggest looking at the following example from ontology Agronomy Ontology [7]:

```
obo:FOODON_00001173 a owl:Class ;
  rdfs:subClassOf obo:FOODON_03460177
  rdfs:label "plant seed food product"@en .

obo:FOODON_00001172 a owl:Class ;
  rdfs:subClassOf obo:FOODON_00001262, obo:FOODON_03460177
  rdfs:label "nut food product"@en .

obo:FOODON_03460177 a owl:Class ;
  owl:equivalentClass [
    a owl:Class ;
    owl:unionOf (
     obo:FOODON_00001172
     obo:FOODON_00001173
   )
  ] ;
  rdfs:subClassOf obo:FOODON_00001015 ;
  rdfs:label "plant seed or nut food product"@en .
```

Three OWL classes can be seen above. Each owl class is **entity** and has IRI (e.g obo:FOODON_00001173), rdfs:subClassOf and rdfs:label.

```
obo:FOODON_00001173 a owl:Class ;
  rdfs:subClassOf obo:FOODON_03460177
```

This class **axiom** declares a subclass relation between two OWL classes that are described through their names (obo:FOODON_00001173 and obo:FOODON_03460177).

```
obo:FOODON_03460177 a owl:Class ;
  owl:equivalentClass [
    a owl:Class ;
    owl:unionOf (
     obo:FOODON_00001172
     obo:FOODON_00001173
  )
  ] ;
  rdfs:subClassOf obo:FOODON_00001015 ;
  rdfs:label "plant seed or nut food product"@en .
```

This class defined by **expression**: class is equivalent to union of obo:FOODON_00001172 and obo:FOODON_00001173 classes. It can also be noticed by the labels.

3.4. SPARQL Protocol and RDF Query Language

SPARQL [22] is a set of standards for graph databases published by the W3C, but the name is most often used to refer to the query language.

As a query language, SPARQL can be used to add, remove and retrieve data from RDF-style graph databases. SPARQL queries can not only match patterns of subject-predicate-object

triples, but can also use mathematical operations and a wide range of utility functions to create filters and new variable bindings. They can test for the absence of a pattern (negation), contain optional sections and even entire sub-queries. The results can be freely ordered, grouped and those groups can be aggregated over.

The SPARQL language specifies four different query variations for different purposes.[**?**]

- **SELECT** query: used to extract raw values from a SPARQL endpoint, the results are returned in a table format.
- **CONSTRUCT** query: used to extract information from the SPARQL endpoint and transform the results into valid RDF.
- **ASK** query: used to provide a simple True/False result for a query on a SPARQL endpoint.
- **DESCRIBE** query: used to extract an RDF graph from the SPARQL endpoint, the contents of which is left to the endpoint to decide based on what the maintainer deems as useful information

Here is an example of RDF data:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>

<http://example.org/john>
    a foaf:Person ;
    foaf:name "John" ;
    foaf:age 30 ;
    foaf:city "New York" .

<http://example.org/alice>
    a foaf:Person ;
    foaf:name "Alice" ;
    foaf:age 25 ;
    foaf:city "London" .

<http://example.org/sarah>
    a foaf:Person ;
    foaf:name "Sarah" ;
    foaf:age 35 ;
    foaf:city "Paris" .
```

SPARQL query to retrieve only John and Alice:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person ?name ?age ?city
WHERE {
    ?person a foaf:Person ;
        foaf:name ?name ;
        foaf:age ?age ;
        foaf:city ?city .
    FILTER (?name = "John" || ?name = "Alice")
}
```

Executing this query on the RDF dataset would yield the following result:

person	name	age	city
<http://example.org/john>	John	30	New York
<http://example.org/alice>	Alice	25	London

3.5. RDF validation

RDF validation plays a crucial role. By validating RDF data, we can verify if it conforms to a set of predefined criteria, which may include syntactic, structural, and semantic constraints. This process helps identify and resolve issues, errors, or inconsistencies in the RDF data. In this section we will discuss two popular validation languages used in the context of RDF data validation. I will describe the pros and cons of each and tell which language I have come to use.

3.5.1. ShEx

Shape Expressions (ShEx) is a language for describing RDF graph structures. A ShEx schema prescribes conditions that RDF data graphs must meet in order to be considered "conformant". In the ShEx model, a shape map specifies which nodes in an RDF graph will be tested against a ShEx schema. ShEx schemas are intended for use in validating instance data, communicating interface parameters and data structures, generating user interfaces, and transforming RDF graphs into other data formats and structures.[27]

A ShEx schema is built on node constraints and triple constraints that define what it means for a given RDF data graph to conform. In the ShEx model, a given RDF data graph is tested against a ShEx schema to yield a validation result. In the validation process, each node in the RDF data is treated, in turn, as a focus node, and triples involving that node are tested against a triple constraint which, in turn, includes the node constraint IRI. This validation process is controlled by a shape map that specifies how the constructs of a ShEx schema relate to the components of RDF data graphs. There are many ways to populate a shape map with nodes to be validated: through queries, APIs, protocols, or simple enumeration.[27]

ShEx may be serialized using any of three interchangeable concrete syntaxes: Shape Expressions Compact Syntax or ShExC, a compact syntax meant for human eyes and fingers; ShExJ, a JSON-LD syntax meant for machine processing; and ShExR, the RDF interpretation of ShExJ expressed in RDF Turtle syntax.[27]

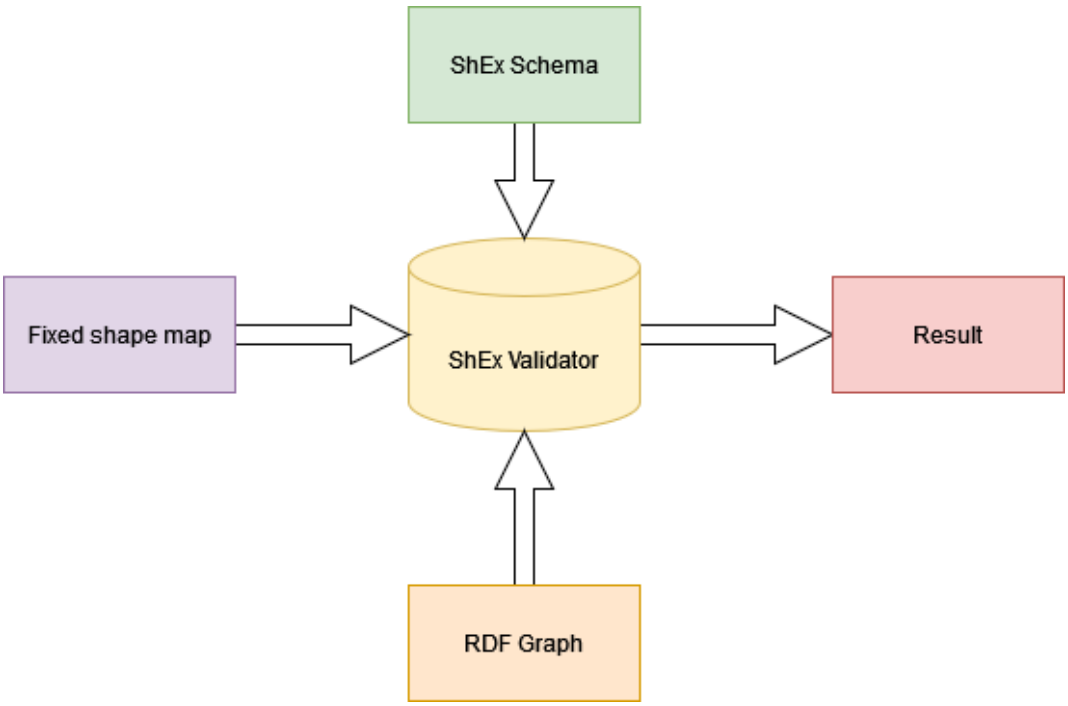

Figure 3.4.: ShEx Validation

3.5.2. SHACL

Shapes Constraint Language (SHACL) [2] is a W3C standard for validating the contents of an RDF-style graph database. It also describes what validation results should be returned so that the user gets a meaningful warning. The syntax of SHACL is typically represented using the Turtle syntax.

SHACL rules are written in RDF and are called "shapes graph". The RDF graphs being validated against such a "shapes graph" are called "data graph". The validation process takes a "data graph" and a "shapes graph" as input to produce a validation report.

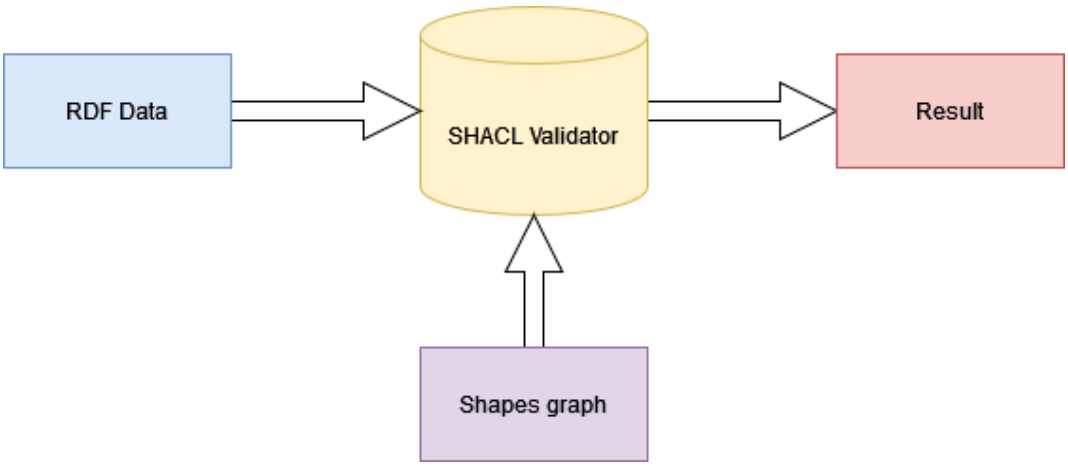

Figure 3.5.: SHACL Validation

The following graph represents an example of shape:

Here's a simple example of SHACL validation with a data graph, shapes graph, and the corresponding output:

Data graph:

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:John a foaf:Person ;
  foaf:name "John" ;
  foaf:age "14"^^xsd:integer;
  foaf:gender "male" .

ex:Alice a foaf:Person ;
  foaf:name "Alice" ;
  foaf:age "30" ;
  foaf:gender "female" .
```

Shapes graph:

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .

ex:PersonShape
  a sh:NodeShape ;
  sh:targetClass foaf:Person ;
  sh:property [
    sh:path foaf:name ;
    sh:minLength 3 ;
  ] ;
  sh:property [
    sh:path foaf:age ;
    sh:datatype xsd:integer ;
    sh:minInclusive 18 ;
  ] ;
  sh:property [
    sh:path foaf:gender ;
    sh:in ("male" "female") ;
  ] .
```

The ex:PersonShape shape defines three properties with corresponding constraints:

- **foaf:name**: Minimum length of 3 characters.
- **foaf:age**: Must be of datatype xsd:integer and have a minimum value of 18.
- **foaf:gender**: Must have a value that is either "male" or "female".

Output:

```
[
    a sh:ValidationResult ;
    sh:resultSeverity sh:Violation ;
    sh:sourceConstraintComponent sh:MinInclusiveConstraintComponent;
    sh:sourceShape _:n1538 ;
    sh:focusNode ex:John ;
    sh:value 14 ;
    sh:resultPath foaf:age ;
    sh:resultMessage "Value is not >= 18" ;
] .
[
    a sh:ValidationResult ;
    sh:resultSeverity sh:Violation ;
    sh:sourceConstraintComponent sh:DatatypeConstraintComponent ;
    sh:sourceShape _:n1538 ;
    sh:focusNode ex:Alice ;
    sh:value "30" ;
    sh:resultPath foaf:age ;
    sh:resultMessage "Value does not have datatype xsd:integer" ;
] .
```

3.5.3. SHACL vs ShEx

While they share a common goal of validating RDF data, they have some differences in terms of syntax, features, and community adoption. Now that I have described SHACL and ShEx above, I would like to summarize by comparing the two languages on the points that were important to me in choosing one of these languages.

1. **Syntax**: SHACL uses an RDF-based syntax, represented using Turtle with SHACL vocabulary. ShEx, on the other hand, primarily uses Shape Expressions Compact Syntax (ShExC) and Shape Expressions JSON (ShExJ) as its syntax formats.

2. **Expressivity**: SHACL has rich set of built-in constraints and features. It supports value constraints, logical conditions, shape composition etc. ShEx focused on simplicity, has more compact syntax and a more focused set of validation features.

3. **Validation Approach**: SHACL validation is typically performed by validating the entire graph against the defined shapes, producing a comprehensive validation report. ShEx validation, on the other hand, is often based on a "focus node" approach, where validation starts from a specific node and traverses the graph based on shape expressions. SHACL has "zero to many" default cardinality, while ShEx has "one to one".

4. **Library support**: Both ShEx and SHACL have library support, but SHACL has broader support and integration within established RDF processing libraries like RDF4J and Apache Jena. These libraries provide extensive features for SHACL validation and are well-maintained by active communities.

In summary, SHACL and ShEx are both powerful tools for validating RDF graphs, but my choice is SHACL. Comparing by syntax, I find it easier to use the regular Turtle in SHACL to write rules.As for the validation approach, as for me SHACL would be more convenient and profitable, besides SHACL is supported by a lot more libraries than ShEx.

3.6. The Data Quality Vocabulary

The Data Quality Vocabulary (DQV) provides a metadata model for expressing data quality. Aiming to facilitate the publication of such data quality information on the Web, especially in the growing area of data catalogues, the W3C Data Web Best Practices Working (DWBP) group has developed the DQV.

DQV [5] is a (meta)data model implemented as an RDF vocabulary which extends the Data Catalog Vocabulary (DCAT) with properties and classes suitable for expressing the quality of datasets and their distributions. DQV has been conceived as a high-level, interoperable framework that must accommodate various views over data quality. DQV does not seek to determine what "quality" means.

The namespace for DQV is `"http://www.w3.org/ns/dqv"`. DQV, however, seeks to re-use elements from other vocabularies, notably DCAT[6].

DQV defines quality measures as specific instances of Quality Measurements, adapting the daQ quality framework DaQ [11], DaQ-RDFCUBE [12].

The following example shows how DQV vocabulary can be used:

```
:myDatasetDistribution a dcat:Distribution ;
    dqv:hasQualityMeasurement :measurement1 .

:measurement1
    a dqv:QualityMeasurement ;
    dqv:computedOn :myDatasetDistribution ;
    dqv:isMeasurementOf :downloadURLAvailabilityMetric ;
    dqv:value "true"^^xsd:boolean .
```

The quality of a given **dcat:Distribution** is assessed via a number of observed properties. One of the properties is **dqv:QualityMeasurement** which represents a metric value providing quantitative or qualitative information about the dataset or distribution. In this example we can see that **:myDatasetDistribution** has one measurement.

:measurement1 has following properties:

1. **dqv:computedOn** refers to the resource (instance of **dcat:Distribution**) on which the quality measurement is performed.
2. **dqv:value** refers to values computed by metric.
3. **dqv:isMeasurementOf** indicates the metric being observed (example below).

```
#definition of dimensions and metrics
:availability
    a dqv:Dimension ;
    skos:prefLabel "Availability"@en ;
    skos:definition "Availability of a dataset is the extent to
    which data (or som portion of it) is present, obtainable and
    ready for use."@en ;
    dqv:inCategory :accessibility .
```

```
:downloadURLAvailabilityMetric
    a dqv:Metric ;
    skos:definition "It checks if dcat:downloadURL is available and
    if its value is dereferenceable."@en ;
    dqv:expectedDataType xsd:boolean ;
    dqv:inDimension :availability .
```

Detailed definitions:

1. **dqv:Metric** gives a procedure for measuring a data quality dimension, which is abstract, by observing a concrete quality indicator. There are usually multiple metrics per dimension; e.g., availability can be indicated by the accessibility of a SPARQL endpoint, or that of an RDF dump. The value of a metric can be numeric (e.g., for the metric "human-readable labeling of classes, properties and entities", the percentage of entities having an rdfs:label or rdfs:comment) or boolean (e.g., whether or not a SPARQL endpoint is accessible).

2. **dqv:inDimension** represents the dimensions a quality metric, certificate and annotation allow a measurement of.

3. **dqv:Dimension** represents criteria relevant for assessing quality. Each quality dimension must have one or more metric to measure it. A dimension is linked with a category using the **dqv:inCategory** property.

4. **dqv:inCategory** represents the category a dimension is grouped in.

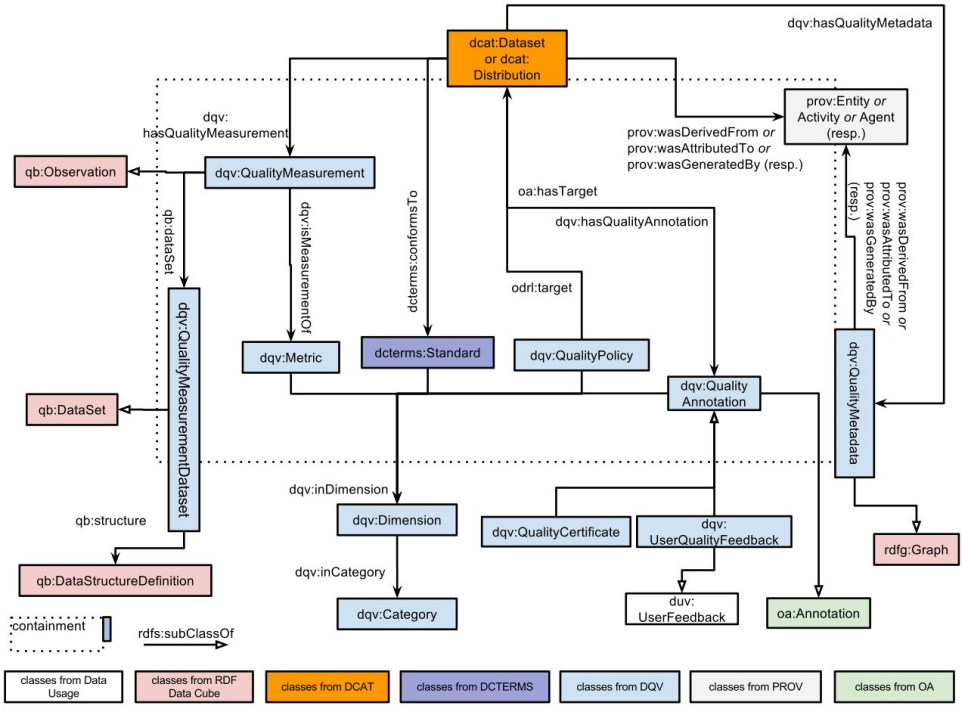

Figure 3.6.: Data model of DQV showing the main relevant classes and their relations.

3.7. Tools for ontology processing

In this section, we will discuss frameworks, databases, libraries and tools used by me to work with ontologies and data in RDF format. I will describe why I used them and how they work.

3.7.1. ROBOT

ROBOT [18] is an open source library and command-line tool for automating ontology development tasks. The library can be called from any programming language that runs on the Java Virtual Machine (JVM). ROBOT provides ontology processing commands for a variety of tasks, including commands for converting formats, running a reasoner, creating import modules, running reports, and various other tasks.

ROBOT supports automation of a wide range of ontology development tasks, focusing on OBO conventions. It packages common high-level ontology development functionality into a convenient library, and makes it easy to configure, combine, and execute individual tasks in comprehensive, automated workflows. This helps ontology developers to efficiently create, maintain, and release high-quality ontologies, so that they can spend more time focusing on development tasks. It also helps guarantee that released ontologies are free of certain types of logical errors and conform to standard quality control checks, increasing the overall robustness and efficiency of the ontology development lifecycle.

It can be said with certainty that ROBOT at the beginning of this work was the key component on which this work was based. The two things I was most interested in in the robot were quality metrics and violations of ontologies.

The quality metrics and violations can be computed using ROBOT by two commands: **report** and **measure**.

The **report** command runs a series of quality control SPARQL queries over the input ontology and generates a TSV or YAML report file based on the results. Each query has a logging level to define the severity of the issue: ERROR, WARN, or INFO.

Level	**Rule Name**	**Subject**	**Property**	**Value**
ERROR	duplicate_label	CARO:0000006	rdfs:label	material anatomical entity
WARN	invalid_xref	PLANA:0000459	oboInOwl: hasDbXref	C90609
INFO	lowercase_definition	PLANA:0003100	IAO:0000115	cell which is part of the testis

Table 3.1.: Example of output lines of report ROBOT command for Planaria ontology

By the **measure** command ROBOT can compute a number of metrics about a ontology. This will generate a table with metrics such as:

- Entity metrics (number of classes, object properties etc.)
- Datatypes used
- Number of axioms (logical and otherwise)
- TBox, RBox, ABox size (size as number of axioms)
- OWL2 profile information

Metric	Metric value	Metric type
axiom_count	13779	single_value
datatype_count	5	single_value
abox_axiom_count	19	single_value

Table 3.2.: Example of output lines of measure ROBOT command for Planaria ontology

3.7.2. RDF4J

RDF4J [13] is framework for working with RDF data. It provides a comprehensive set of APIs and tools for managing, querying, and manipulating RDF graphs. RDF4J offers support for storing and persisting RDF data.

RDF4J also includes features for reasoning, allowing developers to apply semantic inferencing and logical deductions to derive additional knowledge from the RDF data. It supports popular RDF serialization formats like RDF/XML, Turtle, N-Triples, JSON-LD, and more, providing flexibility in working with different RDF data representations.

Additionally, RDF4J offers tools such as a web-based workbench and a command-line console, which provide user-friendly interfaces for interacting with RDF data and executing SPARQL queries against RDF data.

Being a Java-based framework, RDF4J can be integrated into Java applications. It is widely used in various domains for building semantic web applications, knowledge graphs, and linked data systems. It provides a powerful and flexible query API, enabling to retrieve specific data patterns and perform advanced querying operations on RDF graphs.

3.7.3. Apache Jena

Apache Jena [3] is a Java-based open-source framework for building semantic web and linked data applications. It provides a comprehensive set of tools, APIs, and libraries for working with RDF (Resource Description Framework) data and semantic technologies.

Apache Jena offers support for various RDF-related tasks, including parsing and serialization of RDF data, creating and manipulating RDF models, executing SPARQL queries, performing RDF reasoning using RDFS and OWL, and managing RDF datasets. It provides a rich set of APIs and tools that enable developers to work with RDF data.

While Apache Jena is primarily focused on Java, it also offers some web-based components and functionalities, such as a SPARQL endpoint for remote access to RDF data over the web. The core features and capabilities of Apache Jena are centered around Java programming and integration with Java applications.

3.7.4. GraphDB

GraphDB [1] is a high-performance semantic graph database developed by Ontotext, a leading provider of semantic technology solutions. GraphDB is designed to store, manage, and query large-scale RDF datasets, enabling organizations to build knowledge graphs and semantic applications.

GraphDB provides efficient storage mechanisms for RDF data, allowing organizations to handle large and rapidly growing datasets. It supports horizontal scalability, enabling distributed storage across multiple nodes to accommodate the needs of big data applications.

In this case, GraphDB is one of the best solutions for storing large amounts of data, because of having integration and support for RDF4J and Apache Jena, which I mentioned above, allowing interaction with GraphDB using APIs.

3.8. Existing dashboard solutions

In this section, I would like to discuss three dashboard solutions that seemed to me the most interesting and are among the most popular. I would like to mention that for this work I chose between non-commercial dashboards.

3.8.1. Kibana

Kibana[25] is a visual interface tool that allows you to explore, visualize, and build a dashboard over the log data massed in Elasticsearch Clusters. Elastic is the company behind Kibana and the two other open source tools - Elasticsearch and Logstash. The Elasticsearch tool serves as the database for document-oriented and semi-structured data. Logstash supports to collect, parse, and store logs for future use. These three tools can work well together and popularly known as ELK Stack or Elastic Stack.

Architecture: Should be configured to run against an Elasticsearch node. Best suited for logs analysis.

Interface: Web based.

Visualization: Best suited to analyze Logs. Supports Text Based Analysis. Has panels, each panel can have different Data sources.

Supported data sources: Supports only Elastic Search. Exists EEA RDF Indexer for ElasticSearch allows to harvest metadata from SPARQL endpoints or plain RDF files into ElasticSearch.

Querying: Uses Elasticsearch to query data.

3.8.1.1. RDF indexer for Elastisearch

RDF indexer for Elasticsearch [29] is a tool that allows to harvest metadata from SPARQL endpoints into ElasticSearch. The RDF indexer query a RDF data from a endpoint and transforms the extracted RDF triples into a format suitable for indexing in Elasticsearch. This involves mapping the RDF properties and values to Elasticsearch's data model, such as defining index schemas, creating fields, and handling data types. The transformed RDF data is indexed into Elasticsearch, creating an index that organizes the RDF information in a structured manner. The indexer maps the RDF triples to Elasticsearch documents and stores them in the index, making them searchable and analyzable.

The Indexer serves as a bridge between RDF data and Elasticsearch, providing an efficient way to store, index, and search RDF information using Elasticsearch's search and analytics capabilities.

3.8.2. Graphana

Grafana [16] is an open source interactive data-visualization platform, developed by Grafana Labs, which allows users to see their data via charts and graphs that are unified into one dashboard or multiple dashboards for easier interpretation and understanding.Grafana was built on open principles and the belief that data should be accessible throughout an organization, not just to a small handful of people. This fosters a culture where data can be easily found and used by anyone who needs it, empowering teams to be more open, innovative, and collaborative.

Architecture: Needs a DB to install. Best suited for metrics analysis.

Interface: Web based.

Visualization: Best suited to analyze Metrics. Offers different visualizations capabilities.

Supported data sources: Graphite, MySQL, PostgreSQL, Elastic Search.

Querying: Has a query editor to query and Visualize Metrics and also can use Elasticsearch.

3.8.3. Apache Superset

Apache Superset [14] is an easy-to-use Business Intelligence tool that collects and processes data in large volumes to produce visualized results like charts and graphs. Thus, the web application allows users to generate dashboards and reports which aid business growth.Apache Superset is cloud-native, and it is compatible with numerous options in each of the aforementioned customization categories.

Architecture: Apache superset is built entirely on top of python; it uses flask app builder internally. Best suited for Business Intelligence

Interface: Web based.

Visualization: Best suited to provide insights into large numbers or other data points.

Supported data sources: MySQL, MariaDB, PostgreSQL etc...

Querying: Uses SQL.

3.8.4. Dashboard choice

All these dashboards are similar to each other, but Kibana turned out to be the most beneficial because of RDF Indexer for Elasticsearch. This RDF Indexer solves the biggest problem with RDF data - indexing. Using other dashboards would force me to look for ways to get data from RDF into the dashboard.

4. Architecture

The architecture of a software system forms its structural foundation, defining how various components interact and collaborate to achieve the desired functionality. This section discusses the architectural design and solutions adopted for the software system under study. The purpose of this section is to provide a comprehensive understanding of the system architecture, highlighting its key components and the principles used in its development.

4.1. Description of the overall system architecture

The architectural structure of the system is designed to support the storage, management, and retrieval of RDF data. It encompasses several key components:

- **Database:** GraphDB is the primary database component responsible for storing and managing RDF data. It provides efficient storage, querying, and manipulation capabilities for semantic data.
- **Back-end:** Using libraries/frameworks for ontology processing, obtaining and saving data for visualization in the dashboard.

  Robot core: The library refers to the core functionality of the ROBOT tool. It contains the essential functions and modules required for executing ontology-related operations. The library provides a foundation for building and extending the capabilities of the ROBOT tool. Used to obtain metrics about ontology.

  RDF4J: This framework allows to transpose ROBOT results into RDF format using the DQV vocabulary and save this data to the database.

  Apache Jena: Allows to leverage SHACL to define shapes and apply validation rules to ontologies. As well as RDF4J has the ability to connect to a database of data and save the results of validation.

  Spring Boot: Used to start indexing directly from Indexer endpoint using REST API and makes it possible to update the data every week.
- **Front-end:** Kibana serves as the front-end interface for the system. It provides a user-friendly and visually appealing interface for users to interact with and visualize the RDF data stored in GraphDB.
- **RDF Indexer:** The RDF Indexer component plays a critical role in data retrieval. It is responsible for indexing the RDF data stored in GraphDB.

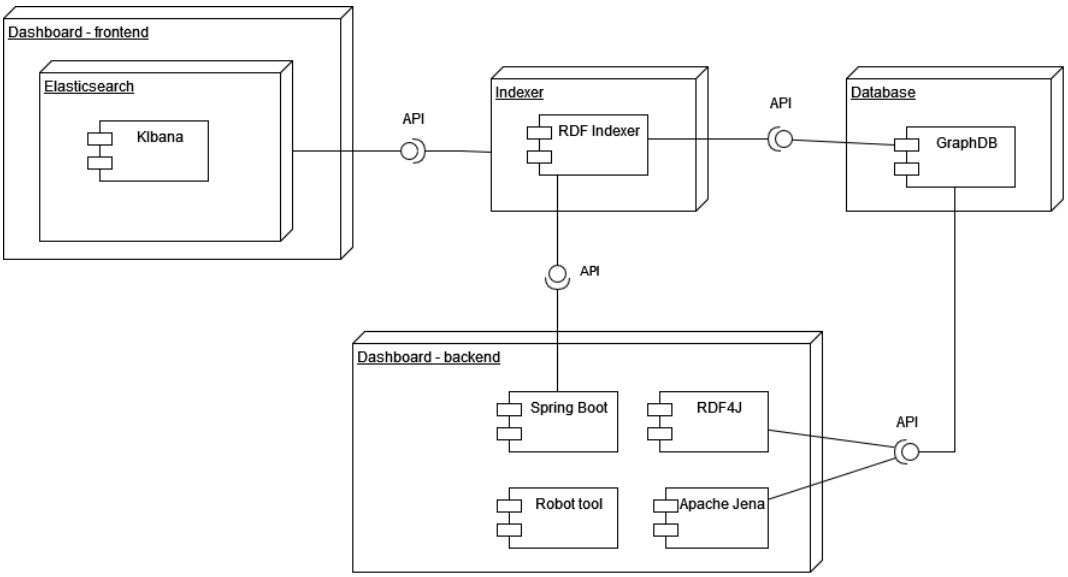

Figure 4.1.: Component diagram.

4.2. Overview of the data flow and communication

This section provides an overview of how data is processed, exchanged, and communicated within the system.

The data flow in the system begins by generating the ontology data we are interested in. This process involves the generation of metrics and the ontology violation report, the details of this process will be described in the section Implementation 5. Metrics are described using DQV vocabulary, violation report is generated using SHACL. After receiving the data for the ontology, the data is saved to the database. As you can see in the diagram below, this process occurs for each ontology. Once the data about each ontology has been saved to the database, the "Dashboard app" sends a request through the REST API to start indexing. In this request, the Dashboard app passes the configuration for indexing the data; we'll discuss the details of what this configuration consists of in the Implementation section. The indexer starts its indexing by querying the data from the database and then starts the indexing of the data in Elasticsearch. At this stage, the last step is the visualization of the data in Kibana. DQV data is visualised like a standard ROBOT measure table, SHACL data is much the same, but is also analysed to create different graphs and metrics.

This process described above will take place every week, as you can see in the diagram below. I would like to note that this process will be carried out only for ontologies that have some changes since the last update, in other words - if the ontology has remained unchanged, then it makes no sense to generate data about it again.

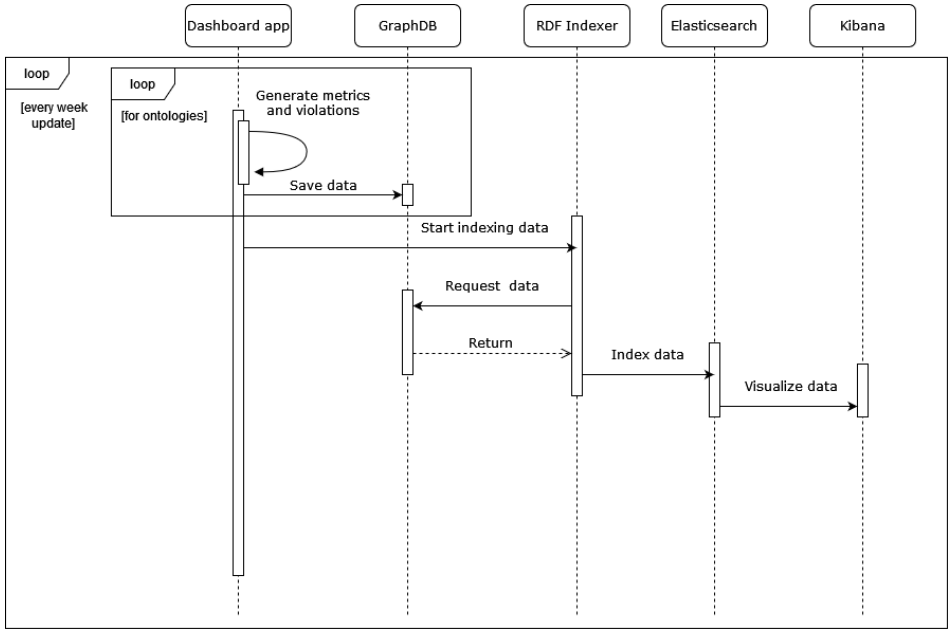

Figure 4.2.: Sequence diagram.

5. Implementation

This section presents the practical realization of the proposed solution. It serves as a comprehensive account of the technical aspects, development process, and implementation details that contribute to the successful execution of the project. This section highlights the efforts made to transform the conceptual design into a functional system, showcasing the practicality and effectiveness of the proposed solution.

Throughout this section, we delve into the step-by-step process of implementing the system.The implementation section provides valuable insights into the challenges encountered during the development phasem.

5.1. Overview of the implementation process

At the beginning of the implementation, the basis for me was the ROBOT tool, in which I was most interested in the two commands `robot measure` and `robot report`. One of my first goals was to get the results of these two commands and find a way to visualize them in a dashboard. First, I started to study the ROBOT tool, I needed to understand how it works and implement it into my code. After experimenting with generating various reports and measurements for ontologies, I set about implementing it. Since the ROBOT tool has its own ROBOT Core library - my choice was obvious. In later sections, I will discuss the implementation of these two commands.

5.2. Robot measure

The `robot measure` command computes a number of metrics about ontology, such as entity and axiom counts, qualitative information such as OWL 2 profiles and more complex metrics aimed at informing ontology developers such as logical expressivity and axiom shape.

The following example shows what the measure output looks like on the Planarian ontology [20]:

Metric	**Metric value**	**Metric type**
class_count	13779	single_value
datatype_count	5	single_value

Table 5.1.: Output of measure ROBOT command

After receiving the metrics, my task was to present them in RDF format. I understood this long before the implementation and started looking for some RDF grammar with which I could describe it. After studying all the possible ways, my choice fell on DQV vocabulary. This grammar was perfect for me because with it I could describe the result of a metric measure with the same meaning as it is presented in the table above.

This is how the Table 5.3 above could be presented in the DQV vocabulary:

```
@prefix ex: <http://example.org/ns#> .

ex:measurement1 a dqv:QualityMeasurement ;
    dqv:isMeasurementOf ex:class_count;
    dqv:value: 13779 ;
```

`dqv:isMeasurementOf` indicates the metric being observed, `dqv:value` refers to values computed by metric and `dqv:QualityMeasurement` represents a metric value providing quantitative or qualitative information.

To generate the output of the `robot measure` command, as I mentioned earlier, I used the ROBOT core library. All this requires is the ontology itself as an input parameter. ROBOT core will generate a Map with metrics, the next step is to transform this data into RDF format, i.e. to describe data using the DQV vocabulary. To implement this transformation, I used the RDF4J framework.

5.3. Robot report

The `robot report` command runs a series of quality control SPARQL queries over the input ontology and generates a TSV or YAML report file based on the results. Each query has a logging level to define the severity of the issue: ERROR, WARN, or INFO.

The following example shows what the report looks like on the Planarian ontology [20]

Level	**Rule Name**	**Subject**	**Property**	**Value**
ERROR	duplicate_label	CARO:0000006	rdfs:label	material anatomical entity

Table 5.2.: Output of report ROBOT command

As for the robot report, there is no well accepted declarative standards for representing constraint violation in RDF. Thus, we decided to explore how easy it would be to map ROBOT constraints in SHACL, reusing also the SHACL result representation in RDF. During implementation, I started using RDF4J to validate ontologies. I started with simple OBO rules. These validation rules are described using SPARQL queries. My goal was to start transforming these rules into SHACL shapes.

I propose to consider the rule **Lowercase Definition**.

This rule has following problem: A definition or elucidation does not begin with an uppercase letter. This may be indicative of inconsistent formatting.

This is what the query looks like in SPARQL:

```
PREFIX obo: <http://purl.obolibrary.org/obo/>

SELECT DISTINCT ?entity ?property ?value WHERE {
  VALUES ?property { obo:IAO_0000115
                     obo:IAO_0000600 }
  ?entity ?property ?value .
  FILTER (!regex(?value, "^[A-Z0-9]"))
  FILTER (!isBlank(?entity))
}
ORDER BY ?entity
```

And this is how this query looked after my transformation into SHACL shape:

```
@prefix ex: <http://example.com/ns#> .

ex:lowercase_definition
    a sh:NodeShape ;
    sh:targetClass owl:ObjectProperty, owl:AnnotationProperty,
    owl:Class ;
    sh:property [
        sh:path obo:IAO_0000115 ;
        sh:severity sh:Info;
        sh:message "lowercase_definition" ;
        sh:pattern "^[A-Z0-9](.*)" ;
    ] ;
    sh:property [
        sh:path obo:IAO_0000600 ;
        sh:severity sh:Info;
        sh:message "lowercase_definition" ;
        sh:pattern "^[A-Z0-9](.*)" ;
    ] .
```

Further, when transforming more complex rules, it turned out that ordinary SHACL shapes do not have the ability to compare nodes. The only way to implement this is to use `sh:sparql` property in my SHACL shapes. This property would allow the use of sparql queries in SHACL shapes.

Unfortunately for me, SHACL in RDF4J does not support this, so I had to use another framework with SHACL validation called Apache Jena, which already supported using SPARQL in SHACL shapes.

The following code shows how the above mentioned Lowercase Definition rule can be represented with Sparql in SHACL shape:

```
ex:lowercase_definition
 a sh:NodeShape ;
 sh:targetClass owl:AnnotationProperty, owl:ObjectProperty,
 owl:Class ;
 sh:message "lowercase_definition" ;
 sh:severity sh:Info ;
 sh:sparql [
    a sh:SPARQLConstraint ;
    sh:prefixes ex: ;
    sh:select """
        SELECT DISTINCT $this ?property ?value WHERE {
            VALUES ?property { obo:IAO_0000115 obo:IAO_0000600 }
            $this ?property ?value .
            FILTER (!regex(?value, "^[A-Z0-9]"))
            FILTER (!isBlank($this))
        }
        ORDER BY $this
    """
 ] .
```

As a result, the decision was to transform those rules that do not require SPARQL into SHACL shapes without SPARQL, while the rest were transformed into shapes with SPARQL. This is what ontology validation using Apache Jena looks like in code:

```
public Model getReport(String shape, String ontology){

 // loading shapes graph
 Graph shapesGraph = RDFDataMgr.loadGraph(shape);

 // loading ontology
 Graph dataGraph = RDFDataMgr.loadGraph(ontology);

 // getting of shapes from shapesGraph
 Shapes shapes = Shapes.parse(shapesGraph);
 shapesGraph.getPrefixMapping().getNsPrefixMap();

 // creation of validation report
 ValidationReport report = ShaclValidator.get()
                              .validate(shapes, dataGraph);

 return report.getModel();
}
```

5.4. Ontology versions

Versions in ontologies is a very interesting topic for discussion. Ontologies are standardized and have special attributes such as `owl:version` or `owl:versionInfo` but problem is that not all ontologies fill them in, and if they do, they are not consistent with the others. Thus, the version information has to be consolidated. There are no tools to track versions of ontologies in chronological order. Versions help in managing changes to an ontology. When modifications are made to an ontology, a new version is created to represent the updated state. This allows users to track and understand the evolution of the ontology and facilitates collaboration among ontology developers. Versioning enables compatibility and interoperability between different systems that use the ontology. By referencing a specific version, applications or services can ensure that they are using a consistent and compatible ontology representation, even if newer versions are released.

My goal was to implement a way to track the number of violations in each version of the ontology, or compare how many classes the ontology has grown by, etc. The problem was that different versioning schemes, such as numeric identifiers (e.g 1.0 1.1), date stamps (e.g 2021-01-01), or combinations of these, were often used to specify versions in ontologies. In order to work with the versions in Kibana, I needed exactly the date in each ontology because of the peculiarities of Kibana. Since the dashboard is updated every week, I decided to use the update date as the version for ontologies that do not have a date. This would allow me to avoid this problem and view the ontology data in chronological order. I took the ontology versions from "ontology-version-extractor", which is a tool that provides up-to-date versions of ontologies. It works so that when I update new ontologies, I add a version to all metrics and violation reports in order to distinguish them from each other in the dashboard.

5.5. Storing data in GraphDB

The data storage structure in GraphDB also plays an important role, as it turns out. In order to store data, I decided to use the so-called "Default graph" to store general information and create a separate graph for each ontology.

The following table shows what the information looks like for each ontology in the "Default graph":

Subject	**Predicate**	**Object**
obo:ado.owl	dc:title	"Alzheimer's Disease Ontology"
obo:ado.owl	owl:versionInfo	"2022-06-11"xsd:date
obo:ado.owl	foaf:homepage	"https://github.com/Fraunhofer-SCAI-Applied-Semantics/ADO"

Table 5.3.: Output of measure ROBOT command

As you can see "Default graph" contains the name, the latest actual version and a link to the ontology homepage.

Now I propose to consider what metrics and violation reports look like in GraphDB. An example is the graph for the Ascomycete Phenotype Ontology (APO), whose name is the IRI of the ontology - http://purl.obolibrary.org/obo/apo.owl

The following table shows what violation data looks like:

Subject	**Predicate**	**Object**
_:genid-209223	rdf:type	sh:ValidationResult
_:genid-209223	sh:sourceShape	ex:missing_obsolete_label
_:genid-209223	sh:resultMessage	"missing_obsolete _label"
_:genid-209223	sh:resultSeverity	sh:Warning
_:genid-209223	sh:sourceConstraintComponent	sh:SPARQLConstraintComponent
_:genid-209223	sh:focusNode	obo:APO_0000133
_:genid-209223	sh:value	"cell cycle arrest in metaphase"
_:genid-209223	owl:versionInfo	"2023-04-27"

The following table shows what metric data looks like:

Subject	**Predicate**	**Object**
_:node1gvh95b0nx631	rdf:type	dqv:QualityMeasurement
_:node1gvh95b0nx631	dqv:value	ex:missing_obsolete_label
_:node1gvh95b0nx631	owl:versionInfo	"20230427"

5.6. Harvesting data into ElasticSearch

Once the data has been saved to the database, our next goal is to visualize it in Kibana. As I mentioned earlier for this purpose, I use RDF Indexer as an intermediary between GraphDB and Kibana. Now I would like to consider the work of the indexer and its capabilities.

On the next picture you can see the graphical interface of the indexer, which includes parameters such as the name, automatic or manual updating and the data source. The indexer then uses SPARQL to query the data.

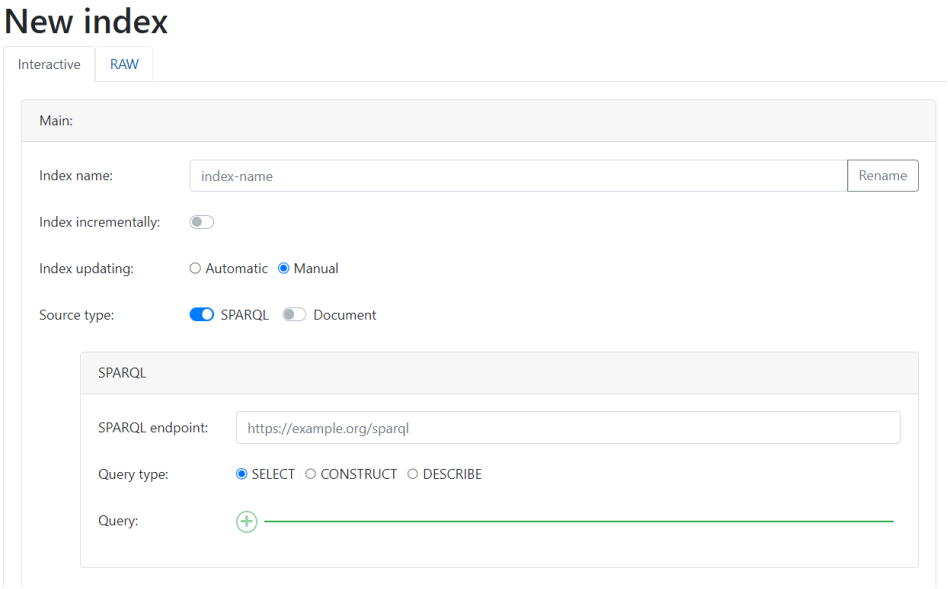

Figure 5.1.: Creating a new index in RDF Indexer

Consider my SPARQL query that I used:

```
PREFIX dqv: <http://www.w3.org/ns/dqv#>
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX dqv: <http://www.w3.org/ns/dqv#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?subject ?predicate ?object.
    ?subject dc:title ?nameOfOntology .
    ?subject foaf:homepage ?homePage }
FROM <http://www.ontotext.com/explicit>
WHERE {
    ?g dc:title ?nameOfOntology .
    ?g foaf:homepage ?homePage .
GRAPH ?g {
    {
        ?subject a sh:ValidationResult; ?predicate ?object;
    } UNION {
        ?subject a dqv:QualityMeasurement; ?predicate ?object ;
    }
  }
}
```

I would like to mention that in this SPARQL query, I excluded the `BIND()` functions that were responsible for formatting in order to improve readability.

Using the `CONSTRUCT` query I retrieve union data from the "Default graph" and other graphs. As you may have noticed, at first I make the query from `<http://www.ontotext.com/explicit>` - this is the "Default graph".

As I mentioned earlier, the "Default graph" contains the IRI of the ontology as the subject, and these same IRIs are the names of the ontology graphs. Then in each graph we select all subjects of type `sh:ValidationResult` and `dqv:QualityMeasurement`. The last thing this query does is to add to each subject the appropriate graph with the name of the ontology and its homepage "Default graph".

Also, you may have noticed that the indexer has the ability to automatically update. But in this case I decided to go a little differently and start indexing immediately after updating ontologies from code:

```
@Component
public class ScheduledTasks {

 String indexUrl =
 http://host.docker.internal:8080/api/configAndIndex";

 public void updateIndex() {
   String indexUrl =
   http://host.docker.internal:8080/api/configAndIndex";
   HttpHeaders headers = new HttpHeaders();
   headers.setContentType(MediaType.APPLICATION_JSON);

   String config = parser.getConfig();
   HttpEntity<String> requestEntity =
   new HttpEntity<>(config, headers);

   ResponseEntity<Long> responseEntity =
   restTemplate.exchange(indexUrl,
                         HttpMethod.PUT,
                         requestEntity,
                         Long.class);
 }
}
```

5.7. Visualization of data

The data have been indexed and the last step is visualization of the data. This dashboard contains a navigation bar 5.1, which contains 3 dashboard variants - dashboard for one ontology, for specific ontologies and for all ontologies.

I propose to consider each part selectively and start with dashboard for all ontologies 5.2. As you can see this part of the dashboard shows us a list of ontologies with their names and homepages, and gives us the current number of ontologies in the dashboard. The next graph 5.3 in dashboard for all ontologies shows the top ontologies by number of violations. And the last graph 5.4 shows us the rules that were most frequently violated among the ontologies.

As for the single ontology dashboard 5.5, the user has the ability to select a single ontology from the list and its version. Then the user will have access to information such as the number of Warnings, Errors and Info. After selecting an ontology, the user can see table 5.6 with `robot measure` output and table 5.7 with `robot report`. The user can also find out if the number of violations increases/decreases in versions 5.8.

To compare several ontologies, e.g. by number of violations, the user can use dashboard for specific ontologies 5.9. In fact, the dashboard for all ontologies differs from the dashboard for specific ontologies only in the ability to select specific ontologies, they are otherwise identical.

5.8. Screenshots of the dashboard

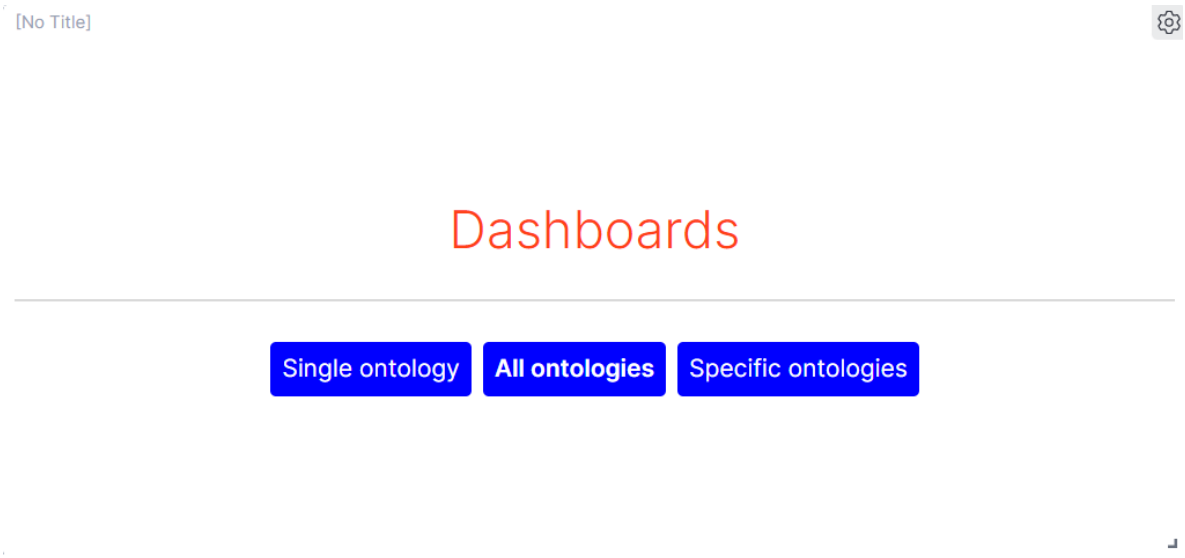

Figure 5.1.: Navigation bar

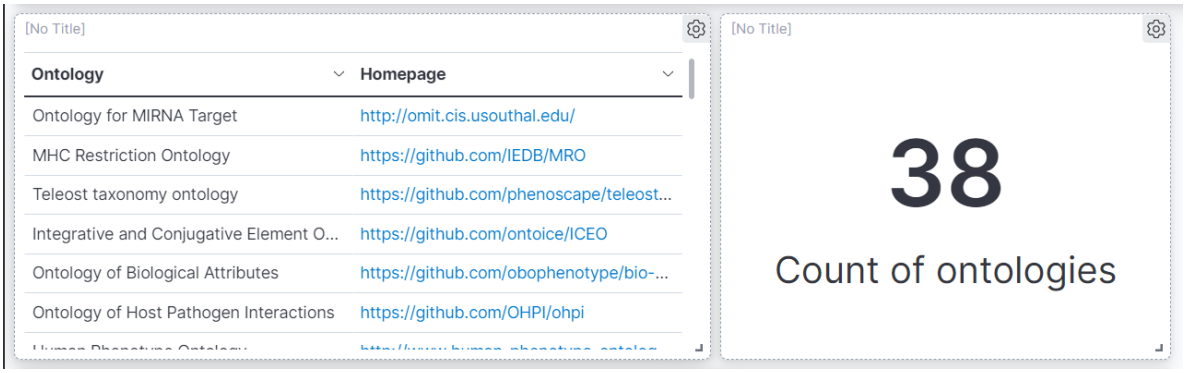

Figure 5.2.: List and count of ontologies

Figure 5.3.: Top ontologies by vilations graph

Figure 5.4.: Top most violated rules graph

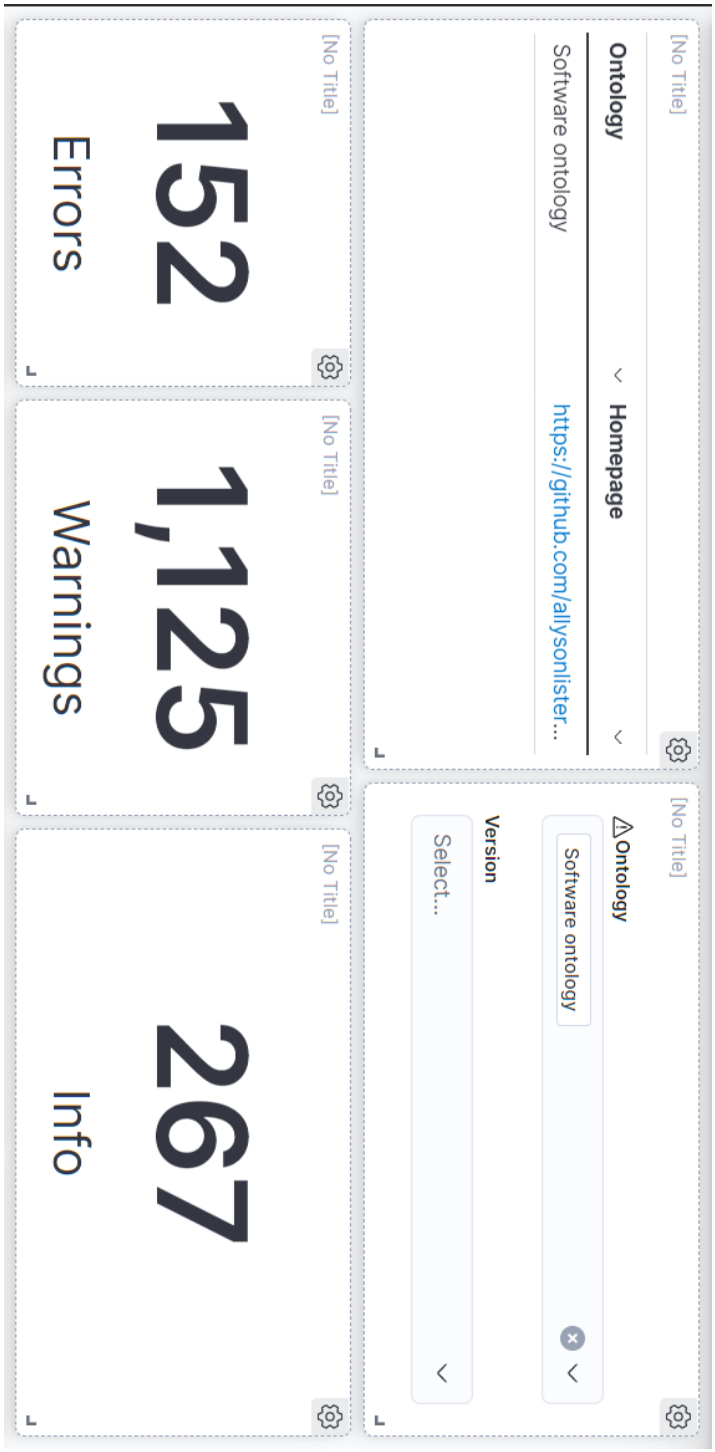

Figure 5.5.: Selection of one of the ontologies and violations information

Robot measure

74 documents

Time ↓	Metric	Metric Value
Mar 5, 2023	ontology_version_iri	http://www.ebi.ac.uk/swo/swo/releases/2023-03-05/swo.owl
Mar 5, 2023	owl2	true
Mar 5, 2023	individual_count_incl	446
Mar 5, 2023	rdfs	false
Mar 5, 2023	axiom_count_incl	19366
Mar 5, 2023	ontology_iri	http://www.ebi.ac.uk/swo/swo.owl
Mar 5, 2023	class_count_incl	1971
Mar 5, 2023	owl2_el	false

Rows per page: 50 1 of 2

Figure 5.6.: ROBOT measure table

Robot report

1618 documents

Time ↓	Level	Rule name	Node	Value
Mar 5, 2023	Warning	missing_definition	http://www.ebi.ac.uk/swo/SWO_9000076	http://www.ebi.ac.uk/swo/SWO_9000076
Mar 5, 2023	Warning	duplicate_label_synonym	http://edamontology.org/data_2603	Microarray data
Mar 5, 2023	Warning	missing_definition	RO_0004046	RO_0004046
Mar 5, 2023	Warning	missing_definition	http://www.ebi.ac.uk/swo/SWO_0000237	http://www.ebi.ac.uk/swo/SWO_0000237

Rows per page: 50 1 of 10

Figure 5.7.: ROBOT report table

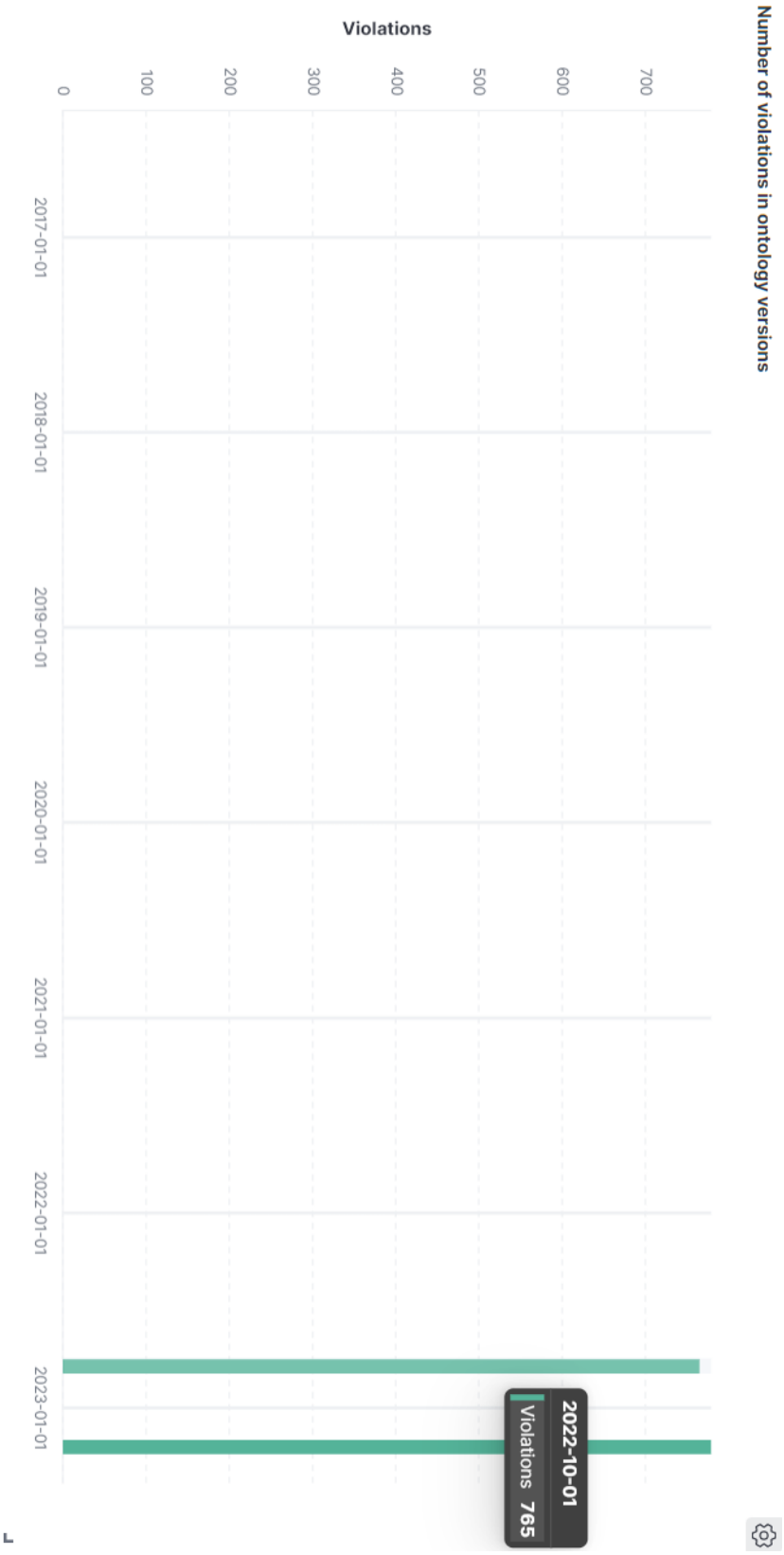

Figure 5.8.: Number of violations in ontology versions

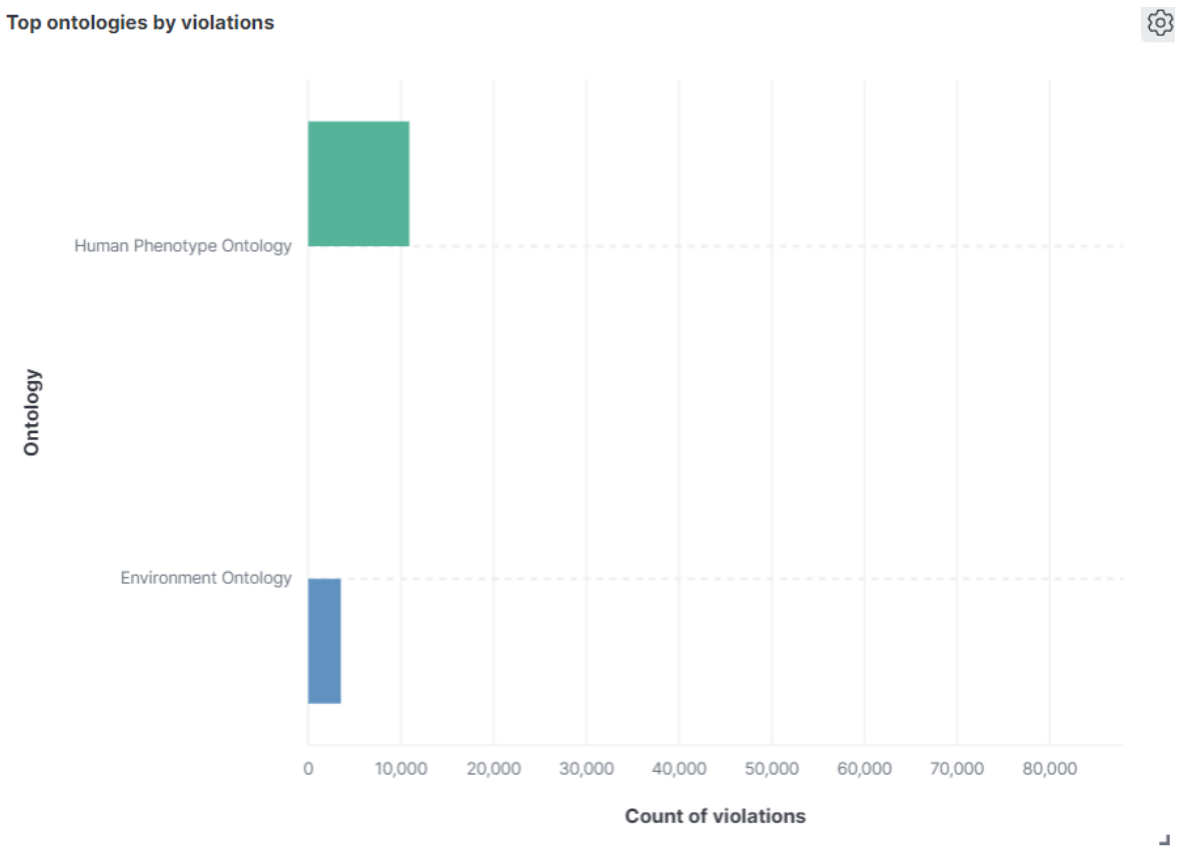

Figure 5.9.: Graph for comparing two or more ontologies by the number of violations

6. Testing

In this section I will describe the testing, namely the automation of SHACL shapes testing and the usefulness and usability test which I did together with the OBO community.

6.1. Automation of SHACL rule testing

The problem was to automate the checking of SHACL rules for correctness. By writing tests, we can check if our SHACL rules are correctly implemented and functioning as intended. The idea was to transform the robot measure output to RDF and run two SPARQL queries and compare SHACL and ROBOT outputs. One way of doing this transfromation was through TARQL.

TARQL [26] is a query language designed for querying and manipulating RDF data. TARQL supports querying data stored in CSV and TSV formats. It allows easily import data from these tabular formats into RDF datasets and perform queries on the combined data.

Suppose we have a table:

Level	**Rule name**	**Subject**	**Property**	**Value**
ERROR	multiple_labels	BFO:0000050	rdfs:label	part of

Using TARQL I can create a SPARQL query to transform the above table into RDF:

```
PREFIX sh: <http://www.w3.org/ns/shacl#>

CONSTRUCT {
    ?id a sh:ValidationResult;
    sh:focusNode ?nodeTyped;
    sh:resultMessage ?sourceShape;
    sh:value ?Value;
}
FROM <file:plana.csv>
WHERE {
    BIND(UUID() AS ?id)
    BIND(REPLACE(?Subject, ":", "_") AS ?node)
    BIND(URI(CONCAT("http://purl.obolibrary.org/obo/", ?node))
    AS ?nodeTyped)
    BIND(STR(?RuleName) AS ?sourceShape)
}
```

At the output we will have:

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix sh:  <http://www.w3.org/ns/shacl#> .

<urn:uuid:8678183e-0920-44bd-a7a2-8d006f4880f7>
        rdf:type          sh:ValidationResult ;
        sh:focusNode      <obo:BFO_0000050> ;
        sh:sourceShape    "multiple_labels" ;
        sh:value          "part_of" .
```

When we have two RDF graphs, one from SHACL and one from ROBOT, we need to make two SPARQL queries to get the data from them and compare the results.

6.2. Usefulness and usability tests

For the usefulness and usability tests I have created test scenarios and questions for which I have sent to the OBO Community. In total I have 4 test scenarios that test usefulness where users gave comments and rated the scenario on a scale of 1 to 5, with 1 being very easy and 5 being very difficult. And also 10 questions on usability with a scale of 1 to 5 where 1 is strongly disagree and 5 is strongly agree.

The results are as follows:

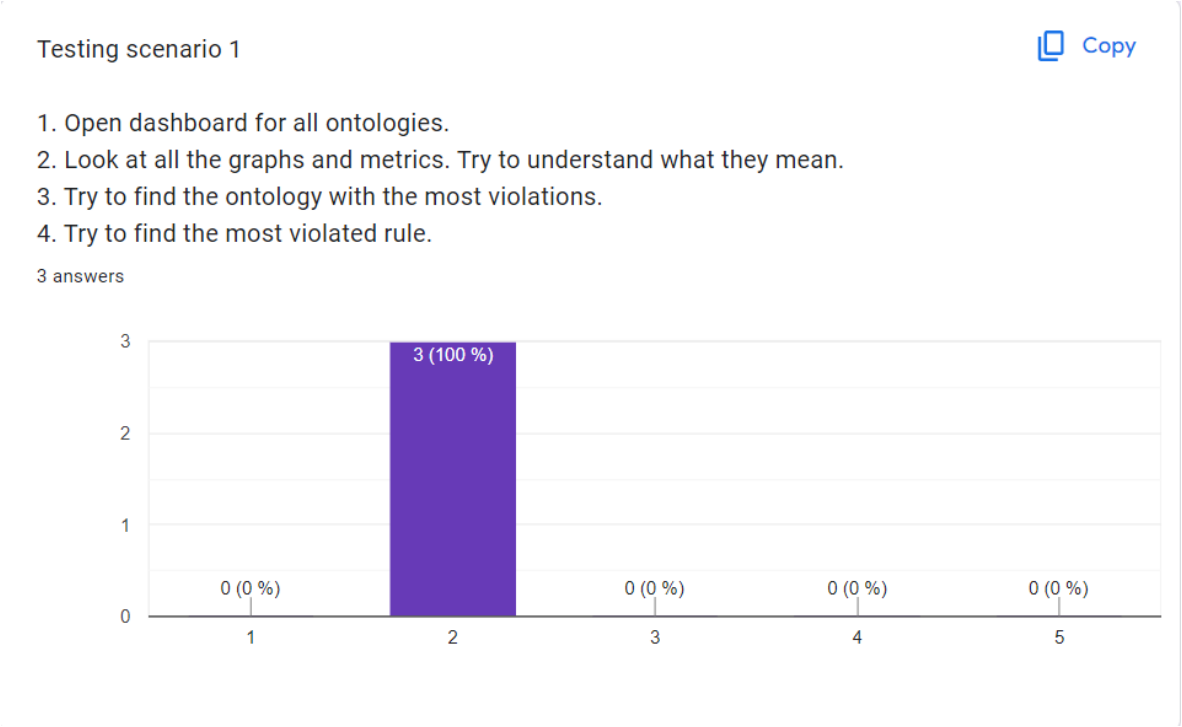

Figure 6.1.: Testing scenario 1

Feedback on testing scenario 1:

"Load time is very slow. I'm not sure I would order by 'number of violations' - I'd order by least number to give the user the best info first. I don't

know why the most screen real estate is dedicated to three buttons that make me choose an ontology. I don't know why I would choose an ontology first. A dashboard should give me an overview of all the ontologies at a glance first. I most likely don't know which ontology to use if I am coming to a dashboard. Or I would like to find the release artifacts for a specific ontology; alphabetically sorted makes this easiest"

"A logarithmic scale or a proportional scale might be useful as 87k violation in an ontology with 1 mio classes might be "small" compared to 700 violation in a 800 classes large Ontology"

"Overall this seems very powerful, but there's a bit of a learning curve. One small problem is that the first 'Dashboards' box is too tall, so I need to scroll down to see most of the interesting information. A larger problem I had is that the various search and filter options were connected in non-obvious ways, so if I wanted to clear my search / filter results then I had to look in a number of boxes. I got stuck a few times when I was exploring, I've used the Elastic Search stack in the past, but not for a while, and I've always found it very powerful but also tricky to just dive in and start using. I'm not always confident that I'm seeing all the information that I was trying to get."

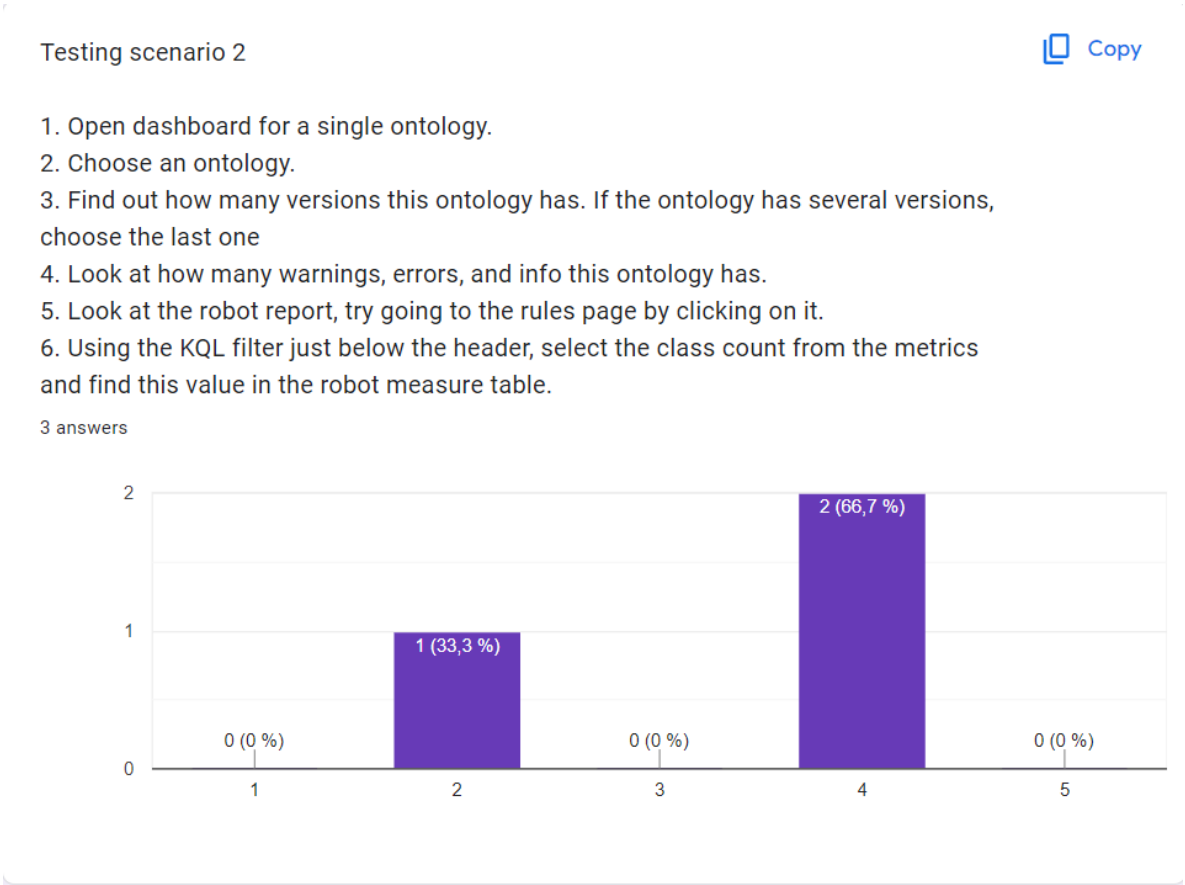

Figure 6.2.: Testing scenario 2

Feedback on testing scenario 2:

"there is only one ontology in the list - software ontology."

"It is not very intuitive to navigate some information is easily available, however finding for example all error level messages from the robot report is quite difficult as the robot report is only sortable by time.
The function and accesibility to the KQL Window can be more emphasized."

"I eventually found 'class_count' in the 'Robot measure' box and typed that in the search bar, but the search bar did not suggest it or autocomplete it. Several times I got stuck with 0 results, and had trouble resetting the interface."

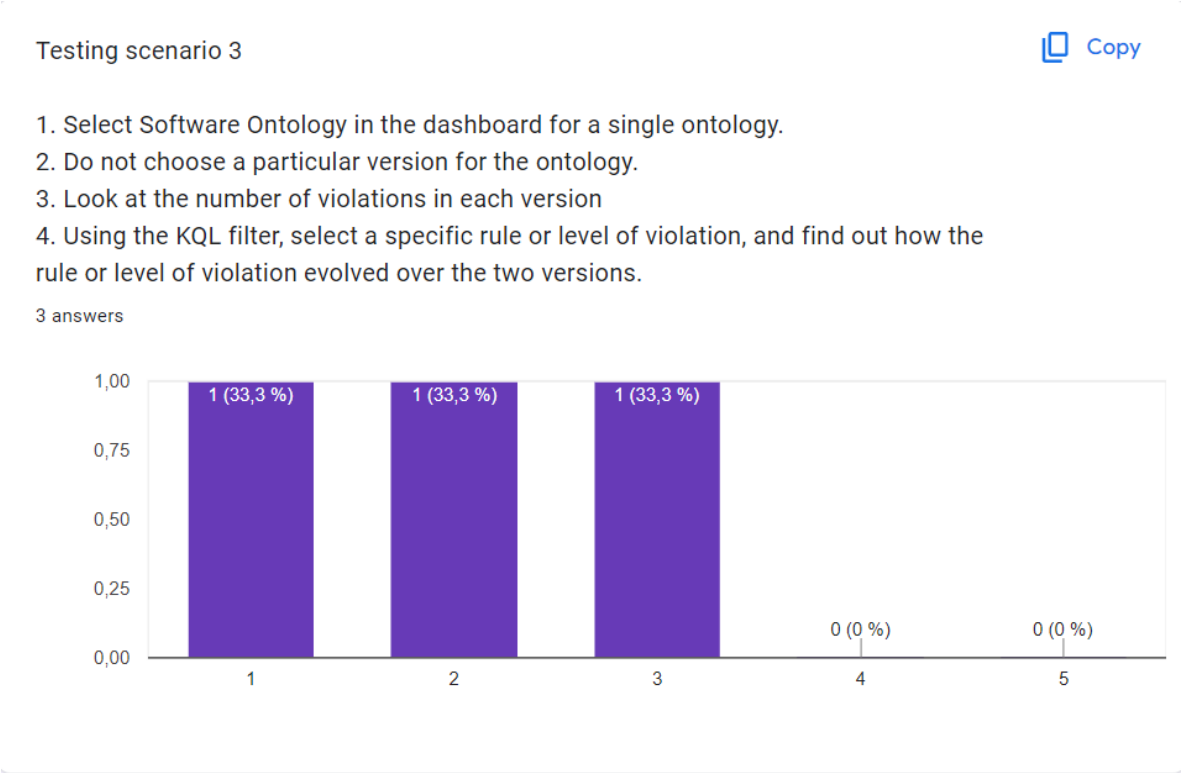

Figure 6.3.: Testing scenario 3

Feedback on testing scenario 3:

"I can't figure out how to select the software ontology - I can only click on the link to the homepage"

"no recommendations for improvements other than maybe other than that a pie chart for groups of violations might be useful"

"I had the easiest time with this task."

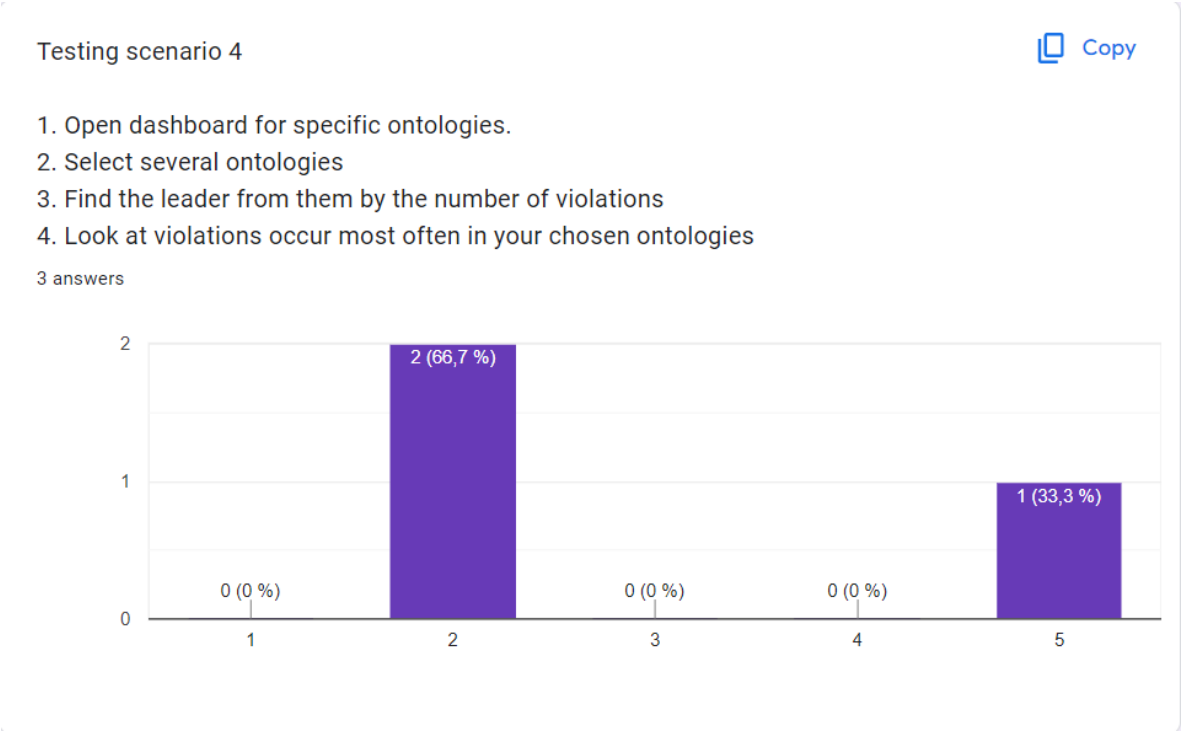

Figure 6.4.: Testing scenario 4

Feedback on testing scenario 4:

> "I don’t see how to select multiple ontologies. if I click
> "all ontologies, I should get a big list, not a corner window where I can
> only see 6 ontologies."
>
> "A header explaining where the selected and where the selectable
> Ontologies are listed would be helpfull. Also at least in my opinion
> "Europeans" normally assume that the tool for selectable structures
> is listed to the left "before" the the list of selected structures"
>
> "The ’Ontologies’ box on the right with multiple selection uses
> case-sensitive
> autocomplete, but I was expecting case-insensitive autocomplete."

Results of usability test you can find in Appendix B

The results of the feedback generally let me know that OBO Foundry was interested in my solution. Basically, all the subjects did not like something in the visualization.One of the subjects said he didn’t want to see single ontology and specific ontology dashboards separated. The subjects didn’t like the fact that the most interesting information is at the bottom and you have to get to it. But also a few subjects liked the ability to track versions of the ontology, as well as the ability to filter data in the tables using Kibana query language - KQL. In general, although each subject did not like something, I think that the dashboard made a better impression on them.

7. Conclusion

In conclusion, this bachelor thesis has successfully addressed the objective of designing and implementing an interactive ontology dashboard backend by standardized vocabularies (Data Quality Vocabulary, SHACL), aimed at providing users with intuitive tool to explore and analyze ontologies in an interactive and visual manner. As a prominent test case, we aimed at the biomedical terminologies, taxonomies, and ontologies maintained by the OBO Foundry.

Throughout the research and development process, we conducted an in-depth analysis of ontology quality problem and identified the key requirements for an effective dashboard.

To ensure the usability and usefulness of the dashboard, extensive user testing and feedback were conducted with the OBO Foundry users community.

According to the results of the tests it became clear that OBO Foundry is interested in this topic and proved to be a very responsive community that is happy to support new tools related to the topic. One of the OBO Community representatives was very surprised by the work done on the description in SHACL shapes of the ROBOT report rules and liked the idea.

Some limitations were found with the visualization capabilities in Kibana, so I could say that Kibana did not meet all of my expectations, but most of them. There is one limitation to the number of ontologies caused by RDF Indexer, the fact that RDF Indexer ran out of memory when indexing a large amount of data at once. But there is also another limitation - huge ontologies that require a lot of memory to process. However, all these limitations give me great intentions for further improvement and development in this direction in general.

In summary, this bachelor thesis has successfully achieved its objective of developing an interactive ontology dashboard. The resulting dashboard provides users with an interactive dashboard to explore and analyze ontologies effectively.

Bibliography

[1] GraphDB Downloads and Resources — graphdb.ontotext.com. URL https://graphdb.ontotext.com. [Accessed 13-May-2023].

[2] Shapes constraint language (SHACL). Technical report, W3C, July 2017. URL https://www.w3.org/TR/shacl/. [Accessed 05-January-2023].

[3] Apache Jena, 2023. URL https://jena.apache.org/. [Accessed 13-May-2023].

[4] JSON, 2023. URL https://www.json.org/json-en.html. [Accessed 01-May-23].

[5] R. Albertoni and A. Isaac. Introducing the data quality vocabulary (DQV). *Semantic Web*, 12(1):81–97, 2021.

[6] P. Archer. Data catalog vocabulary (dcat) (w3c recommendation). Online, January 2014. URL https://www.w3.org/TR/vocab-dcat/. [Accessed 01-January-2023].

[7] C. Aubert, P. Buttigieg, M. Laporte, M. Devare, and E. Arnaud. Cgiar agronomy ontology, 2017. URL http://purl.obolibrary.org/obo/agro.owl. licensed under CC BY 4.0.

[8] B. M. D. Beckett. RDF/XML syntax specification. W3C Recommendation, February 2004. URL http://www.w3.org/TR/rdf-syntax-grammar/. [Accessed 31-December-2022].

[9] D. Beckett and T. Berners-Lee. Turtle - terse rdf triple language. W3c team submission, W3C, January 2008. URL http://www.w3.org/TeamSubmission/turtle/. [Accessed 7-May-2023].

[10] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. Rdf 1.1 turtle. *World Wide Web Consortium*, pages 18–31, 2014.

[11] J. Debattista, C. Lange, and S. Auer. daQ, an ontology for dataset quality information. In *LDOW*, 2014.

[12] J. Debattista, C. Lange, and S. Auer. Representing dataset quality metadata using multi-dimensional views. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 92–99, 2014.

[13] E. R. developers. Welcome · Eclipse RDF4J™ — The Eclipse Foundation — rdf4j.org. https://rdf4j.org/. [Accessed 13-May-2023].

[14] I. S. Ganiyu, S. Mittal, H. Balasankula, and Y. Sanghvi. What is apache superset?: 3 important factors - learn, Dec 2022. URL https://hevodata.com/learn/apache-superset/.

[15] W. W. Group. Rdfa primer: Bridging the human and data webs, October 2008. URL http://www.w3.org/TR/xhtml-rdfa-primer/. [Accessed 10-January-2023].

[16] R. Hat. What is grafana?, May 2022. URL https://www.redhat.com/en/topics/data-services/what-is-grafana. [Accessed 10-January-2023].

[17] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. *OWL 2 Web Ontology Language Primer*. W3C, 2009. URL http://www.w3.org/TR/owl2-primer/. [Accessed 5-January-2023].

[18] R. C. Jackson, J. P. Balhoff, E. Douglass, N. L. Harris, C. J. Mungall, and J. A. Overton. ROBOT: A Tool for Automating Ontology Workflows. *BMC Bioinformatics*, 20(1):407, 2019.

[19] D. Longley, P.-A. Champin, and G. Kellogg. JSON-ld 1.1. W3C recommendation, W3C, July 2020. https://www.w3.org/TR/2020/REC-json-ld11-20200716/.

[20] obophenotype. The PLANarian Anatomy Ontology, Mar 2023. URL https://github.com/obophenotype/planaria-ontology. [Online; accessed 22-May-2023].

[21] A. Phillips and M. Davis. Tags for identifying languages, March 2006. URL https://tools.ietf.org/html/bcp47. [Accessed 23-May-2023].

[22] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, Jan. 2008. URL https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/. [Accessed 15-January-2023].

[23] G. Schreiber and Y. Raimond. Rdf 1.1 primer w3c working group note. Online, 2014. URL https://www.w3.org/TR/rdf11-primer/. [Accessed 10-January-2023].

[24] G. Schreiber and Y. Raimond. Rdf 1.1 primer w3c working group note, section - 3.2 iris. Online, 2014. URL https://www.w3.org/TR/rdf11-primer/#section-IRI. [Accessed 10-January-2023].

[25] A. Sureka. What is kibana used for? URL https://www.clariontech.com/platform-blog/what-is-kibana-used-for-10-important-features-to-know. [Accessed 10-January-2023].

[26] Tarql. Turn CSV into RDF using SPARQL syntax, 2019. URL http://tarql.github.io/. [Accessed 24-May-2023].

[27] B. Thomas and P. Eric. Shape Expressions (ShEx) Primer, 2023. URL https://shexspec.github.io/primer/. [Accessed 05-May-2023].

[28] Wikipedia. URI — Wikipedia, the free encyclopedia. http://ru.wikipedia.org/w/index.php?title=URI&oldid=125172905, 2023. [Accessed 03-January-2023].

[29] M. Švagr. Interactive dashboard over RDF, 2021. URL https://dspace.cvut.cz/handle/10467/94712. [Accessed 23-May-2023].

A. Overview of RDF Syntaxes

Standards for encoding RDF statements now include the following five most popular syntaxes:

- **Turtle** [9] is the most popular text syntax for RDF statements. The W3C describes it as a "compact and natural text form" that includes abbreviations for commonly used patterns.It provides a compact syntax for writing RDF triples and is commonly used for exchanging and sharing RDF data.

  Here's an example of RDF data represented in Turtle format:

```
@prefix ex: <http://example.com/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:John a foaf:Person ;
    ex:knows ex:Alice .

ex:Alice a foaf:Person .
```

  In this example, we define two individuals ('ex:John' and 'ex:Alice') as persons and their friendship relationship.

- **JSON-LD** [19] uses the JSON[4] syntax. JSON-LD is designed to be usable directly as JSON, i.e. easily accessed by people without RDF background. It enables the expression of relationships between resources using standard RDF vocabularies and ontologies.

  Here's an example of a JSON-LD document:

```
{
  "@context": {
    "ex": "http://example.com/",
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "foaf": "http://xmlns.com/foaf/0.1/"
  },
  "@graph": [
    {
      "@id": "ex:John",
      "@type": "foaf:Person",
      "ex:knows": "ex:Alice"
    },
    {
      "@id": "ex:Alice",
      "@type": "foaf:Person"
    }
  ]
}
```

- **N-Triples** [10] is a subset of the Turtle syntax, designed to be a simpler text-based format for RDF statements for improved ease of use by humans writing statements. N-triples is suitable for batch processing of large RDF documents.

  Here's an example of RDF data represented in N-Triples format:

```
<http://example.com/John>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://xmlns.com/foaf/0.1/Person> .

<http://example.com/John>
    <http://example.com/knows>
        <http://example.com/Alice> .

<http://example.com/Alice>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://xmlns.com/foaf/0.1/Person> .
```

- **RDF/XML** [8] is a syntax format used to represent RDF data in an XML. It provides a way to express RDF triples and their relationships using XML tags and attributes.

  Here's an example of RDF data represented in RDF/XML syntax:

```
<rdf:RDF xmlns:ex="http://example.com/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/">

  <rdf:Description rdf:about="http://example.com/John">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
    <ex:knows rdf:resource="http://example.com/Alice"/>
  </rdf:Description>

  <rdf:Description rdf:about="http://example.com/Alice">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  </rdf:Description>
</rdf:RDF>
```

- **Resource Description Framework in Attributes (RDFa)** [15] is a way to express RDF data within HTML. It provides a set of markup attributes to augment visual information on the Web with machine-readable hints.

Here's an example of RDF data represented in RDFa syntax:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ex="http://example.com/"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <head>
    <title>RDFa Triples Example</title>
  </head>
  <body>
    <div typeof="foaf:Person" about="ex:John">
      <div property="ex:knows" resource="ex:Alice"></div>
    </div>
    <div typeof="foaf:Person" about="ex:Alice"></div>
  </body>
</html>
```

B. Usability test results

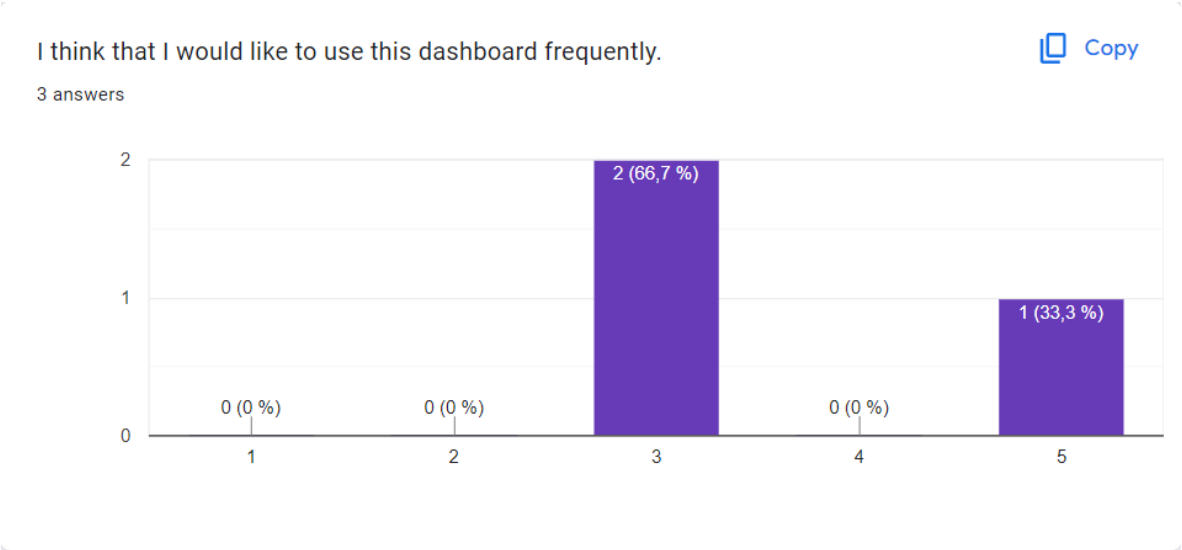

Figure B.0.1.: Usability question 1

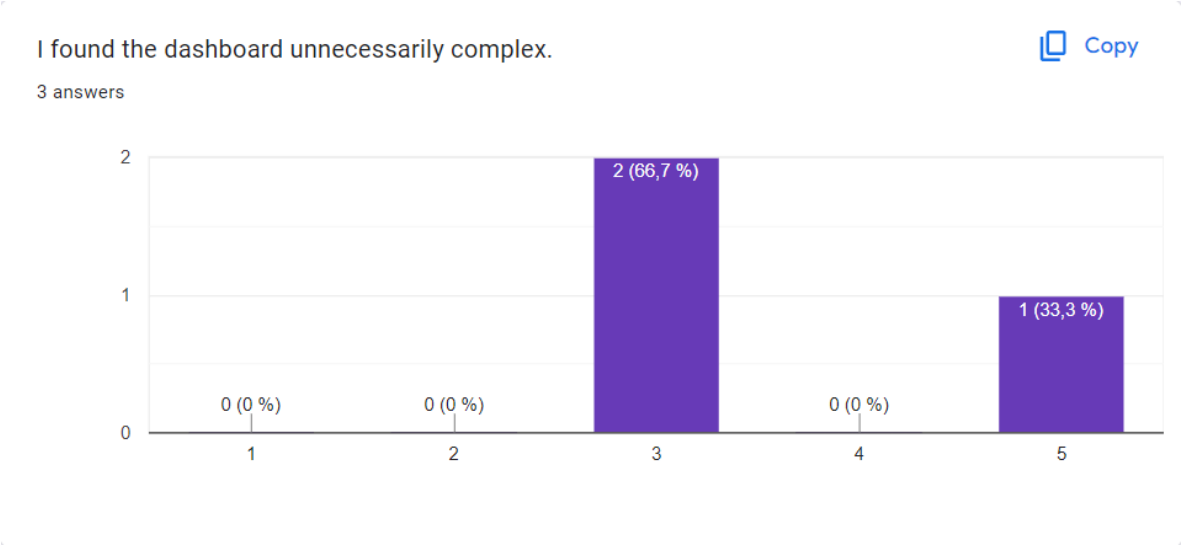

Figure B.0.2.: Usability question 2

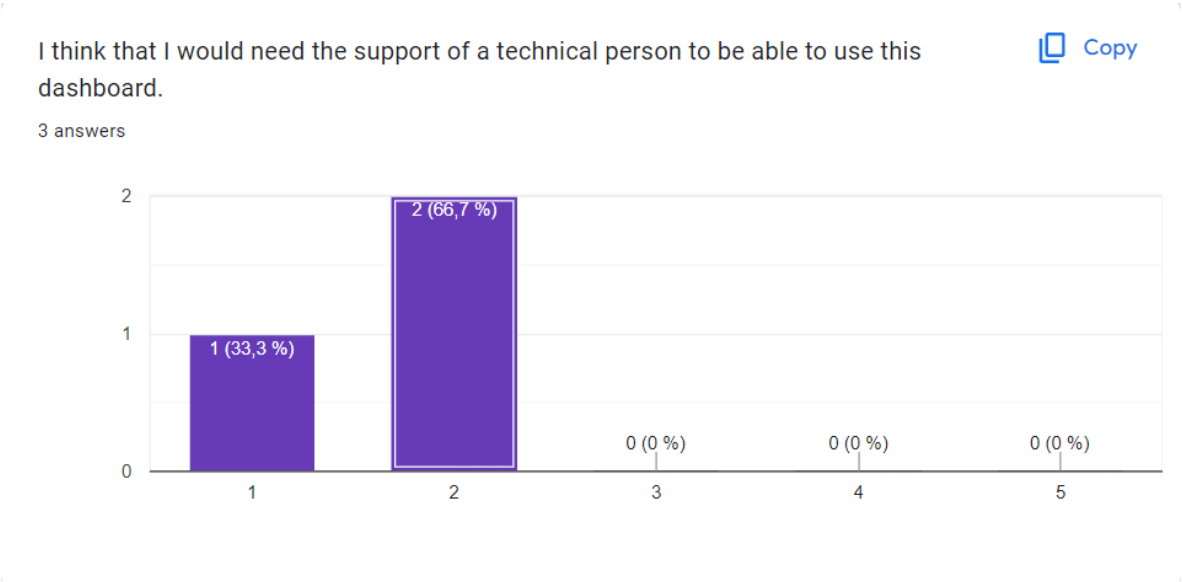

Figure B.0.3.: Usability question 3

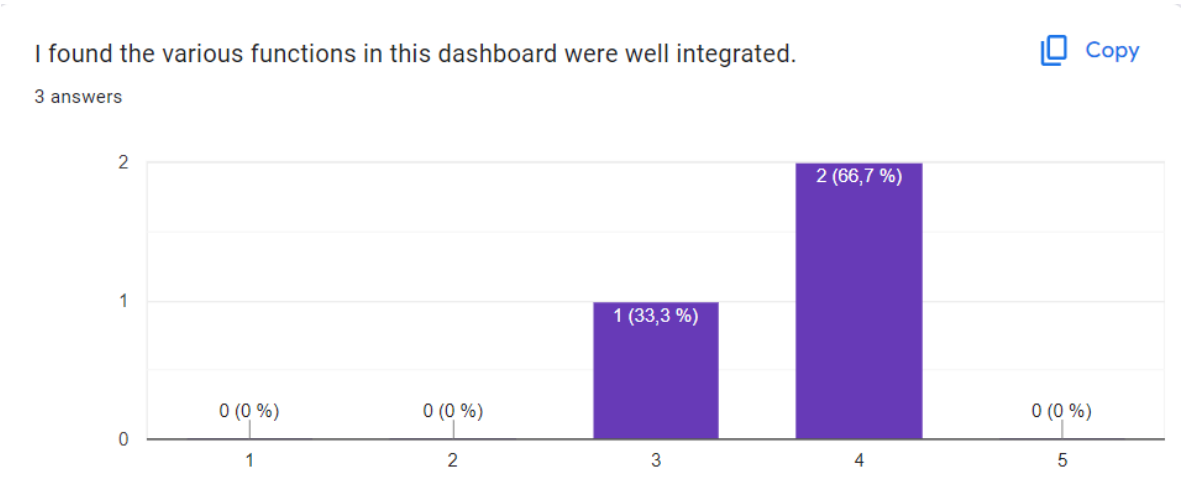

Figure B.0.4.: Usability question 4

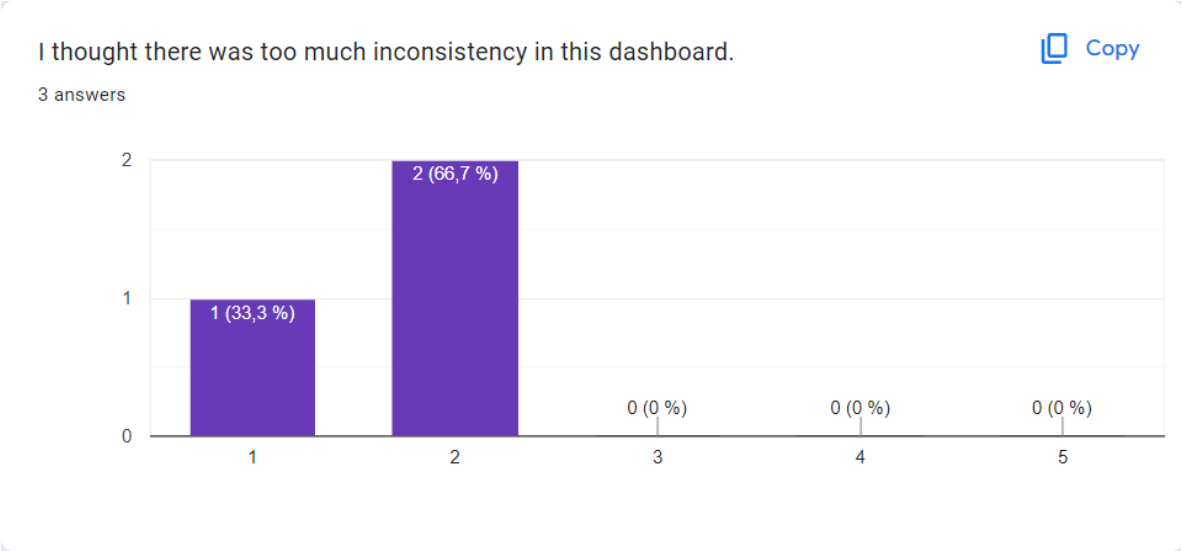

Figure B.0.5.: Usability question 5

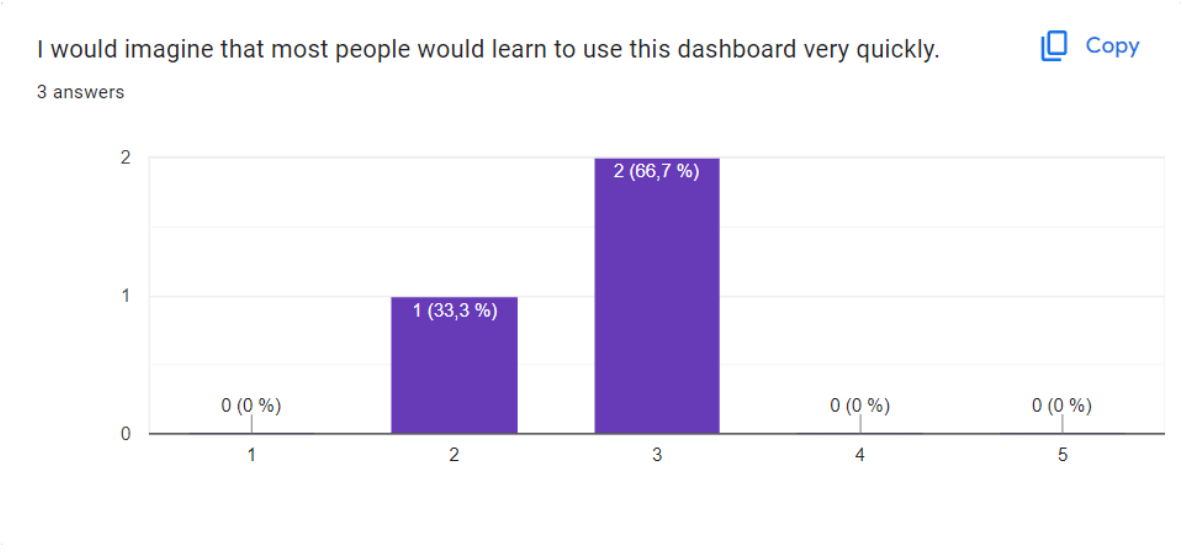

Figure B.0.6.: Usability question 6

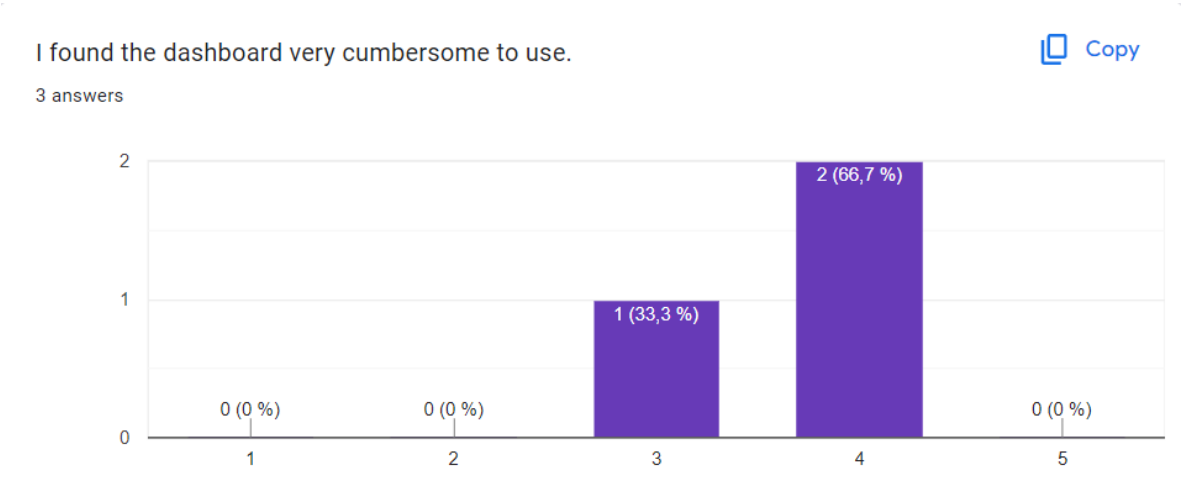

Figure B.0.7.: Usability question 7

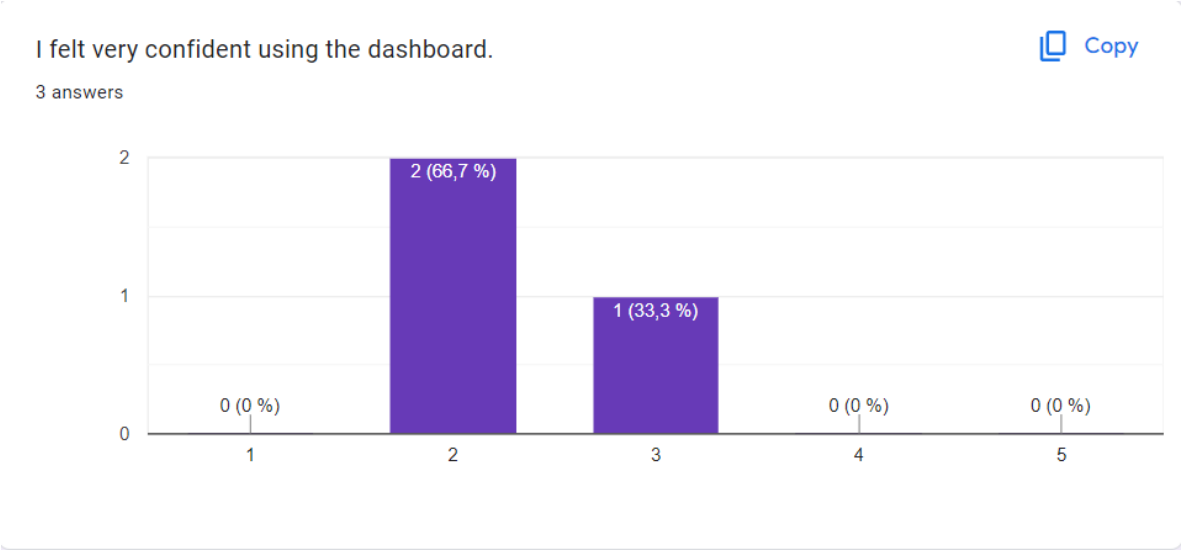

Figure B.0.8.: Usability question 8

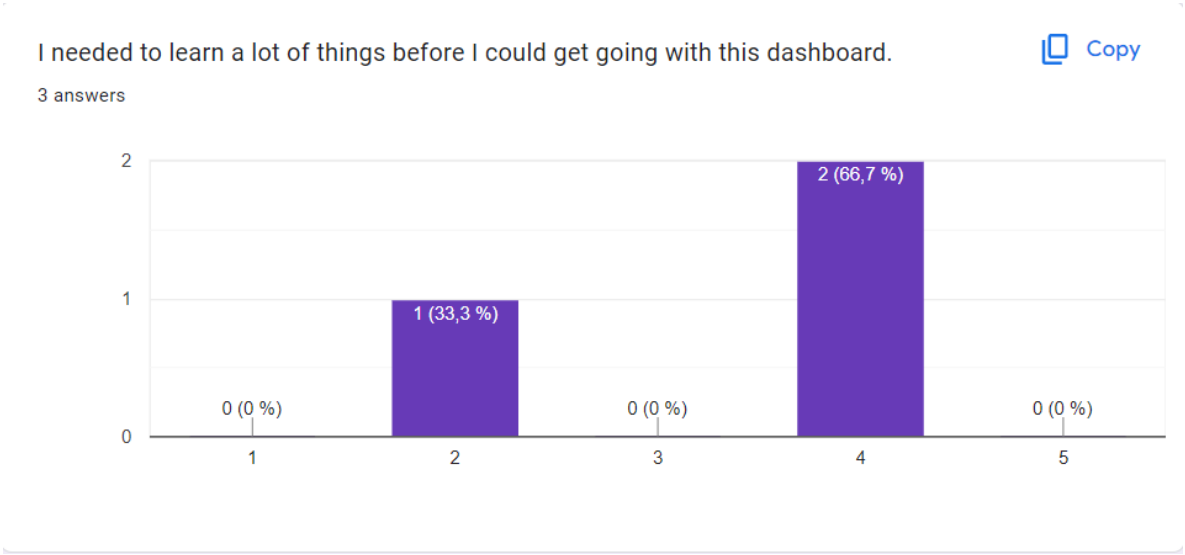

Figure B.0.9.: Usability question 9

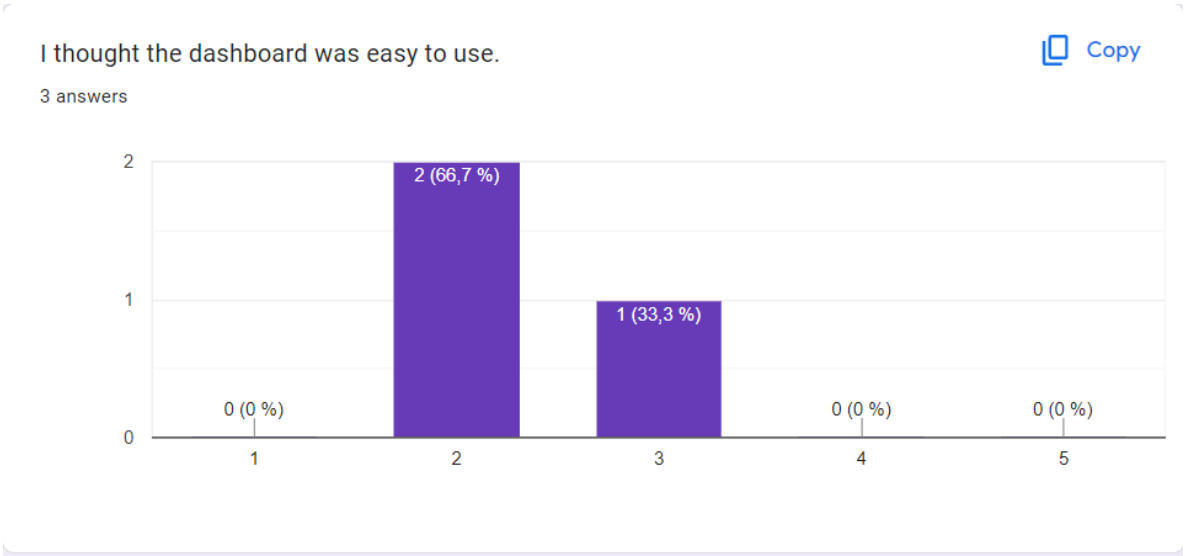

Figure B.0.10.: Usability question 10