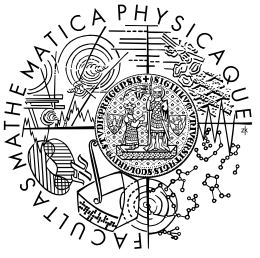

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Kryštof Hrubý

Semi-automated Data to Ontology Mapping

Department of Software Engineering

Supervisor of the master thesis: Mgr. Petr Škoda, Ph.D.

Study programme: Computer Science - Software and Data Engineering

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Petr Škoda, Ph.D. for always having time to discuss the thesis with me and for his helpful suggestions that let me stay on track. A huge thanks goes to my family who gave me a tremendous support throughout my studies.

Title: Semi-automated Data to Ontology Mapping

Author: Kryštof Hrubý

Department: Department of Software Engineering

Supervisor: Mgr. Petr Škoda, Ph.D., Department of Software Engineering

Abstract: The most valuable data publication is in the form of linked data. The creation of linked data includes in particular the transformation of data into RDF and the appropriate use of identifiers and ontologies. However, the entire process is still challenging even for linked data experts. Therefore, the main goal of this thesis is to create a semi-automatic solution to facilitate the transformation of data into high-quality linked data. The basis of the proposed solution is the creation of a data model that the user can interactively edit manually or with the help of recommendations. These recommendations suggest model transformations based primarily on expert knowledge of data domains. The solution was implemented as a proof-of-concept web editor and is based on the visual modification of the schema of user provided data.

Keywords: linked data, data catalog

Název práce: Semi-automatické mapování dat do ontologií

Autor: Kryštof Hrubý

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Petr Škoda, Ph.D., Katedra softwarového inženýrství

Abstrakt: Nejhodnotnější publikace dat je formou propojených dat. Tvorba propojených dat zahrnuje zejména transformaci dat do RDF a vhodné použití identifikátorů a ontologií. Celý proces je však dosud náročný i pro experty na propojená data. Proto je hlavním cílem této práce vytvoření semiautomatického řešení pro usnadnění transformace dat do kvalitních propojených dat. Základem navrženého řešení je tvorba modelu dat, který může uživatel interaktivně upravovat manuálně či s pomocí doporučení. Tato doporučení navrhují transformace modelu zejména na základě expertní znalosti domén dat. Řešení bylo implementováno jako proof-of-concept webový editor a je založené na vizuální úpravě schématu uživatelem vložených dat.

Klíčová slova: propojená data, datový katalog

Contents

1 Introduction

As the web of data grows, more data publishers are inclined to publish their data as 5-star linked open data [1]. Each star adds a further data quality requirement. The 3-star data [2] were always prevalent as the published type of data. The 3 stars only state that the data must be structured in a non-proprietary open format and available on the Web under an open licence. However, such data do not contain global and unique identification and cannot contain references to other data which is what the last two stars add. The fourth star states to denote things uniquely and globally (e.g. URIs) to allow others reference the data. The fifth star states to include links to related relevant data of others so that context is created.

However, to make the data understandable to not only to the authors, the meaning of certain URIs and links need to be defined. The term definitions are defined in vocabularies. Therefore, it is important to reuse already defined terms in existing vocabularies and not create new vocabularies (i.e. definitions) immediately. However, there are still challenges in the process of transforming data to linked data. One challenge can be to even find appropriate vocabularies for representing the given data. Sometimes even deciding whether a found vocabulary is even appropriate can be challenging due to the complexity of the vocabulary. Therefore, there has been a lot of research done on the topic of representing data using linked data and applications facilitate the process were created.

There are several existing applications (e.g. Linked Open Vocabularies [3] or domain Bio Portal [4]) providing term or vocabulary search that are mostly based on writing a text query where a user practically has to guess the right keyword(s) to find suitable terms or vocabularies. An alternative option is to use one of the standard web search engines. However, both ways require user to look at data to think of a query, use it in external environment to find vocabularies and study if matched vocabulary fits their use case.

Hence, the goal of this thesis is to propose an approach for facilitating the process of conversion of data to linked data by recommending vocabularies and their terms based on user provided data in one environment so that they do not have to shift between multiple ones. The basic idea is to provide an environment that would let user semi-automatically transform data to high-quality linked data. The user could combine interactive manual updates of the data based on the data schema with automatic recommendations. These recommendations would propose transformations of the schema and data to aid user in the transformation to linked data.

The structure of the rest of the thesis is following. In Chapter 2 we analyze the transformation of structured data to RDF as well as the existing methods and tools for vocabulary recommendation and structured data to RDF transformation. Based on this analysis, we propose our approach. In Chapter 3 we discuss the design of the solution based on the proposed approach. Afterwards, we describe the implementation of the solution in Chapter 4. In Chapter 5 we show how the solution can be used for transformation of structured data to RDF. In Chapter 6 we conclude.

1.1 Terms

Ontology, Vocabulary Ontology and Vocabulary terms are used interchangeably throughout the thesis and both represent a standard RDF vocabulary described in RDFS or OWL.

2 Analysis

In this chapter we analyze the process for converting structured data to RDF, tools and methods that aim to improve it. Based on the analysis we present the idea of our own approach. First, we showcase on real but simplified data how such a conversion can be done using manual means with available tools (Section 2.1). Then, we study the approaches for searching through linked data, recommending vocabularies and transforming data to RDF in Section 2.2 and Section 2.3. Lastly, we discuss our approach for facilitating the process of transforming data to linked data in Section 2.4.

2.1 Motivating Example

In the motivating example, we take food product data and convert them to RDF. The example data (shown in Figure 2.1) are taken from Open Food Facts [5] non-profit organization providing information about food products including ingredients, nutrition or environment impact of food production. Our example data is about a noodle food product which has a name, countries where it is sold and a shortened list of nutrients. The original data for each product have roughly a thousand lines in expanded JSON format with many properties incomprehensible to outside audience; therefore, it was substantially redacted.

```
[
    {
        "product": {
            "_id": "0737628064502",
            "product_name": "Thai peanut noodle kit",
            "countries": ["United States"],
            "nutriments": {
                "carbohydrates_100g": 71.15,
                "carbohydrates_unit": "g",
                "energy-kcal_100g": 385.0,
                "energy-kcal_unit": "kcal"
            }
        }
    }
]

```

Figure 2.1 Food Product Example Data

The manual approach of converting structured data to RDF is shown in Figure 2.2. It consists of searching for suitable vocabularies, deciding whether to reuse found vocabularies or create a new vocabulary and transforming data

to RDF. There could be additional steps such as publishing data on the web or performing linking task which we do not discuss here.

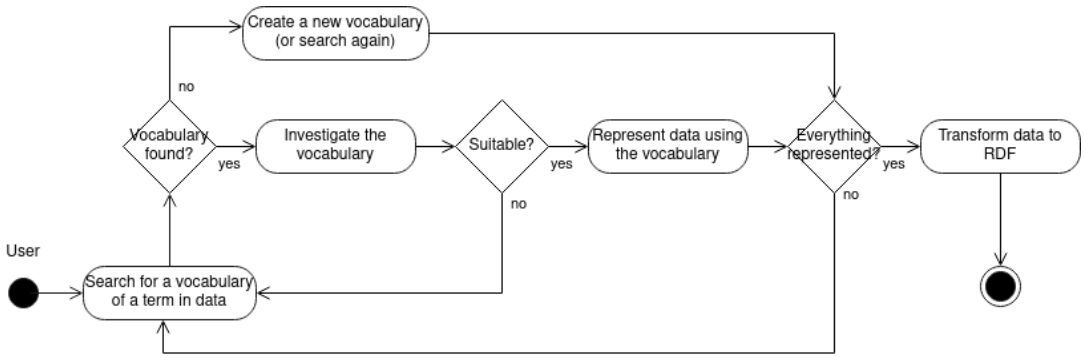

Figure 2.2 Data to RDF Transformation Process

2.1.1 Vocabulary Search

We tried searching for vocabularies using Linked Open Vocabularies (LOV) [3] and Google. We briefly describe each found vocabulary and its relevance to our example data. Searching LOV with queries such as *"food"* or *"carbohydrates"* yielded the following vocabularies. Longer queries were not effective.

SmartProducts Food Ontology (SPFOOD) [6] SPFOOD ontology defines classes for food products, food courses, nutrients, recipes, food ingredients and properties among them. While there are classes for product and nutrients and properties to interlink them, there seems not be a clearly defined way how nutrient values should be represented. For example, Carbohydrate term is derived from Nutrients term but both of them or any their super class have no properties for adding carbohydrates value or unit.

Food Ontology in OWL (FOWL) [7] FOWL is based on now unavailable wine ontology. While it has many properties to describe wine products, it lacks, for example, nutrients description; therefore, it is not fit for our use case.

LIRMM Food Ontology [8] Somewhat popular vocabulary for food which is not available but it is referenced in LOV, used to represent some observed food data and mentioned in general. It seems as a good match for our food example data based on partial information and observation, which is why it is mentioned as a possible relevant candidate.

Queries for Google needed to be longer than for LOV and include words such as *"vocabulary"*, *"ontology"*, *"RDF"* or there would be no relevant result. Moreover, more variations of queries and more result browsing was necessary to find vocabularies (apart from FoodOn [9] which is one of the first matches when searching for *"food ontology"*). Searching Google yielded the following vocabularies.

FoodOn [9] FoodOn vocabulary names all parts of animals, plants and fungai bearing food for humans or domesticated animals. Moreover, it contains any derived food products and processes used to make them. The primary objective of this vocabulary is to describe food while describing food products is secondary. Although we surmise that the vocabulary would cover the example data, it is quite complex for such simple use case. Moreover, since describing food products is secondary, finding representation for our food product with nutrients is difficult in their documentation.

AGROVOC [10] AGROVOC is a multilingual thesaurus spanning the area of interest of Food and Agriculture Organization of the United Nations. It names food products, nutrients, ingredients and the structural composition to great detail in multiple languages. There is similar issue to FoodOn that it is complex. Furthermore, since it is a thesaurus, there is no prescribed way how use defined terms. However, it makes sense provide links to it for multilingual support.

Food Ontology [11] This vocabulary is designed for food products. It is based on Good Relations [12] vocabulary. It describes terms for food products, ingredients as well as nutrient values and nutrient units. It does not contain much more but given our data it fits well. It is clear how our data should be represented using this vocabulary.

Schema.org [13] Schema vocabulary certainly contains terms for products and nutrition information. It is quite clear how our data should be represented using our vocabulary.

We believe that spending non-trivial time browsing FoodOn, AGROVOC and even SPFOOD might bring enough information to sufficiently represent our food data. Since there were two vocabularies that clearly have terms to represent our data with, we forwent such undertaking. Since Schema.org Energy[1] or Mass[2] terms used for nutrients specification have literal format of a value and an unit separated by a space, we chose the second to last food ontology.

Figure 2.3 shows how the part of Food Ontology which is relevant to our example data extends Good Relations. It defines subclass food:Food[3] of class gr:ProductOrService[4] and properties for nutrients which are subproperties of gr:quantitativeProductOrServiceProperty[5]. Nutrient values are then standardly represented using Good Relations gr:QuantitativeValueFloat[6] with units specified using UN/CEFACT Common Codes [14].

The rest of example data is an identifier property which is represented by schema:productID[7] and sold-in-countries property which a new property vocabulary term is created for for simplicity.

[1] `https://schema.org/Energy`
[2] `https://schema.org/Mass`
[3] `http://purl.org/foodontology#Food`
[4] `http://purl.org/goodrelations/v1#ProductOrService`
[5] `http://purl.org/goodrelations/v1#quantitativeProductOrServiceProperty`
[6] `http://purl.org/goodrelations/v1#QuantitativeValueFloat`
[7] `https://schema.org/productID`

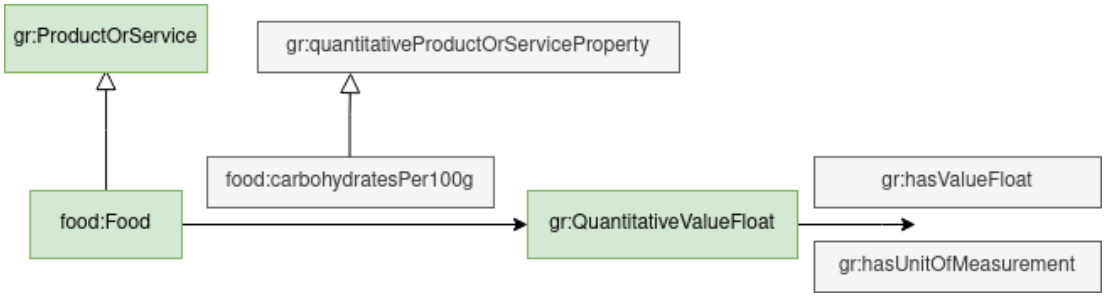

Figure 2.3 Food Ontology

2.1.2 Data Transformation

When suitable terms are known, the actual transformation to RDF can be done. For example, adding JSON-LD context to create a JSON-LD format or using transformation tools such as Easygen [15] or XSLT (using json-to-xml[8]). Figure 2.4 shows the RDF representation for the food data.

```
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix food: <http://purl.org/foodontology#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .

@prefix ex: <http://example.org/> .

ex:noodles a food:Food ;
    schema:productID "0737628064502"^^xsd:string ;
    gr:name "Thai peanut noodle kit"@en ;
    ex:soldInCountries <http://publications.europa.eu/resource/authority/country/USA> ;
    food:carbohydratesPer100g [
        a gr:QuantitativeValueFloat;
        gr:hasUnitOfMeasurement "GRM"^^xsd:string;
        gr:hasValueFloat 71.15
    ] ;
    food:energyPer100g [
        a gr:QuantitativeValueFloat;
        gr:hasUnitOfMeasurement "K51"^^xsd:string;
        gr:hasValueFloat 385.0
    ] .

```

Figure 2.4 Noodle Food Product in Food Ontology

2.1.3 Conclusion

In this subsection, we conclude the individual steps were taken to transform the example data to RDF.

The example presented what a data publisher might have to do to publish their data. As mentioned in the introduction, our aim is to make this process easier. We had to formulate text queries somehow describing our data to either LOV or Google to find any related vocabularies. It was necessary in both cases to understand how to create queries to find relevant results (vocabularies). Possibly more convenient approach would be to upload data to an application and it would recommend vocabularies or their terms based on the given data.

[8] https://www.w3.org/TR/xslt-30/#json

We found seven relevant vocabularies. After spending non-trivial time reading the vocabularies (or their documentations if they had any), we ruled out all vocabularies found using LOV. It was necessary to use a standard web search engine such as Google to find suitable vocabularies for our use case. Investigating whether vocabularies are suitable for a given use case can be quite demanding; therefore, more focus on whether any vocabulary is suitable while recommending it might serve decrease user's investigation time and make the overall experience more pleasant.

After suitable vocabulary terms were found, it was time to do the actual transformation. It required having some non-trivial knowledge of transformation tools or JSON-LD since input data were in JSON.

We had to use different software to search for vocabularies, then inspect them manually and use different software for the transformation itself. It might again be convenient to be able to perform all of the above within the confines of a single software.

2.2 Existing Methods

In this section we discuss the existing methods for recommending vocabularies or their terms. Since there are not many published methods for recommending vocabularies, we include methods for searching through linked data which can serve as a part of a recommendation method. While some of the manuscripts summaries might seem somewhat too detailed with respect to the analysis chapter, they are referenced from implementation parts of this thesis.

2.2.1 TermPicker

TermPicker [16] is a method for recommending vocabulary terms to reduce term heterogeneity in published data. The method requires a partially modeled data as an input and recommends types and properties based on how similarly modeled data on Linked Open Data (LOD) cloud are already represented.

Schema Level Patterns

To capture how RDF data are modeled, the authors define Schema Level Patterns (SLP). SLP is a triple of three sets $(\{sts\}, \{pts\}, \{ots\})$ where *sts* is a set of subject types, *ots* is a set of object types and *pts* is a set of properties linking subject resources (with *sts* subject types) to object resources (with *ots* object types). The following SLP is created from the example data in Figure 2.5 which model a person who is also a chess player that knows another person who is also a coach.

```
(
    {foaf:Person, dbo:ChessPlayer},
    {foaf:knows},
    {foaf:Person, dbo:Coach}
)
```

```
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.com/> .

ex:sports1
    a foaf:Person, dbo:ChessPlayer ;
    foaf:knows ex:sports2 .

ex:sports2
    a foaf:Person, dbo:Coach .
```

Figure 2.5 Example RDF data for modeling that a chess player knows a coach.

SLPs of RDF data can be generated using two hash tables. The former hash table $M1$ contains a mapping from resources (i.e. subjects and objects) to the set of their RDF types. The latter hash table $M2$ contains a mapping from subject and object pairs to the set of all properties between them. SLPs are then constructed by iterating over $M2$ using resource type information from $M1$.

Recommending Vocabulary Terms

TermPicker takes an input SLP which represents a part of the currently modeled data and considers a recommendation candidate which can be any term from LOD cloud that can be a part of any set in SLP (e.g. property or type). TermPicker computes feature values for all candidates and subsequently ranks the candidates by a ranking model.

There are five feature values. The first three are based on popularity of the candidate in LOD cloud: a number of occurrences, a number of datasets with the candidate, a number of datasets with the candidate's vocabulary. The fourth feature considers whether the candidate is from vocabulary present in the input SLP. The last feature is called the SLP-feature and calculates how many SLPs in all of SLPs in LOD cloud are supersets of the input SLP with the recommendation candidate added to it. SLP A is a superset of SLP B if all sets of A are supersets of the corresponding sets of B.

Training And Evaluation

The ranking model is based on Learning to Rank (L2R) [17] algorithms. These algorithms are supervised machine learning algorithms aiming to create a ranking model from a set of given features. The training and evaluation of various ranking models is done via simulating term recommending.

A SLP from train data is taken and one or more terms are randomly extracted from it to create an input SLP for training. The extracted terms serve as relevant candidates for the input SLP. All other candidates are ignored. Therefore, a L2R algorithm is provided with input data SLP and a set of recommendation

candidates with all five features computed as well as relevance information to train on. The evaluation is done using 10-fold leave-one-out.

The best performing algorithm in the evaluation was Random Forests L2R algorithm.

2.2.2 Karma

Karma [18][19] is a system for semi-automatically creating mappings from structured data to RDF in a target ontology. Initially, a user has to provide the target ontology and source data. The whole process consists of three main steps: Semantic Type Inference, Graph Construction and Source Model Refinement. At any point the user can manually intervene to override some Karma's decisions which results in the system recomputing the steps with the user's change. Note that a column of data refers to an array of literals of a property of an entity (e.g. a column in a relational database).

Semantic Type Inference

Karma automatically assigns a semantic type to each column of data based on the semantic types learned from previous modeling sessions. A semantic type captures the meaning of the column data. It is either an OWL class in case of the column being an automatically generated database keys or a pair of a data property and an OWL class for columns with meaningful data. The OWL class in that case represents the domain of the data property.

If any assigned semantic type is incorrect for a column, the user can manually set the correct semantic type from provided options from the target ontology for the column. This action triggers the system to remember the assignment of the correct semantic type along with the column data and use it in training to be able to recognize the semantic type in future. Semantic types are trained using Conditional Random Fields (CRF) [20].

Graph Construction

Once all semantic types are assigned, a graph representing all possible mappings from the source data to the target ontology is constructed. First, a node is created for each semantic type representing a column. Afterwards, the ontology is searched through and nodes are created for all classes which there exists a path to a semantic type for. These paths comprise of properties or isa relationships. The last step revolves around adding links between nodes. A link is added between two nodes if there is a datatype property, an object property or an isa relationship between their corresponding classes in the target ontology.

Source Model Refinement

Since the graph represents all possible mappings, it must be refined by creating a source model by selecting a subgraph connecting all semantic types. Karma uses heuristic Steiner Tree algorithm variant [21] to find the minimal tree connecting all semantic types. The minimal tree might not be the desired subgraph; therefore,

Karma allows the user to specify some constraints on the algorithm in order to change the tree to the desired state.

RDF Generation

When the user is satisfied with the found tree, they can instruct Karma to generate the corresponding RDF.

2.2.3 Swoogle

Swoogle [22] is a search engine for Semantic Web Documents (SWD). A user provides a keyword query and matching SWDs are returned in a ranked order. SWDs are online documents containing semantic annotations (RDF) and having references to other SWDs. SWDs are either classified as Semantic Web Ontologies (SWO) or Semantic Web Databases (SWDB). A document is a SWO when it defines a proportionally significant number of new terms or extends the definitions of other terms. Otherwise it is classified as a SWDB.

For obtaining SWDs, Swoogle implements a web crawler which utilizes Google to get documents with high probability of having RDF triples and then recursively crawls other referenced SWDs. Since the crawler cannot parse all web documents, it employs special heuristics for determining if a file is SWD (e.g. considering only extensions such as *".rdf"*, *".owl"*, *".n3"*) and creating queries for Google in order to skip parsing documents with no semantic annotations and obtaining as many SWDs as possible.

Having crawled the SWDs, Swoogle uses a an inverted TF/IDF model with standard cosine similarity to index any SWD. SWDs are indexed as bags of URIrefs and n-grams.

Results for a query are ranked based on modified Google PageRank [23] algorithm. Swoogle takes into account the kinds to links between SWDs and treats them differently. What is important that the results are ranked by popularity and SWOs are typically ranked higher than SWDBs since they are linked from many other SWDs.

2.2.4 Falcons

Falcons [24] is a search engine for objects (i.e. entities with URI) as opposed to aforementioned Swoogle which provides searches for SWDs. A user gives Falcons a keyword query and Falcons provides a ranked list of objects along with a short structured snippet showing corresponding literals and linked objects. Falcons does class inferring for every object and tracks class hierarchies; therefore, query results can be refined by user navigating class hierarchies.

Indexing

Falcons system implemented a crawler to obtain RDF data. The input URIs of the crawler were either URIs of potential RDF documents acquired using Google and Swoogle search engines as well as manually set URIs.

Having obtained input RDF data, Falcons system creates for each object with URI a virtual document that includes its local name obtained from its URI, associated literals and textual descriptions of associated predicates and objects.

In order to more precisely define virtual documents, the authors of Falcons work with a notion of *RDF sentence*. Two triples are *b-connected* if both contain a common blank node. *RDF sentence* is a maximum subset of *b-connected* triples. Virtual document of an object *o* then contains all local names of URIs of objects and predicates as well as all literals which are part of *RDF sentences* where *o* is a subject. All such local names and literals are weighted. The largest weights is assigned to local names and literals of object *o*.

Virtual documents are then stored in an inverted index (terms to virtual documents index) using Apache Lucene [25]. The authors do not mention the exact configuration of the index, only that terms from virtual document with different weights are indexed in different fields so that different weights can be applied when queries arrive.

Querying

When a query arrives, relevant virtual documents are retrieved from the index based on the query terms, which are considered in conjunction. Such virtual documents are ranked based on the relevance to query and their popularity. Query relevance is measured by the cosine similarity of the resulting virtual document and the query. Popularity of a virtual document is measured by the number RDF documents where the virtual document occurs. The final rank score is product of both measures.

Rather than returning the virtual documents themselves to user, short snippets are generated and returned to user instead. These shorts snippets consist of *Property Description Threads (PD-thread)* which are paths leading from a virtual document to either a literal or an object with URI. PD-thread path can contain distinct blank nodes. For each virtual document to return to user, its PD-threads are computed and ranked by their cosine similarity to query terms. Top three PD-threads are then returned to user.

Refining Results

We mentioned that user can refine these results by navigating class hierarchies. Falcons system actually creates another inverted index from classes to objects; therefore, if any (sub or super)class is specified along with the keyword query, both indices are searched and only the intersection of resulting objects (resp. virtual documents) is considered for ranking. To create such an index, Falcons not only considers explicit class assignment (e.g. *ex:o a rdfs:Class*) but computes implicit reasoning by finding all superclasses for each class of an object.

2.2.5 Keyword Search over RDF Using Document-Centric Information Retrieval Systems

The manuscript [26] whose name is in the heading discusses approaches for keyword search through RDF data using a standard document-centric information retrieval system. They also study how such system compares to dedicated keyword

search systems for RDF. The authors pick Elasticsearch [27] as representative of such a system. Their proposed method is similar to those aforementioned methods. Index data using inverted index in Elasticsearch and when a keyword query comes, use the index to find matching RDF data and return them ranked.

Basic Challenges of Keyword Search over RDF Data

The authors present the basic challenges of keyword search over RDF data. First, the authors debate how to select *Retrieval Unit* which is what conceptually represents an indexed document and is retrieved by search. There are three main options: An Entity with URI, A Triple, A Subgraph.

Having entity as a retrieval unit can satisfy entity search information needs which are related to retrieving one or more entities. Property as a retrieval unit contains more information and can satisfy entity searches for a property of an entity. A Subgraph is the most complex retrieval unit satisfying more complex searches.

Second, the authors discuss how to index data based on chosen retrieval unit. If the retrieval unit is an entity, its URI and properties can be indexed. If it is a triple, its URIs and both subject and object properties can be indexed. For a subgraph, the authors mention that an inefficient option is to index all subgraphs of a given size. They also propose to index triples instead and select the triples forming a subgraph of a given size during the retrieval process.

The last two challenges are weighting of index fields and ranking results.

Experiments

The authors perform a series of experiments on a DBpedia-Entity test collection for entity search [28]. All experiment apart from the last one were run on a slightly reduced version of the test collection. The authors do not perform experiments with different retrieval units or different ways of ranking results. They choose a triple to be a retrieval unit due to it being more informative than an entity and being a simplest representation of a fact in RDF data. It is also quite flexible in terms of structuring final results, for example, for using aggregation methods to provide a ranked list of entities in an entity search.

Results are first ranked according their scores from Elasticsearch. The triples are then grouped by entities (i.e. subject and object URIs) and these entities are ranked by a discounted sum of its triples' scores and returned.

While the chosen part of a triple indexed are different in experiments, the URI is tokenized into keywords to get a local name and other parts of the URI and indexed.

As for the configuration of Elasticsearch the authors choose for the following experiments below the default unless stated otherwise in an experiment.

Field Separation Experiment The first experiment aimed to discover what parts of a triple and how field separation influence the quality of results. The authors created five different indices indexing only URIs of only subject, only predicate, only object, the whole triple in one index field and the whole triple where each URI was in separate fields. The best results were produced when all fields were indexed and the single field index slightly outperformed

the multi field index. The authors note that the index with only object indexed outperformed the index where only subject was indexed. Moreover, the performance of the index which indexed only property was exceedingly low.

Field Weighting The experiment examines whether boosting the importance by raising the weight of a field can improve performance. The authors took the index which indexed the whole triple URIs into distinct fields and created three indices with doubled doubled weights of only subject, only object and both. Doubling the importance of a subject dropped the performance by 10% to the baseline with no weight doubling while doubling the importance of an object slightly increased the performance. Therefore, the authors concluded that object keywords were more useful than subject keywords for the tested collection.

Extending Index In this experiment, the authors added fields to index containing the values of properties of a subject and an object. The first index included *rdfs:label*, the second *rdfs:comment* and the last all outgoing properties. All these indices had object weight doubled and were compared to the index without property values with doubled object weight. Adding *rdfs:comment* increased performance by 5% while adding all outgoing property values slightly decreased performance. The index with *rdfs:label* had the same performance as the index without property values which made sense because labels of DBpredia are often the same as the local names in URIs.

Similarity Model In this experiment, the authors compared the performance of the different similarity models[9] provided by Elasticsearch on their default settings. The performance was measured on the best performing index (i.e. index with double object weight and comments) with BM25, DFR, LM Dirichlet and LM Jelinek Mercer similarity models. Models BM25, DFR and LM Jelinek Mercer had similar performance with LM Jelinek Mercer slightly outperforming the rest.

Comparison Experiment with DBpedia Specific Methods In the last experiment, the authors compared the indices with different similarity models to other external methods ran on the test collection. This experiment was run on the full test collection. The best performing model was BM25 out of the methods proposed by the authors and the authors claim that it was very close in terms of performance to the best external method DBpedia-Entity-v2 SDM [28].

2.2.6 LOD Search Engine

LOD search engine [29] is, as the name implies, an engine for searching in a large amount of RDF data. A user provides a keyword query and gets matching triples in a ranked order. Apart from providing the query the user can choose between Forward or Backward search method.

[9] `https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-similarity.html`

Indexing

The data used in the paper are the Linked Open Data (LOD) cloud from 2014 [10]. The authors used the knowledge that the cloud was split into nine main domains and identified 1014 distinct URIs related to the domains. These distinct URIs were used to split the whole cloud to 1014 distinct files. These files were loaded to Apache Lucene [25] to create an index. The main purpose of the index is to for an input word return all files containing the word so that relevant files can be quickly identified when a query arrives.

Querying

Each query is first processed using Brill's tagger [30] to lemmatize it and each lemmatized word is assigned a Part of Speech (POS). These POS tags are used to create keywords used for searching in the index. Keywords are either single words or phrases (multiple words). A phrase means adjacent two or more nouns, adjectives or verbs.

Relevant files and their data are then retrieved based on these keywords which are then further searched through to get relevant triples. This subsequent search is done either with Forward or Backward search. Forward search returns triples that contain given keywords in the subject of a triple. It is meant to be used when user knows triple subject and is looking for associated objects. On the other hand Backward searches objects and is meant to retrieve subjects.

Both search methods return all matched triples which are then ranked by Domain and Triple ranking. Domain ranking ranks domains based on their the number of matched triples. Triple ranking ranks triples in one domain based on the number of occurrences of either subjects or objects. If Forward search was performed, the number of objects linked to a given subject of a triple is counted to rank the triple. The number of subjects is computed for Backward search instead. The more occurrences, the higher rank a triple is given.

Evaluation

The engine was compared with Swoogle [22] and Falcons [24] using mainly Precision, Recall and F-Measure. The authors claim that the LOD search engine outperformed both significantly with respect to these measures.

2.3 Existing Tools

In this section we first study the existing software for a vocabulary search. While that primarily means a vocabulary search, we include also catalogs with RDF data which can also help user to represent their data using RDF. Afterwards, we discuss the implementations of the methods from papers and whether they are still usable. Lastly, we study tools for transformation of structured data to RDF. While we touched on these topics in the previous section when discussing the existing methods, we focus here more on the technical and user points of view.

[10] `https://data.dws.informatik.uni-mannheim.de/lodcloud/2014/ISWC-RDB/`

2.3.1 Vocabulary Search Software

In this subsection we describe the existing software for searching for vocabularies and their terms.

Linked Open Vocabularies

LOV [3] provides vocabulary and term search based on an internal catalog of curated vocabularies ranging from general purpose ones to domain specific. Only vocabularies following best publishing practices and meeting the standards of LOV can be added. These include, for example, URI stability and availability as well as the quality of metadata and documentation. The search engine indexes all vocabulary terms and provides a full-text search. The results ranking is based on term popularity among datasets and on the category of term label properties matched. For example, a match for `rdfs:label` is more important than `rdfs:comment`.

Moreover, for data access LOV provides a SPARQL endpoint or data dump as well as a public API of all the services provided in its user interface.

BioPortal

BioPortal (software [4], original paper [31]) software is a repository of biomedical ontologies. It contains roughly a thousand biomedical ontologies which comprehensive metadata are provided for. The following list includes the main parts of metadata shown for each ontology:

- name, description, acronym, status
- category (domain)
- the number of visits on the BioPortal web page
- projects using the ontology
- the number of various terms such as classes and properties
- versioning and publishing information
- mappings to other ontologies

It also provides a user friendly browser of vocabulary terms as well as graph visualizations of class relationships.

BioPortal provides the following services related to search.

Ontology Browser Ontology browser enables browsing of ontologies with filtering capabilities based on the aforementioned metadata.

Class Search Class search is based on user providing a text input describing a class (e.g. name, synonym, id) and optionally on a few advanced settings such as filtering results by domain category or ontologies. Results are linked to the aforementioned browser of vocabulary terms.

Ontology Recommender Ontology Recommender [32] recommends ontologies based on a list of keywords or an excerpt from a biomedical text. Ontologies are ranked by a weighted sum of four criteria: Coverage, Acceptance, Detail and Specialization. Coverage represents the extent of an ontology covering the input data. Acceptance represents how well-known and trusted an ontology is. It is computed partially from the number of its visits on the web page. Detail represents the level of detail of ontology classes covering the input data. Specialization represents how much an ontology is specialized with respect to the domain of the input data. The user can manually override default weights for these criteria to adjust the ranking.

BioPortal also offers API [11] to the aforementioned services and data for logged users.

EMBL's European Bioinformatics Institute Ontology Search

EMBL-EBI Ontology Search [33] software provides ontology and term search on a catalog of hundreds of biomedical ontologies. Based on a brief evaluation there is an ontology overlap with BioPortal. EMBL-EBI Ontology Search provides a text input based search for ontology terms with an option to filter by ontologies. No documentation of how the search is done was found.

Moreover, there is an ontology browser with an optional text input filter. The software shows some metadata for each ontology such as numbers of terms, a description, contributors, ontologies importing the ontology and ontologies the ontology imports.

Similarly to BioPortal EMBL-EBI Ontology Search software contains a browser for vocabulary terms with class relationship graph visualization, class hierarchies and related terms.

2.3.2 Data Catalogs

Apart from software for searching for vocabularies it might useful for user to browse a catalog of RDF data when transforming data to RDF. Finding similar use case and seeing how such data are represented can be a way to find a vocabulary to use. Or the user can find already published data they might link their data to (e.g. countries, cities, organizations, ...).

Since there are many such catalogs, we only provide a few examples that will be referenced later on in the thesis.

Czech National Open Data Catalog

Czech National Open Data Catalog hosted on Czech National Open Data Portal [34] contains metadata records about open datasets in Czechia. It contains both records registered directly into it and records of more local Czech catalogs. Metadata records are specified using DCAT-AP-CZ [35]. The user might, for example, want to reuse the URIs from the code list of Czech districts [12] or link to them.

[11] `http://data.bioontology.org/documentation`

[12] `https://data.gov.cz/dataset?iri=https%3A%2F%2Fdata.gov.cz%2Fzdroj%2Fdatov%C3%A9-sady%2F00551023%2F04e0a699be153c780a0dde2c38dc3b13`

European Data Portal

European Data Portal [36] contains its own controlled datasets and metadata records from national catalogs. For example, there are published code lists for currencies[13] and countries[14].

2.3.3 Transformation Tools

We now survey the programs for transforming structured data to RDF. We first list programs that support multiple formats of structured data and then list programs supporting one format by the formats.

Karma

Karma [37] system and its underlying method for enabling creating mappings from structured data to RDF in a target ontology was discussed in Subsection 2.2.2. We now inspect the more practical aspect of the software mainly from the user point of view. Any ontology that the user would like their data be represented in must be loaded as a file to Karma. Karma supports loading source data from relational databases, CSV, XML, JSON or spreadsheets. Figure 2.6 shows how karma presents our example food JSON. It shows the hierarchical schema with the food product being at the top. The literal values (columns) of the data are shown below the hierarchical view. Above, Karma shows the initial model of the source data (red nodes). Karma does not take the hierarchical structure into account and instead considers the model to be only unconnected columns.

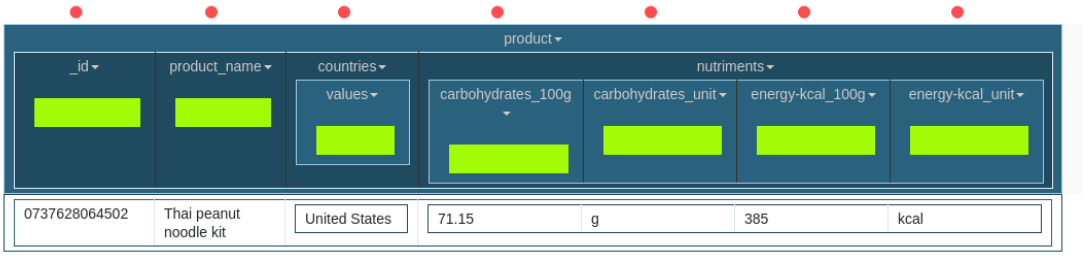

Figure 2.6 Food Data in Karma

The transformation workflow consists of clicking on the graph nodes and selecting properties going to or from these nodes. On clicking on a node, Karma shows a dialog with recommended properties which the user can select. If recommending properties are not fitting, the user can manually choose a property (and a source or target class) from target ontology. This is how a graph of source data specified using the target ontology terms is created.

Karma also provides quite rich data manipulation options. It is possible to create new columns or transform an existing one by writing a python script which has access to the value of all columns. Moreover, the user can split or merge columns or even aggregate their data.

[13] `https://op.europa.eu/en/web/eu-vocabularies/dataset/-/resource?uri=http://publications.europa.eu/resource/dataset/currency`

[14] `https://op.europa.eu/en/web/eu-vocabularies/dataset/-/resource?uri=http://publications.europa.eu/resource/dataset/country`

When user is satisfied with the created model, it can be exported to various RDF formats.

Note that while the tool is actively maintained based from Github commit history[15], the documentation is quite dated.

Silk

Silk [38] framework is mainly focused on creating links between different RDF datasets. Silk supports selecting which entities from two datasets to link using simple SPARQL-like path language. To link two entities, user creates a pipeline computing a single confidence value based on which these entities are linked or not. The pipeline typically retrieves and optionally transforms related data of both entities and compares them using a similarity metric.

While linking is a part of the process for publishing RDF data, we are more interested in the transformation of data to RDF which comes before linking. Silk supports loading data from a non-RDF source such as JSON. The user converts such data to RDF by specifying object and value mappings. An object mapping is used for creating a RDF resource and a property connecting it to already created resources. A value mapping is used for creating literal properties and their values for already added resources. A visual pipeline collecting, transforming and aggregating data from various sources from the original data can be constructed to compute literal values. Silk also provides an option to load vocabularies and suggest terms from them. Moreover, URIs of resources can be generated by a pattern.

While Silk is maintained based on its Github commit history[16], a documentation for this entire transformation to RDF process is missing.

Linked Pipes ETL

Linked Pipes ETL [39] enables users to create general linked data transformation pipelines. A pipeline consists of interconnected components. There are the following basic types of components.

Extractors Extractors are responsible for fetching data to a pipeline from external places. For example, data can be downloaded using HTTP GET or loaded from a file system.

Transformers Transformers perform transformations on data in a pipeline. These components are where user can transform structured data to RDF. Tabular component transforms CSV to RDF in a standard generic way. There is a component for adding a JSON-LD context to JSON file. For XML, there is support for running XSLT scripts. Transformers also include components capable of running SPARQL to transform data in a pipeline.

Quality Assessment Components Quality assessment components check if data in a pipeline meet predefined criteria.

[15] `https://github.com/usc-isi-i2/Web-Karma`
[16] `https://github.com/silk-framework/silk`

Loaders Loaders are responsible for loading data from a pipeline to external locations such as a file system or a database.

The user creates a pipeline by assembling a visual graph of these interconnected components. Linked Pipes ETL also offers more functionality such as sharing component configurations among pipelines or allowing users to define custom components.

Relational Databases to RDF

RDB to RDF Mapping Language (R2RML) [40] is a language for specifying mappings from relational databases to RDF. These mappings can be specified in RDF using R2RML custom vocabulary. A R2RML mapping is based on Triples Map which is a rule which maps each row in a logical table to a set of RDF triples. A logical table is a physical table, view or SQL query. Triples map consists of two parts:

Subject Map Subject map specifies the subject of RDF triples generated from one row.

Predicate-Object Maps Multiple Predicate-Object maps specify predicates using Predicate Map and objects via Objects Map. Objects map can specify a column name which literals are taken from. Object resources are specified either by defining a join operation with another triples map or by creating URIs as patterns of strings and columns.

Alternatively, user can use simpler Direct Mapping (DM) [41] to perform a generic transformation from relational database to RDF without any need to create mappings. However, the resulting RDF has no vocabularies used and all URIs are based on names of tables and columns. Such RDF data typically need further processing using, for example, SPARQL CONSTRUCT query.

CSV to RDF

Tarql [42] tool is capable of running SPARQL queries on CSV files. Namely, SPARQL CONSTRUCT query can be used to transform CSV to RDF.

Alternatively, tools following the specification for a generic transformation of tabular data to RDF [43] can be used to create RDF in a similar raw way as in using Direct Mapping for relational databases. This generic transformation turns rows into RDF resources and columns into their properties whose URIs are based on column headers.

JSON to RDF

If source data are in JSON, no special tool is required to convert the data to JSON-LD [44]. It is only necessary to specify JSON-LD Context so that the original JSON file is interpretable as RDF. We only mention this option briefly since we expect the reader to know JSON-LD serialization.

XML to RDF

For completion on the main data formats, XML data can be transformed to RDF by standard XSLT [45].

2.4 Proposed Approach

In the previous sections in this chapter, we studied the process of transforming structured data to RDF. Now we propose our approach to improve the experience of users undertaking such process. We identify the following three challenges related to the process which we discuss below in detail. The first challenge is that there is currently no single system which would let users transform data to RDF while providing vocabulary recommendations from vocabularies unknown to the users. The second challenge is that while the discussed recommendation and search methods are quite general and can be used on any data, using them can lead to a number of unusable term recommendations which require a user's manual investigation. The last challenge lies in that while all tools implement only their proposed methods for search or recommendation, there is no software that would support adding multiple methods out of the box.

These challenges with along our proposal how to resolve them are discussed in the following subsections. Lastly, we summarize the proposals to convey our approach in a more concise way.

2.4.1 Challenge One - No Single System for Transforming and Recommending

Consider a user with some structured data which they want to transform to RDF. If they know what vocabulary to represent the data in, they can use interactive Karma [19] (see Subsection 2.3.3) software or any other transformation tool to perform the transformation. However, when the user does not know a target vocabulary, there is no tool which the user could load the data into and it would recommend relevant target vocabularies (or their terms).

While both Karma and Silk [38] (see Subsection 2.3.3) provide term recommendations when transforming the structured data, they only do so from known vocabularies provided to the programs by the user. Other described transformation tools are focused on completely manual transformation without taking account of vocabularies.

In contrast, using vocabulary search tools such as LOV [3] or BioPortal [31] web applications (see Subsection 2.3.1) requires the user to create keyword queries based on the input data. While these tools curate the vocabularies and provide advanced search capabilities, they are not capable of search based on structured data nor provide transformations based on search results.

Moreover, sometimes using standard web search engine such as Google is required to find the most relevant vocabulary as it was in our food example where only Google found fitting vocabularies (see Subsection 2.1.1).

Our System for Transforming and Recommending

To summarize, such user always needs to shift between a place where they search for vocabularies and a place where they transform. Therefore, we aim to provide a system which is capable of both recommending vocabularies based on structured data and and data transformation.

In addition, while recommending vocabularies and their terms by itself is useful and letting the user transform data manually, we consider a vocabulary recommendation to also propose a way how transform the current version of data and execute the transformation on user accepting it.

Rather than focusing on creating a recommendation which is capable of transforming the whole data to the target vocabulary and which might not be even feasible, we focus on smaller or even unit recommendations considering only a part of the data. For example, we mean a recommendation which recommends to set the URI of a property in the source structured data to a vocabulary property.

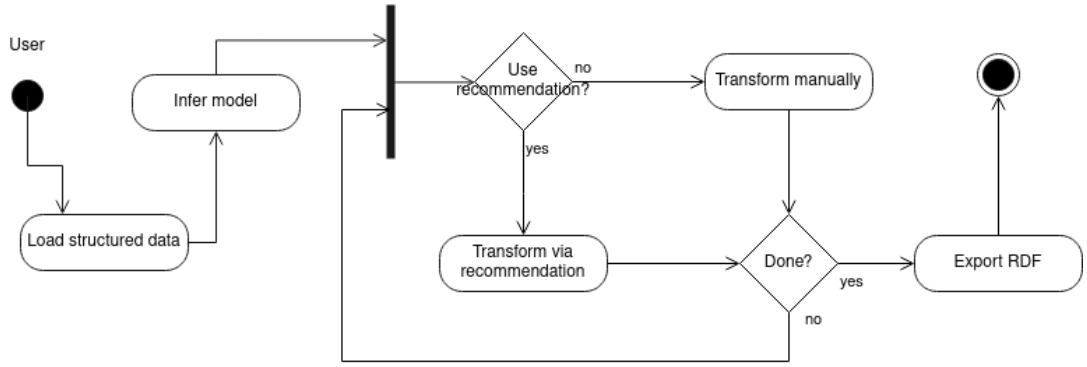

Figure 2.7 Transform Workflow

The whole process of data transformation is meant to be semi-automatic and user guided. The workflow of the proposed system is shown in Figure 2.7. The user loads structured data based on which a model is inferred that represents the RDF representation of the loaded data. The model contains a schema capturing the structure of the loaded data which is presented to the user. The system creates recommendations suggesting transformations of the model such as recommending to use vocabulary terms in a part of the model but it can even suggest structural changes along with vocabulary terms. The user then interactively updates the model until they are satisfied and export the desired representation of the loaded structured data to RDF. The user can either update the model manually or use the recommendations which are capable of automatically transforming the model. The user can also just investigate the recommendations for recommended and related terms and decide to add the terms manually. Chapter 5 contains an example transformation of the food data from the motivating example (Section 2.1) using the final implemented system. It contains screenshots and usage description; therefore, it can provide a better understanding of the proposed transformation environment.

Compared to Karma where the structure of the initial input data is not considered for the initial model, we make the assumption that the original data are represented in a conceptual structure mirroring to some extent the real world. For example, we suppose that an object in JSON represents a concept and its

JSON properties also relate to the concept. Therefore, the model contains a schema accounting of the structure of the data.

2.4.2 Challenge Two - Many Recommended Terms

Consider a user in a process of transforming structured data to RDF and one of discussed search methods (see Subsection 2.3.1) return matched vocabulary terms to represent a part of their data. All of these methods are general and can be used on any data. However, the discussed methods typically match many results which are ranked using a collection of general purpose metrics such as term popularity. Therefore, even terms that are ranked at the top might not be the most relevant out of the matched results and a result investigation as well as result browsing is often necessary. This was also the case for the example food data transformation (see Subsection 2.1.1).

While this by itself might not be an issue when performing one search with a specific keyword, recommending based on input data using possibly many keywords searches would result in even more matched results. It seems infeasible and inconvenient for the user to have to browse through many term options.

Recommending More Than Vocabulary Terms

Furthermore, the discussed vocabulary recommendation methods and tools (Karma [19], TermPicker [16], LOV [3], BioPortal [31], ...) recommend only strict vocabulary terms defined by RDFS or OWL or a similar ontology definition vocabulary. However, recommending other terms for the representation of data is also important. In the example food use case (see Section 2.1), the product had a literal reference "United States" to the country where it was sold. One possible recommendation was to replace the literal with a code list value from European Data Portal [36] (see Subsection 2.3.2). Another use case is linking to DBpedia [46] resources representing real world entities. Not only do we want to support recommending from code lists or DBpedia but also from all possible such sources.

Expert recommending

Instead of primarily using general recommendation methods generating many recommendations, we propose to recommend based on many small recommenders specific to a certain domain using expert domain knowledge to provide term recommendations. Each recommender recognizes whether it can recommend something from its domain and if not, it outputs nothing. Therefore, such recommenders only produce results if they are fairly certain that the input data are relevant to their domain. Each recommender also has access to RDF data (e.g. LOD cloud) in preprocessing to extract whatever necessary data it needs.

This way of recommending also alleviates the other issue of recommending only strict vocabulary terms. For example, there can be an expert recommender for code lists which searched the whole LOD cloud for code lists and recommends their values. The recommenders can even use the general search methods and quickly limit the results based on their expert domain knowledge.

Obviously, this approach to recommending is only useful if there are recommenders of high-quality for the domains of the imported user data. Therefore, it

is necessary to design a solution which enables adding new recommenders to the system easily.

2.4.3 Challenge Three - No System for Multiple Methods

The discussed paper recommendation methods are usually implemented for the evaluation of the said methods but not released with enough documentation or maintained. Some prominent examples which are maintained are LOV [3], Karma [37], Silk [38] or BioPortal [4]. But even then, they implement their method and do not take account of other methods whose implementation could make their service better.

Although we preferred to use the expert recommendation idea to these general methods in the challenge two section, we do not dismiss their value. Therefore, we expect the system to have support for adding such general purpose methods for both a recommendation improvement and a method comparison. This goes in hand with expert recommendations which are small self-contained recommendation methods.

2.4.4 Summary

We stated three challenges related to transforming structured data to RDF and our ideas for solving them. In this subsection we merge the presented ideas to provide a summary of our approach based on which we can design our system in the following chapter.

We start by summarizing Subsection 2.4.1. We propose to provide an interactive environment (i.e. system) which a user can use to transform structured data to RDF semi-automatically. The environment lets user import structured data based on which their model representing their RDF representation is inferred. The model contains a schema capturing the structure of the imported data as well as the underlying imported data values that the user can browse. The model can be exported to RDF. The user can manipulate the schema (i.e. the structure) and add RDF artifacts (such as RDF types, resource and property URIs) to influence how the data should be represented in exported RDF. This can either be done manually or by using transformation recommendations. These recommendations suggest how the parts of the model should be represented in RDF. They can be used for automatically transforming the model based on the suggestions or for searching for fitting vocabulary terms which the user can then use manually. The workflow is captured in Figure 2.7.

How exactly recommendations work is discussed in Subsection 2.4.2. Recommendations are generated by recommenders having some built-in expert domain knowledge they use to produce recommendations for the specific domains. The recommenders have access to the imported data as well as external RDF data serving as a knowledge base (e.g. LOD cloud). The recommenders only produce recommendations for the domain and if the imported data are outside it, they produce nothing. Also, it is important that new recommenders can be added fairly easily since they are typically small and domain focused.

Subsection 2.4.3 then specifies that we want our system to support creating recommendations based on general purpose search or recommendation methods

discussed in Section 2.2 and Subsection 2.3.1.

We identify the main functional components and design the system architecture based on the described approach in the next chapter.

3 Design

In this chapter, we design a system based on our proposed approach from Analysis (Section 2.4). First we identify the main components of the system and how they should interact with each other based on the proposed approach. Based on these components we design the architecture of the system. Lastly, we look at how to represent user's structured data.

We iteratively analyze and design the system. Both the architectural and data representation sections introduce the requirements based on which the architecture or data representation are designed. These requirements are typically influenced by some design decisions made in the previous sections or chapters; therefore, we do not employ the classical approach of defining all system requirements beforehand.

3.1 Main Components

In this section, we identify the main components of the designed system based our proposed approach Section 2.4 from Analysis. We consider a component to represent a group of semantically related functionality without specified runtime allocation.

The proposed approach proposes to create an interactive environment where a user imports their structured data and lets the user manually or by using transformation recommendations transform the data represented in an internal model into a state from which the user can export RDF representing the data using suitable vocabulary terms. There are two main components hidden. One is the application providing the environment that the user interacts with which we identify as the **Editor** component and the other is a component producing the recommendations which we consider to be the **Recommendation Provider** component.

When a user imports structured data to **Editor**, Editor transforms the data into a model representing the RDF representation of the data containing the aforementioned schema that captures the structure of the original data and information about the original data. The model also contains RDF related artifacts such as RDF types or URIs assignable to the data. The model's design is described in Section 3.3. The user then can iteratively make transformations of the model such as changing the structure of the data, changing data values, adding RDF types, assigning URIs and when they are satisfied, export to the desired RDF. These transformations can be done manually or by using recommendations that suggest how to transform the model. The user can investigate the any recommendation to see what terms it recommends using, its description and visual changes of the internal model. The user can either apply the recommendation which means its suggested transformations are done or use the gathered information from the recommendation and do the changes manually.

Recommendation Provider is responsible for creating transformation recommendations for the current version of the Editor model. Therefore, when recommendations are requested, the current Editor data (e.g. the model or a structure derived from it) must be given to Recommendation Provider. The component consists of smaller recommenders responsible for creating recommendations

that are based on expert knowledge or on general-purpose search or recommendation methods. We also mentioned that each recommender can use external RDF data as a knowledge base to create recommendations. It can precompute any kind of index from these data to later use it to create recommendations for the Editor data. To bring more clarity to what a recommender can be, we define the following types of recommenders based on what kind of input they use along with recommender examples.

Editor Data & Expert Knowledge This kind of recommender uses only the Editor data and some built-in expert domain knowledge to create recommendations. We include example below.

Suppose that in the imported structured data there is a country specific date string such as `"DD.MM.YYYY"` for Czechia which does not conform to `xsd:dateTime`. A recommender for czech dates could recognize czech date time string and provide recommendations for their transformation to a format compatible with `xsd:dateTime`.

A recommender does not need to work on the level of literal value change but can be capable of transforming a part of the Editor data, even transforming the structure. Such example can be recommending terms from a certain vocabulary as such Food Ontology [11] used in the motivating example (Section 2.1). It has predicates for nutrient information of a food product (e.g. carbohydrates). A recommender then could try to find nutrient information strings using simple or advanced text similarity in the Editor data and if any part of the data matched, it would provide recommendations for transforming that part of data to the Good Relations [12] model that Food Ontology is based on for adding quantitative properties along with any nutrition predicates from Food Ontology.

An even more complex example that works on the whole Editor data could be a recommender that is capable of detecting statistical data and provides recommendation for transforming them to a Data Cube [47] model.

Editor Data & RDF Data This type of recommender produces recommendations for any Editor data independently of the data domain using the external RDF data (e.g. LOD cloud) preprocessed to some kind of index. It can, for example, be based on a general recommendation or search method discussed in Analysis (Section 2.2). We again include example below.

An example of such a recommender is a recommender that preprocesses the external RDF data to a full-text index. Then, when it receives the Editor data, it finds the best matches between the index and the Editor data. It can then recommend to represent the matched Editor parts by vocabulary terms used by the matched data from the index.

Another example is a recommender that when recommendations are requested, it uses the LOV [3] public API to perform search for RDF terms in LOV vocabularies based on the received Editor data and subsequently creates recommendations based on the top results.

While these recommenders can change the structure of the Editor data, they typically only recommend predicate or type URIs along with description of why they are recommended.

Editor Data & RDF Data & Expert Knowledge The last type of recommender uses both an index created from external RDF Data and expert knowledge. We again include examples below.

An example of such a recommender is a recommender for code lists. There is a code list for currencies published by Publications Europa Portal [1]. If this code list is provided to the code list recommender, then it can compare the currency codes from the code list to literals in the Editor data when recommendations are requested. If there are literals matching codes, the recommender can suggest to replace the literals with the well-known URIs of the codes.

Another example is a recommender that saves triple that contain RDFS vocabulary terms from any RDF data provided to the recommender. The recommender could then compare the Editor data with the saved triples and produce recommendations assigning RDF types or property URIs to matched parts of the Editor data.

We mention that these recommenders can process external RDF data and create supporting structures for recommending. It is not clear what these RDF data are, where they are taken from, how they are processed and when recommenders have access to them.

One option is that RDF data can be collected by crawling (such as in Swoogle - Subsection 2.2.3 - or in Falcons - Subsection 2.2.4) and then provided to all recommenders (of the second and third type that require external RDF data) either before the system is live or when the system is running to update their recommendation data structures. This approach makes us search for the data to use.

An alternative to this approach is to create a catalog where RDF data can be uploaded. The catalog would then notify recommenders of new RDF data. This approach is used in LOV [3] where they curate the uploaded vocabularies. Moreover, RDF data can be sent to the catalog in a standard and efficient way using RDF dataset vocabularies such as DCAT or VOID. There are many external catalog services that provide dataset records; therefore, it is a quite elegant way to get inputs to out system.

If we compare the options in terms of data quality, there are no guarantees in the case of the crawler unless we explicitly implement data quality filtering functionality. In the catalog approach the responsibility for providing high-quality data is on the user uploading the data to the catalog. Moreover, the quality can be offloaded to the external catalogs that curate their datasets. Therefore, we can choose which catalogs we trust to use their data.

Comparing the crawling and catalog approaches in terms of handling data updates favours the catalog. Again the catalog could rely on the external catalogs tracking updates or on dataset metadata to check update dates. Therefore, it would refetch only dataset data for changed datasets. In the crawling approach, the crawler would need to retrieve all data again and compare them against the last crawled version (or just update all system data). We choose the catalog approach for the aforementioned reasons as well as it being less complex solution in

[1] `https://op.europa.eu/en/web/eu-vocabularies/dataset/-/resource?uri=http://publications.europa.eu/resource/dataset/currency`

terms of implementation. Since the Recommendation Provider would have many responsibilities, we define a component **Catalog** that is responsible for managing datasets. If a dataset is uploaded, it stores it and notifies all recommenders that there is a new dataset they can process.

Each defined recommender (and thus Recommendation Provider) recommending based on external RDF data still does two tasks - process external RDF (dataset data) to create internal structures for recommending and provide recommendation based on the Editor data. If these tasks are combined in one component, then the produced internal structure is likely to be heavily optimized for the recommendation task. However, splitting each task into a component forces the RDF processing task to create a well-defined structure that can be reused by other recommenders.

Therefore, we replace Recommendation Provider from the main components by a group of **Recommender** components and a group of **Analyzer** components. Analyzer performs the processing of datasets producing analysis (i.e. the originally called internal structures). The analysis are well-defined and not tightly connected to a recommender. Recommender does the recommending task. We prefer to call them a group of components instead of creating one component for the whole group since we then can reason based on the individual Recommenders and Analyzers.

Recommender can be dependent on analysis of one and more Analyzers. This solution is also more flexible, since new Analyzers can be added using some already existing analysis definition to seamless make its analysis usable in Recommenders using the analysis definition. And new Recommender can be added and use already defined Analyzers and analyses created by them.

3.2 Architecture

In this section we design the architecture of the system based on the identified main components. We first specify the goals based on which we evaluate any proposed architectures. Then we explore iteratively various simplified architectures and arrive at the one fitting the requirements the most. We subsequently take the chosen simplified architecture and remove simplifications to derive the final architecture based on which the system is implemented. The simplifications typically concern communication between one and more components or communication with databases whose exact description would unnecessarily complicate the idea of the simplified architectures.

We model all architectures in the C4 model [48]. The terminology is slightly different from other models. Components are static blocks of related functionality encapsulated by a well-defined interface. Containers are runtime deployable units that execute codes or store data. Containers typically consist of components.

Note that when we mention the term **dataset** in the following sections, we mean a DCAT dataset not the actual data which would be a distribution in the DCAT terminology.

3.2.1 Architecture Design Goals

We develop the simplified architectures based on the four main defined components. Editor lets users interactively transform structured data to RDF and

requests recommendations from Recommenders. Recommender takes the Editor data as an input, optionally fetches any required analyses from Analyzers and produces recommendations. Analyzers create analyses based on RDF data inserted into Catalog in the form of DCAT datasets. Note that terms Recommenders and Analyzers mean the groups of all Recommender or Analyzer components.

The goal is to find an simplified architecture that supports the following.

- Adding new Recommender or Analyzer is possible and fairly easy.
- Multiple Recommenders can use analyses of multiple Analyzers.
- While Recommenders are closely knit to Editor and its data format, analyzers analyze any RDF data to infer knowledge that may be useful in other applications. Therefore, a minor goal is to be able to reuse analyzers in different contexts.
- Very subjective implementation simplicity.

3.2.2 Catalog Monolith With RDF Triplestore Architecture

Perhaps the simplest architecture is to group the main components in a single container and have only one store for all kinds of data. The architecture in Figure 3.1 groups Catalog, Analyzers and Recommenders into a single container named Catalog whose main function is to let administrators upload datasets for analysis and provide recommendations for Editor. Any data used by Catalog is stored in a RDF triplestore.

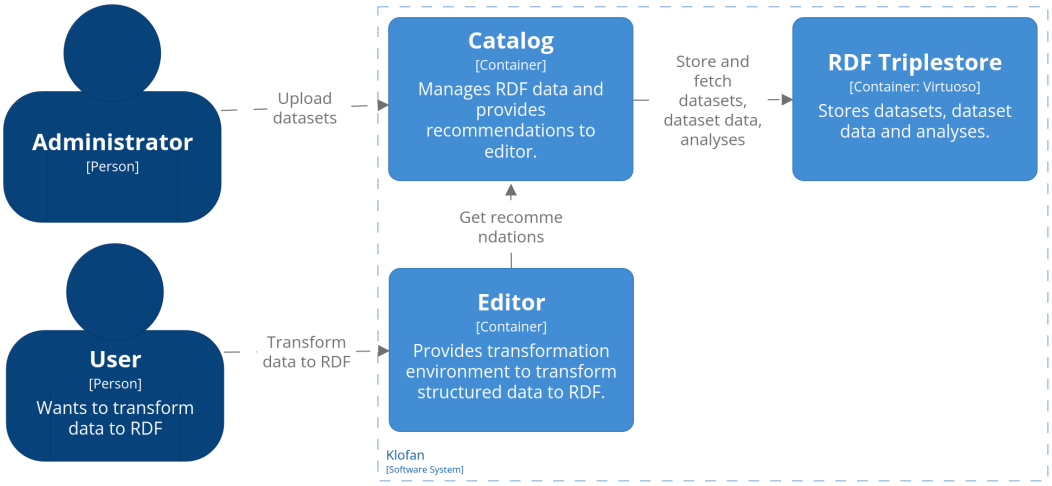

Figure 3.1 Catalog Monolith Container View

Having the three main components in one container does not mean that the respective functionality is interlinked in a black box. Instead, the Catalog container contains Dataset Uploader, Analyzers and Recommenders components as shown in Figure 3.2). Dataset Uploader receives any uploaded dataset by a user, retrieves its data and saves both in the triplestore. Afterwards, it notifies

Analyzers that new data were uploaded, where they are in the triplestore (e.g. identified by graph URIs) and that they are ready to be analyzed. Each Analyzer then accesses the uploaded data in the triplestore and performs analysis whose results (i.e. analyses) the Analyzer saves in the triplestore as well. While all these different kinds of data (datasets, their data, analyses) are saved in different RDF graphs, they contain links among them. For example, an analysis can have a link to the data created from or the dataset it is meant for.

Recommender components are responsible for providing recommendations to Editor. Each Recommender takes the Editor data as input, optionally fetches analyses from the triplestore and produces recommendations as output.

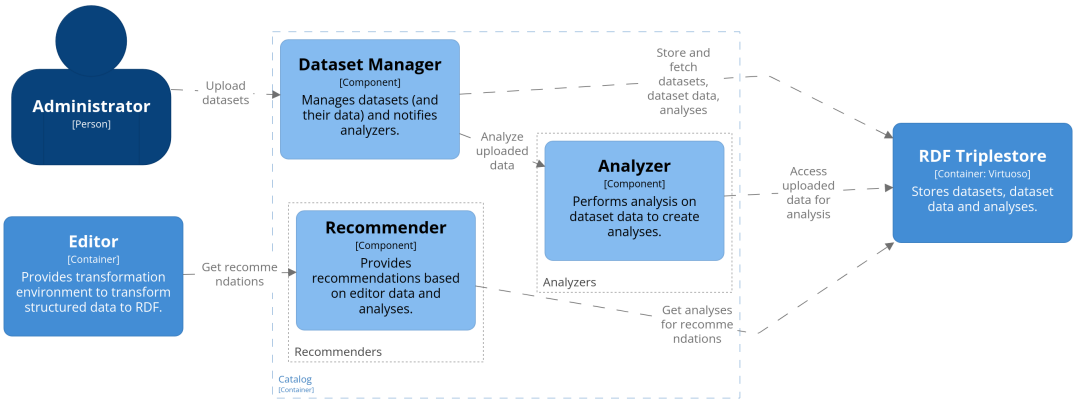

Figure 3.2 Catalog Monolith Component View

Architecture Evaluation

We first consider the single storage being a RDF triplestore and then the monolithic nature of the architecture. The RDF graph model in a triplestore is quite powerful for representing any kind of information. Each analyzer can choose an arbitrary graph structure of analyses it produces and can link and reuse original data instead of having to store them which would be necessary if another database type such as a relational database was used to store analyses. Moreover, graph databases can store the data close to conceptual reality as opposed to other types of databases which can restrict the structure of the data such as to relational tables. Since analyses are in the RDF database, a recommender can write SPARQL queries to access multiple types of analyses and convert them to its desired output even in the query.

While relying heavily on RDF is highly flexible, adding a new analyzer or recommender requires knowing SPARQL and writing non-trivial SPARQL queries. Based on personal experience, the result of SPARQL - bindings or triples - also requires some grouping and merging work to convert it into a suitable structure usable from code in comparison to retrieving simple JSON documents from a document database. Therefore, the complexity is shifted onto the programmers of analyzers and recommenders making adding new analyzers or recommenders difficult.

Another problem with the single triplestore is that any dataset data or analyses must be in RDF. In the presented workflow, we cannot consume non-RDF datasets.

Furthermore, if we want to recommend using a full text search or using the discussed TermPicker (Subsection 2.2.1), we would need to somehow fit their recommendation support structures (i.e. analyses) in RDF or add data stores in a custom way not represented in the architecture.

For these reasons we might want to split the catalog storage and analysis storage which we consider in the next presented architecture.

Now we consider the catalog monolith and how it fits out predetermined goals. There the main advantages lies in its (runtime) simplicity and deployment. The main disadvantage of the approach is that a new analyzer or recommender must be statically included. Therefore, it must be implemented in the same language as the rest of the code or use language bindings which increases the difficulty of adding new analyzers or recommenders. Moreover, reusing analyzers and their analyses is not readily possible apart from including and referencing the Analyzer components in code.

Hence, we shift to more runtime architecture approaches.

3.2.3 Recommender Container Architecture

This architecture is based on the proposed architecture changes for the monolithic architecture. The previous Catalog monolith container is split into new containers - Catalog container and a group of Recommender containers. Since we found having a single RDF triplestore non-optimal, we consider a triplestore only for datasets and separate stores for analyses. This architecture is shown in Figure 3.3. Catalog provides API for uploading datasets which are forwarded to each Recommender container and saved in Dataset Triplestore. A Recommender container fetches dataset data and analyses them. Created analyses are saved in Recommendation Analyses Store. Not to lose track of which analysis was created for which dataset, provenance about each analysis such as who created it and where it is located is sent back to Catalog.

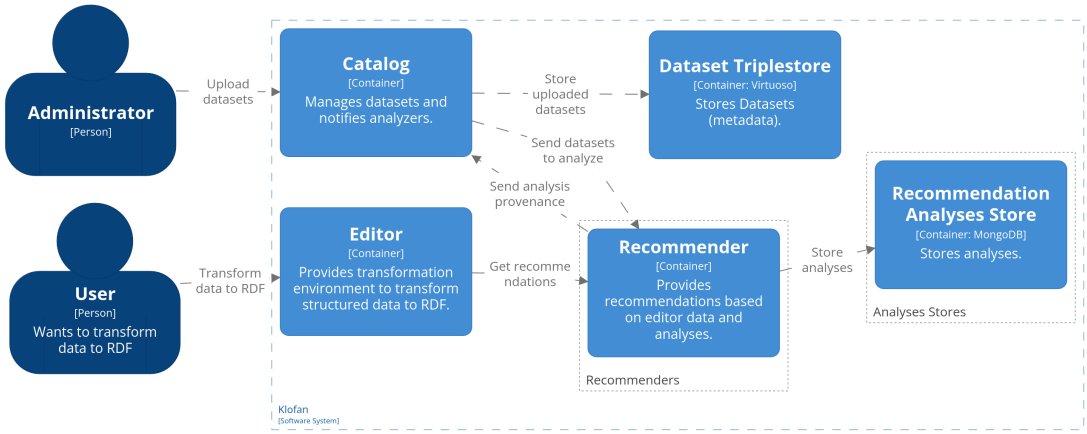

Figure 3.3 Recommender Architecture Container View

Similarly to before, Editor requests recommendations from Recommender containers which use Recommendation Analyses Store containers to get analyses. There is a group of containers for analyses stores since each Recommender might have different requirements for storing analyses. While most could be satisfied

with a key value store or a document store, there may be a need to store data, for example, in an information retrieval system.

Since a Recommender container has both analyze and recommend responsibilities, it consists of Analyzer and Recommender component which handle their corresponding responsibilities (see Figure 3.4).

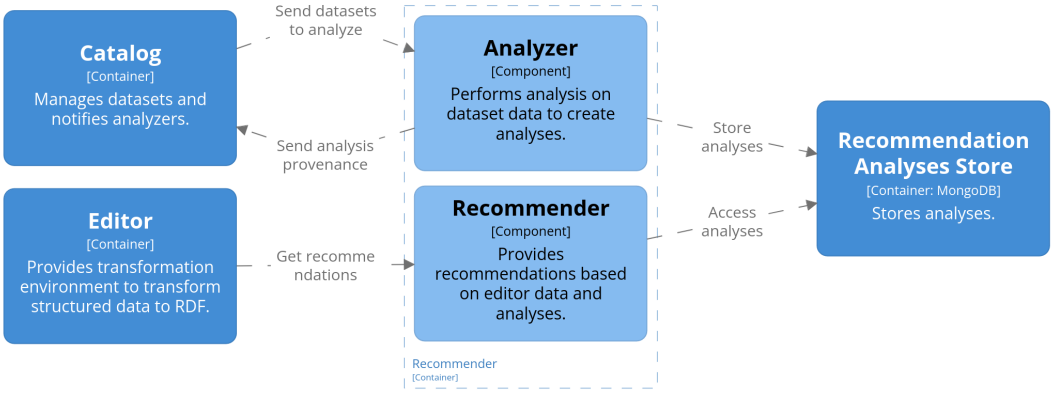

Figure 3.4 Recommender Component View

Architecture Evaluation

We evaluate the presented architecture. Let us reiterate the main issues identified in Catalog Monolith Architecture. We could not add an analyzer or a recommender in a language independent way and all analyses were required to be stored in RDF. That also implied that each recommender retrieving analyses would have to retrieve the analyses from the triplestore typically using SPARQL and do some processing of the query results to get a structure suitable for doing the actual recommending. Despite the approach being flexible and powerful, we opt to make implementing new analyzers and recommenders simple.

These issues are now resolved. A new analyzer or recommender can be added implemented in any language added as a runtime container. Analyses also do not have to be stored in RDF and an analyzer creator can choose to store analyses in existing stores or add a new analysis store.

Another benefit is that the cataloging functionality is in a runtime container designed for cataloging datasets which makes it simpler to understand and implement. Moreover, the cataloging part can be replaced by connecting an existing external catalog application to the Recommender containers. Or there does not even have to be a catalog to begin with and the administrator can directly send datasets to Recommender containers to get analyses for them. However, the entire Recommender containers would have to be taken for such use cases which might be inelegant if the main focus is to get analyses to be used in different contexts.

Even if no such reuses were considered, coupling both analyzer and recommender functionality in one container is a bit incomprehensible when adding either a recommender or an analyzer. For example, there can be an already mentioned recommender for recommending converting dates in czech format to a `xsd:dateTime` compatible format. This recommender does not require any

analysis or analyzer implementation. Should the resulting container have an analyzer producing always no analyses?

What about implementing an analyzer for multiple recommenders? Should there be a Recommender container with an empty recommender that does nothing? Should the analyzer be coupled to one recommender that uses the analyzer's analyses? In that case the other recommenders using the same analyses would be tied indirectly to the recommender as well.

While at least partial answers to these questions can be found on a technical level by, for example, providing guidelines how such things should be done, they cannot be reflected in the proposed architecture. We want to solve this issue on an architectural level; therefore, we discuss one last architecture where no such problem occurs.

3.2.4 Analyzer Container Architecture

In this architecture we build on top of Recommender Container Architecture and resolve its discussed problem of coupling analyzer and recommender functionality together in one runtime container. Therefore, we have a group of Analyzer containers which perform analyses and save them in Analyses Stores as well as a group of Recommender containers which can retrieve said analysis from the stores and provide recommendations for Editor. This architecture is shown in Figure 3.5. The rest is the same as before in Recommender Container Architecture; hence, we skip further description.

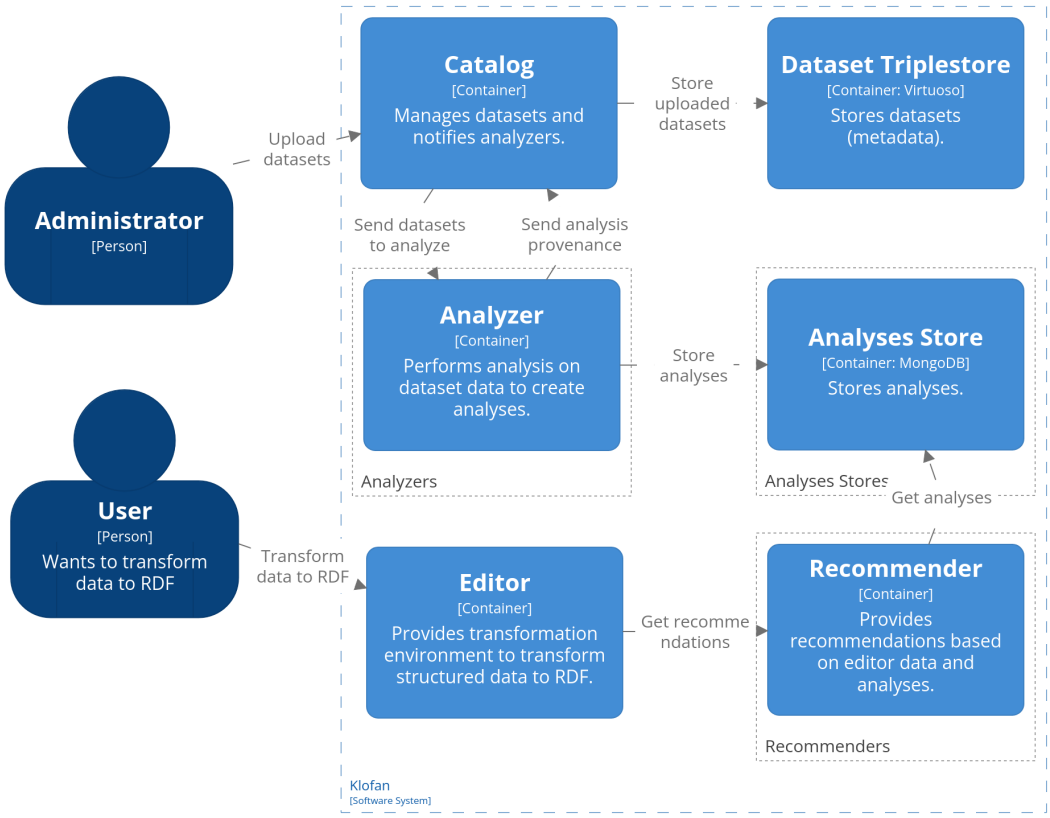

Figure 3.5 Analyzer Architecture Container View

We now evaluate the architecture in terms of our given goals to make sure there are no major issues. New analyzers and recommenders can be added by adding a runtime container implementing analyzer or recommender well-defined API using a language of preference. It is clear that adding only an analyzer or recommender is supported. Recommenders are not coupled to analyzers but rather to their specified types of analyses; therefore, they can use the analyses of multiple analyzers of their choice. Moreover, since analyzers are completely split off of Editor and are standalone containers, they can be reused in different contexts. This flexibility adds, however, some runtime complexity in terms of communication and deployment.

To sum up, this architecture supports the main defined goals with the trade-off of having increased runtime complexity in comparison to the architectures discussed before.

3.2.5 Final Architecture

In this section, we derive the final architecture of the system. All of the previous architectures were focused on how to represent the main components and on the high-level communication among them. While drafting architectures on such a level is useful for presenting the main ideas, the drafts are too high-level to present the complete picture from which the system can be implemented. Therefore, we delve deeper and arrive at the final architecture based on which we implemented our system.

We first identify the vague places in Analyzer Container Architecture, where more detail is necessary to describe how it should work without relying on ambiguous simplifications, and propose our solutions. Then we showcase the whole architecture.

Communication between Catalog and Analyzers

In Analyzer Container Architecture, Catalog sends datasets to Analyzers in order to get dataset analyses. It is not clear how exactly the communication happens. Based on the brief explanation it seems the Catalog must know of all Analyzers and send a request to each. However, knowing the addresses of Analyzers and handling communication with all of them should not be Catalog's responsibility. Therefore, we introduce a container named Analyzer Manager with such responsibility. It provides an endpoint that accepts datasets and on accepting its requests, the manager sends the datasets to all Analyzers. Catalog then only needs to send datasets to Analyzer Manager.

Each analyzer can run analysis of a dataset for a significant amount of time. Moreover, requesting analyses of many datasets at once (such as uploading a large DCAT catalog) can throttle the performance and memory consumption. Therefore, there is a queue for each Analyzer from which the Analyzer retrieves datasets to analyze. An item in such a queue is called an analysis job and it contains one dataset among other things discussed later.

One question still remains. How does Analyzer Manager keep track of available Analyzers? There are dynamic and static solutions. An example of a dynamic solution is to have Analyzers register to Analyzer Manager at runtime. An example of a static solution is to have analyzer queue locations be a part of the

configuration of Analyzer Manager. We choose the static configuration approach for the sake of simplicity.

Sending Analysis Provenance

In Analyzer Container Architecture, we mentioned that each Analyzer container sends analysis provenance to Catalog. The provenance includes information about the analyzed dataset, the analysis creator or the URI of the analysis for retrieving it. It can be sent to Catalog when Analyzer has analyzed a dataset and stored it. Since each analyzer can run for long time and we employ queues to submit datasets to analyze, the provenance information cannot be sent as a response to Catalog sending datasets to analyze.

Hence, analyzer sends provenance to Catalog in a separate request after creating analysis. We could include the Catalog address in configuration and hardcode sending provenance. However, there may be another pieces of information that analyzer can send to not only Catalog. For example, if Catalog does not receive any analysis provenance for a dataset, it can be due to various reasons. Either the dataset contains no suitable data for analysis or the analyzer crashed and subsequently a potential analysis was lost. Or we could send information about when analysis of a dataset was started.

Therefore, we consider the notion of sending general notifications. Catalog can pass a list of notification requirements along with datasets to Analyzer Manager. An example of such a requirement could be that Catalog wants to receive provenance in PROV Ontology [49] when analysis is done. Each such requirement must contain the address of the target and when the notification should be sent. These requirements are passed to the analyzer queue along with datasets. Then, when an analyzer is done, it reads the requirements and sends `analysis done` notifications.

Communication between Editor and Recommenders

Editor requests recommendations from multiple Recommenders; therefore, it needs to know their locations. This is a similar problem as sending datasets to multiple Analyzers and we employ a similar solution. We create a container Recommender Manager whose responsibility is to keep track of Recommenders and provide an endpoint for requesting recommendations from all Recommenders. Recommender addresses are a part of the manager configuration similarly to how analyzer queues are part of the configuration of Analyzer Manager. No queues are used in this case since recommendations are demanded from user and should be provided as soon as possible. To summarize, Editor needs to only communicate with Recommender Manager to get recommendations from all Recommenders.

Storing and Retrieving Analyses

In Analyzer Container Architecture, each Analyzer stored its analyses in a analysis database of its choice and each Recommender requiring analyses retrieved them from the databases. Instead of requiring each Analyzer and Recommender to access databases directly using special clients and know where each analysis should be stored, we add a container Analyses Store that is responsible for storing

analyses and retrieving them as well as providing standard HTTP endpoints for these operations. In background, it can manage multiple databases no one else has access to. At the cost of more runtime complexity this solution ensures that adding analyzers and recommenders means focusing largely on their main responsibility and not having to resolve implementation details related to storing analyses.

The only exception is when the essence of an analyzer or a recommender is dependent on a external system such as recommending based on full-text search using an information retrieval system such as Elasticsearch [27]. However, analyses from such an analyzer are still always sent to Analyses Store describing, for example, the full-text index name that a recommender should use.

Architecture Presentation

Having discussed the vague parts of Analyzer Container Architecture, we present the final architecture. We take the introduced containers from this subsection and include them in Analyzer Container Architecture to provide a clear summary of the architecture. No new containers or concepts are added.

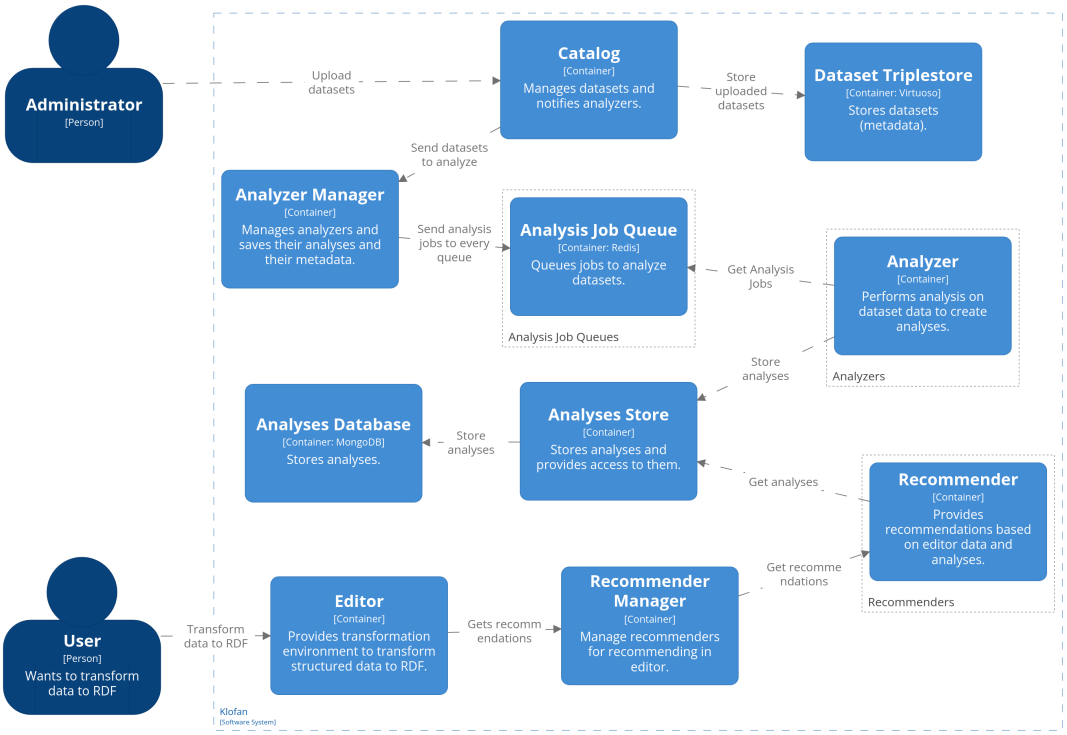

Figure 3.6 Final Architecture of the System

The architecture is shown in Figure 3.6. There are two main workflows - uploading datasets to Catalog in order to analyze them and requesting recommendations from Editor. A system administrator uploads DCAT datasets to Catalog which stores them and sends them to Analyzer Manager. Notification requirements for sending analysis provenance to Catalog are sent along with the datasets. Analyzer Manager then parses and validates the request and all datasets are subsequently added to all analyzer queues named Analysis Job Queues in the

figure. Afterwards, Analyzer Manager immediately sends response to Catalog that datasets were submitted for analysis.

Each Analyzer then retrieves and analyzes datasets from its corresponding queue. It saves any resulting analyses by sending them to Analyses Store which stores them in a persistent storage. Once the analyses are stored, the Analyzer sends their provenance based on received notification requirements.

The other workflow starts with User requesting recommendations via Editor. Editor sends a request to Recommender Manager which in turn requests recommendations from all Recommenders in its configuration. Recommenders can get analyses from Analyses Store and return recommendations in response to Recommendation Manager which groups all recommendations and subsequently returns them to Editor.

3.3 Structured Data Representation

In this section, we discuss how to represent input structured data by a model which would represent their RDF representation. We first define what we consider the structured data to be. Then we list requirements based on which we design the model. Then we follow with the design of the model. Then we split the model functionality into static components and present the component view.

3.3.1 Structured Data Definition

We already mentioned that we help a user transform structured data to RDF. However, we did not define what exactly we consider the structured data to be. Knowing exactly what we aim to support is crucial when designing its representation that is used throughout the system. Therefore, we explicitly state what the structured data mean in the context of our application.

Simply said, we consider anything that can be converted to one or more JSON files structured data that can be loaded to Editor and transformed to RDF. That means hierarchical data consisting of arrays, objects with properties and literals. The common formats such as XML, CSV or JSON are the target. Graph data would have to be split into blocks of hierarchical data and loaded by itself and subsequently joined in Editor by for example using a join mapping.

3.3.2 Representation Requirements

We present the requirements that we have for the representation of the input data (i.e. model). Based on the architecture (Subsection 3.2.5) the data are used in Editor and Recommender containers. Editor is meant to be a JavaScript frontend application.

- The model captures the structure of the input data in a schema so that the user can view the structure and make structural changes on all underlying data.
- The model supports adding and manipulating resource URIs, RDF types, language tags or data types to literals.

- The model supports structural changes and value changes.
- The model can be stored as a state in Editor.
- The model must be serializable and deserializable or convertable to such a data structure since it is sent between Editor and Recommenders when recommendation are requested.
- The user can interactively change the model; therefore, support an efficient integration with history (undo/redo) functionality.
- Transformation of large data to RDF must be supported. For example, large data that would be unwieldy to work with due to them overflowing memory and slowing performance.

3.3.3 Representation Model Design

In this subsection, we design the model based on the specified requirements.

Since the input data can be quite large and might no fit in memory or at least significantly slow performance, we split the model into two parts - schema and instances (of the schema). Schema represents the structure of the input data without any raw data values while instances represent the raw data. Therefore, schema is small and can always be kept in memory. It can be always presented to the Editor user in full to let them browse and change data structure. Instances (of schema) represent the potentially large data structure with data values that does not necessarily need to be kept in memory and is materialized on demand.

We continue with the design of schema and of instances separately.

Schema

Schema captures the schema of the structure of the input data. We mentioned that the input data consists of arrays, objects with properties and literals. If there are no arrays, we deem that the schema representing the structure of the data should structurally mirror the data. If there are arrays, consider the example below. There is a root object with property `products` targeting an array of objects. These objects then represent food products and they can have different properties such as the latter having property `vegan`. Note that the there are also different properties in objects targeted by properties `nutriments`. If schemas were created for the product objects in the array, they would be different. The former would contain `carbohyrates_100g` in `nutriments` while the latter `energykcal_100g` in `nutriments`.

```
{
  "products": [{
    "product_name": "Thai peanut noodle",
    "countries": "United States",
    "nutriments": { "carbohydrates_100g": 71.15 }
    }, {
    "product_name": "Pasta",
    "countries": [ "United States", "Canada" ],
    "vegan": true,
    "nutriments": { "energykcal_100g": 300.0 }
```

```
  }]
}
```

We define the schema of the array to be a schema created by merging the schemas of all array elements by including all of their objects and properties. The schema of the example data could have the following structure. `<Literal>` term represents that there are only literals in the corresponding position in the underlying data.

```
{
  "products": {
    "product_name": <LITERAL>,
    "countries": <LITERAL>,
    "vegan": <LITERAL>,
    "nutriments": {
      "carbohydrates_100g": <LITERAL>,
      "energykcal_100g": <LITERAL>
    }
  }
}
```

The schema consists of three different elements - objects, properties and literals. We define Entity Set to be an object in such a schema. It conceptually represents a set of objects in the input data at the same position as it is in the schema. Such objects are then named Entities. We define Literal Set to be represent a place of `<Literal>` in the schema which represents the set of literals at the corresponding position in the data. Lastly, we define Property Set corresponding to a property in the schema which represents the set of properties at the corresponding position in the data.

This presented schema structure can be encoded in multiple ways. We discuss these ways with respect to the requirements (Subsection 3.3.2). A straightforward solution is to represent exactly as it is shown above by having a tree of objects representing entity sets where its properties are property sets. This representation, however, does not allow storing anything else in the objects outside the structure of the data.

If the model is to be converted to RDF, it needs to support adding URIs to data objects and properties, assigning RDF types to objects, adding language tags or types to literals. While the schema is not suitable for storing individual entity URIs, language tags or data types, it is the place to set something for all underlying entities of entity sets or properties of property sets. All such entities might share the same RDF types and all such properties might share URIs. Therefore, the straightforward representation is not suitable.

To solve this issue, we create explicit interfaces for the schema elements as shown below. All of the elements contain id to be referencable. As we discussed, it is possible to set URIs for all properties of a property set and types for all entities of an entity set.

```
interface EntitySet {
  id: string;
  types: string[];
  properties: PropertySet[];
}
interface LiteralSet { id: string; }
interface PropertySet {
```

```
  id: string;
  uri?: string;
  value: EntitySet | LiteralSet;
}
```

This representation of a data schema suffers, however, when something, for example, a property set URI of property set `A` is changed. If we want to keep versions of the schema for undo and redo operations as we mention in the requirements, then we have to store a copy of the schema and then update the URI in `A` in case all of the objects are mutable. If they are immutable, then a new property set `A'` with the updated URI must be created from `A`. This new property set must be added to `properties` fields of entity sets that contained `A` and `A` must be deleted from them. Therefore, new entity sets with updated properties must be created. However, the old entity set objects were contained in `value` of property sets which must be subsequently changed. It is also necessary to handle a graph schema which could recursively update schema indefinitely in a circle. Therefore, a simple change requires copying of possibly the entire schema.

Hence, we need to introduce less coupling between entity sets and property sets. Instead of entity sets and property sets containing objects, they can contain only ids of other entity sets or property sets. If a property set or entity set is updated then, its id stays the same. It is only necessary to create the new updated object and other objects automatically point to the new objects. However, the old and the new object have the same id. Therefore, we need to remember which objects are in the current schema which are old. We solve this by storing all property sets, entity sets and literals set in one object which defines a version of the schema. The new schema representation can be seen below. Note that to be more general about the content of the schema, we say it consist of items and relations between items. Items then are entity and literal sets. Relations are property sets.

```
interface EntitySet {
  id: identifier;
  types: string[];
  properties: PropertySet[];
}
interface LiteralSet { id: string; }
interface PropertySet {
  id: identifier;
  uri?: string;
  value: EntitySet | LiteralSet;
}
interface RawSchema {
    items: { [key: identifier]: EntitySet | LiteralSet };
    relations: { [key: identifier]: PropertySet };
}
```

If something changes in the presented schema, the items and relations of `RawSchema` only need to be shallow copied (e.g. using spread operator) to a new schema object and updated or new items or relations just have to be set in the `items` or `relations` object maps which overwrites their old versions by default. Therefore, the integration with undo and redo operations is fairly easy.

We did not consider more object oriented representation for two reasons. First, the schema must be somehow stored as a state and storing classes with functions in

states is not recommended. Another reason is that the schema is shared between Editor and Recommenders; therefore, using simple objects makes serialization and deserialization straightforward.

The schema is not large even for large data since it only captures the structure; therefore, no special handling is done.

We encapsulate the internal structures of the schema (i.e. `RawSchema`) and provide an interface for accessing the schema. Schema can be transformed only by using transformation objects. Its endpoint accepts an array of transformation objects and returns a transformed version of the schema. This is useful for implementing undo/redo operations on top of the schema since many small update operations can be merged to one operation. Another advantage is that Recommenders can send recommendations containing only defined schema transformations instead of an internal schema state. Note that the interface also needs to provide the raw internal state of data so that Editor can store it as a state.

Instances

Instances represent the original data. Since the structure of the data is represented by the schema, it is not necessary to handle it here. Instances need to remember for each entity its properties and its values. It should be also possible to add language tags or data types to literals and URIs to entities. The representation could be done using the same structure as was used in schema - having an entity object with properties linking to either literal values or other entities.

However, performing a the same conceptual transformation in the instances can be more difficult than in the schema. For example, changing a property set source to a different entity set requires updating all old and new source entities while the update in schema is done only one new and old entity set. Basically, the transformation done for schema would be repeated here for every entity. Since implementing transformations in this representation is quite complex, we choose instead to rely on schema representing structures and implement more column database approach and perform operations not on single entities or properties but on arrays of entities or properties.

For each property set with its source entity set, we store an array of properties. Each property contains an array of literals and an array of indices of entities of the target entity set. Moving a property set transformation is then easy to implement in the representation. It is only required to move the literal and entity arrays to be under the new source entity set. As for entities, we need to be able to assign URIs to them. The representation in code is shown below. The keys of the objects are IDs from schema which is entity set id for representing entities and a combination of source entity set id and property set for properties.

```
interface Literal {
  value: string;
  type: string;
  language?: string;
}
interface Property {
  targetEntities: number[];
  literals: Literal[];
}
interface RawInstances {
```

```
  entities: { [key: identifier]: {uri?: string}[]; };
  properties: { [key: identifier]: Property[] };
}
```

Each change in instances then again requires to shallow copy entities and properties object maps in `RawInstances` to create a new version of the instances and set new entity arrays or property arrays in the object maps.

Now we discuss the rest of the requirements. Using the same arguments as in schema this representation can be used to store the state as well as it is easily serializable. Similar argument can be made with the history since copying the instances requires only shallow copying the root objects in `RawInstances` and then immutable changes can be done locally by setting array of properties or array of entities.

The remaining requirement to fulfill is supporting transformations of large data. We actually design two different solutions. The transformations work the same way as for schema. There are transformation objects passed to instances based on which transformation is done on a shallow copied instances object. The simple solution is to have the instances API asynchronous. The actual data can then be stored somewhere on a server and only a proxy can be part of the editor.

The other solution is a clever definition of transformation operations. The idea is that the user can upload only example data with the same structure and therefore the same schema as the full data. They could then perform the entire transformation on the example data and export all transformations. Then the exported transformations could be run on the original large large data. Therefore, it is necessary to define transformations in a way not dependent on any set number of entities or properties or on any references to another entity by its index in the list of the entities of its entity set. For example, if a new property set is created between two entity sets, the property mapping between underlying source and target entities must be defined. A simple definition is to list the pairs of entities to be linked by a property. However, the pairs would contain only properties between the entities in the example which would not work when run on large data. Instead, a mapping must be specified based on some rules so that the transformation can be run on any data with the same schema. For example, a join, preservation, one-to-one mapping options.

3.3.4 Data Model Component View

In the last subsection, we discussed that the model representing the structured data is split into a schema and instances. We discussed mainly the low level representation. In this subsection we discuss how the model and related features are split into components. We summarize communication among the components and their main interfaces.

Both schema and instances are encapsulated into their own components that provide API for querying, transforming, exporting and loading. The API for schema is synchronous since we expect the schema to be small and sent whole if needed elsewhere while the API for instances is asynchronous. Export means producing the desired RDF representation.

The loading functionality means creating a schema and instances. Creating schema and instances separately from the structured data would require a syn-

chronization between both the loading algorithms so that they work the same way for shared areas such as an id assignment. Therefore, there is a component Parse whose function is to parse the input structured data into a tree data structure that contains both the structure (i.e. schema) of the data and the actual data values. Schema and Instances component then supports loading (i.e. creating) schema and instances from this tree.

There is also a component Transform providing transformations for both schema and instances. While both Schema and Instances components support transformations as we discussed it before, if one conceptual transformation is done across both schema and instances, separate different transformations objects must be created for both schema and instances and the user (e.g. UI code in this case or a recommender) is responsible for that the correct transformation is done in both. Since transformation creation is an important part of many places in Editor and Recommenders, we do not want to burden these places with synchronization of the states of schema and instances. Moreover, the code for common transformation such as moving a property set in both schema and instances is always the same, so there would be a code duplication.

Therefore, we the component Transform that provides factory methods creating schema and instance transformations together for various conceptual transformations such as updating literals or moving properties. Note that one such one conceptual transformation can consist of multiple schema transformations and multiple instance transformations. The correct synchronization of schema and instances is then done by this component and consumers need only to call the factory methods to create desired transformations objects.

These components are shown in Figure 3.7. Unsuprisingly, Transform is needs to access schema and instances to create their transformation and both Schema and Instances are loaded from a tree structure that is created by Parse. Instances does not include the structure of the structured data and is dependent on the ids from of the corresponding schema that it stores its data under. Therefore, it needs access to Schema.

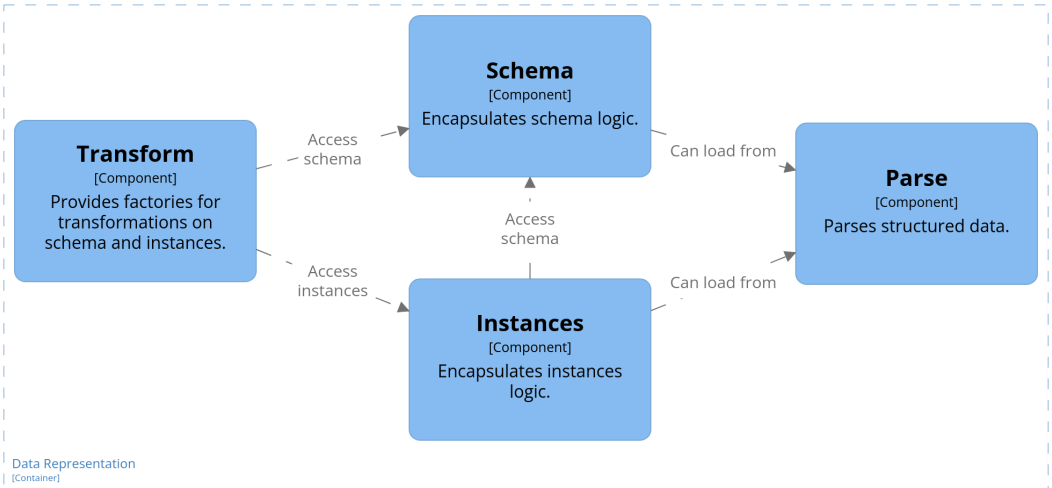

Figure 3.7 Data Model Component View

4 Implementation

This chapter describes the implementation of system based on the performed analysis Chapter 2) and design (Chapter 3) decisions. In the introduction of the chapter, we discuss the implementation overview and the system project structure. Which sections the chapter contains is presented at the end of the overview.

The entire system is written in TypeScript (TS) and managed as a monorepo using Turborepo [50] on Github[1]. Below we show the main parts of the repository. Each container mentioned in architecture is a Node.js project under in apps directory apart from Editor which is a frontend application. Each piece of reusable code used from multiple apps such as mentioned components for data representation are a TS project under packages directory.

```
repository
├── apps ..... Runtime Containers
│   ├── analysis-store
│   ├── analyzer-manager
│   ├── analyzers
│   ├── catalog
│   ├── editor
│   ├── recommender-manager
│   └── recommenders
├── packages ..... Reusable Static Components
│   ├── analyzer
│   ├── instances
│   ├── parse
│   ├── recommender
│   ├── schema
│   └── transform
├── databases ..... Database Containers
├── package.json
├── turbo.json ..... Monorepo Task Definitions
└── compose.yaml ..... Deploy Script
```

This chapter contains the following sections. First, we discuss the technology stack that is shared among many system projects in Section 4.1. Then we describe the projects. We discuss the packages which are mainly code libraries used throughout the system in Section 4.2. Afterwards, in Section 4.3 we discuss the backend applications (i.e. servers). Then we discuss Editor in Section 4.4. Implemented analyzers are listed in Section 4.5 followed by Section 4.6 listing all implemented recommenders. Lastly, we include user documentation in Section 4.7 and deployment instructions in Section 4.8.

4.1 Technology Stack

In the system there are many technologies, tools, framework and libraries that we use. Individual projects (i.e. packages, apps) use the same libraries; therefore,

[1] `https://github.com/georgeus19/klofan`

we list the most used libraries in this section.

TypeScript The whole system is implemented in TypeScript since much functionality such as the model for representing structured data in RDF is used both on backend and frontend. Moreover, we also needed to create many lightweight servers.

Turborepo Based on the architecture design, it was clear that the system would contain many servers that need to share code; therefore, we decided to structure the project as a monorepo and use Turborepo [50] monorepo manager.

Express The architecture contains many runtime lightweight server containers; therefore, we chose to use Express [51] framework for building the servers. Express is a minimalist and unopinionated framework web framework for Node.js for which there are many available plugins to customize its behaviour.

Zod With many runtime containers communicating with each other using HTTP there is a need for validation of the data in requests. Zod [52] is a TypeScript framework for validation of data that does static type inference; therefore, it can convert the validated object with typescript typing and no casting is necessary.

Lodash Lodash [53] is a library that provides rich operations on top of arrays, collection and objects. It is used typically for processing the results of SPARQL SELECT queries since the results typically contain values for the searched data in multiple result rows (bindings) which need to be grouped, filtered and made unique.

Formidable Formidable [54] is a Node.js module for parsing form data which is useful in endpoints that accept files in requests to get the file data.

Axios Axios [55] is promise based HTTP Client for Node.js and browser which is used whenever any backend code needs to send a HTTP request.

Winston Winston [56] is a universal logging library that supports multiple transports. A transport is where logs are stored. It can be for example a console or a database.

N3 N3 [57] is a JavaScript library for parsing and writing RDF data to and from files as well as storing RDF data in memory efficiently. It implements the RDF.js [58] specification for handling RDF in JavaScript.

Communica Communica [59] is an RDF graph querying framework. It offers querying various sources such as a in-memory store or a file using SPARQL.

Jest Jest [60] is a JavaScript testing framework supporting TypeScript. Any unit tests are implemented using this framework.

4.2 Packages

Packages are projects meant to be reused in other parts of the system. There are configuration projects, for example, for setting the TS configuration or the ESlint configuration. We discuss the most important TS library projects that are reused by many applications in the system below.

4.2.1 Analyzer

Analyzer library is a library for helping implementing new analyzers. An analyzer consumes its analysis job queue to receive analysis jobs. Each analysis job includes a DCAT dataset to analyze and a list of notification requirements. The analyzer inspects the dataset for suitable a DCAT distribution and gets the data. It then analyzes the data and produces analysis which it stores to Analyzer Store. It also needs to send any notifications from the received list of notification requirements such as analysis provenance.

This library aims to make the implementation of new TypeScript analyzers easier. The library provides a function `runAnalyzerServer` that starts an analyzer server that automatically consumes analysis job queue for new jobs and on analysis completion it stores any found analyses in Analyzer Store automatically as well as sends analysis provenance. It mainly requires a function that takes a dataset as an input and returns analyses.

The library also defines the dataset data interfaces in which the analyzer gets datasets in. It also provides function for retrieving dataset data in a desired distribution.

Lastly, the library provides interfaces for analyses that analyzers produce. There is an interface for all types of analyses that analyzers produce. If an analyzer needs a new type, it should add a new analysis into the library. Each analysis typically contains an id, a type, provenance information and its internal data. The only provenance representation currently supported is in PROV Ontology [49]. The provenance of an analysis contains link to the dataset created from and to the creator (i.e. the analyzer). The provenance also represents the analysis using URI that is dereferenceable to get the whole analysis object.

Note that analyzers using the aforementioned function for consuming datasets and storing resulting analyses typically also do not have to fill in analysis id or provenance fields in returned analyses. They are filled in automatically if left without values.

4.2.2 Instances

Instances library implements the Instances component defined in Subsection 3.3.4. When user uploads structured data, a model is inferred from them that the user can update to transform the data iteratively to RDF. One part of the model is instances that represent the data values using entities, properties and literals. An entity represents an object and contains properties with values of entities of literals. The representation of model in general and instances is described in Subsection 3.3.3).

This library contains all instances related logic. That includes the internal representation. The interfaces of the library is asynchronous. There is a main

interface that represents the instances of a model which the following API.

```
interface Instances {
  // To enable storing instances in state.
  raw(): unknown;
  // Query instances
  entities:(entitySet): Promise<Entity[]>
  properties(entitySet, propertySet): Promise<Property[]>
  // Transform instances by producing new instances
  // with applied transformations.
  transform(transformations: Transformation[]): Promise<Instances>;
}
```

It provides a raw version of data so that they can be stored as a state on frontend. Querying API provides a way to get entities of entity sets and properties of property sets (see Subsection 4.2.5 for information about entity sets and property sets). The transformation API is based on transformation objects that are provided to the `transform` method which returns a new version of instances with applied changes defined by the passed transformation objects. The current version of the instances in unchanged.

There is also a function for exporting the instances to RDF which produces the desired RDF and a function for loading instances from a tree structure produced by the Parse (Subsection 4.2.3) library that is capable of parsing structured data.

4.2.3 Parse

Parse library implements the Parse component (Subsection 3.3.4). The library provides a function that parses structured data into a hierarchical tree that contains both the structure (i.e. schema) of the data where nodes contain the values that occur in the corresponding place in data. What is the structure of the data is described in Subsection 3.3.3. Both Instances (Subsection 4.2.2) and Schema (Subsection 4.2.5) libraries contain load functions that take this tree and produce instances and schema data structures.

Supported input formats are currently JSON and CSV.

4.2.4 Recommender

Recommender library is a library for helping creating new recommenders. A recommender exposes a HTTP endpoint that accepts the Editor schema and instances and returns recommendations. It may fetch analyses from Analysis Store (Subsection 4.3.1) if it requires them. The library defines a recommendation interface for the recommendation data that is sent from recommenders to Editor (Section 4.4). Its simplified definition containing the main fields is below.

```
interface Recommendation {
  transformations: Transformation[];
  description: string;
  category: string;
  recommendedTerms?: string[];
  related?: { name: string; link: string }[];
}
```

It contains a list of transformation that can be applied on the editor model (i.e. schema and instances) to transform it to a new model using the recommended

transformations. Each recommendation should contain a description describing the main idea of the suggested changes, category to give a broader idea what the recommendation is about and recommended and related terms for the user to investigate.

Another important functionality the library provides is a function to create and run a whole recommender server. The simplified signature of the function is below.

```
function runRecommenderServer: (
  recommend: (editorData: { schema: Schema; instances: Instances })
    => Promise<Recommendation[]>
)
}
```

It takes a function as an argument that accepts the editor model and returns recommendations. The function `runRecommenderServer` creates an Express HTTP server with one endpoint that provides recommendation based on the editor data.

4.2.5 Schema

Schema library implements the Schema component defined in Subsection 3.3.4. When user uploads structured data, a model is inferred from them that the user can update to transform the data iteratively to RDF. One part of the model is schema that captures the structure of the data. Schema consists of items - entity sets and literal sets - and relations - property sets. An Entity set represents an array of entities/objects at a given position in the data. A literal set represents an array of literals at a given position in the data. A property sets represents a list of properties at a given position in the data. A more detailed explanation can be found in Subsection 3.3.3.

This library contains all logic related to only schema. That includes its internal representation. There is one main interface that represents the schema of a model with the following API.

```
interface Schema {
  // Get raw data for state purposes.
  raw:() => RawSchema
  // Query schema
  // entitySets(), entitySet(ID), propertySet(ID)
  // Transform schema - get new schema
  transform: (transformations: Transformation[]) => Schema;
}
```

It provides a raw version of the data so that they can be stored as a state on frontend. The querying API provides a way to gen entity sets, property sets and literal sets. The transformation API is based on transformation objects provided to `transform` method which returns a new version of schema with applied changes defined by the passed transformation objects. The current version of schema in unchanged.

The library also contains a function for loading (i.e. creating) the schema from a tree structure produced by the Parse Subsection 4.2.3 library that is capable of parsing structured data.

4.2.6 Transform

Transform library implements the Transform component defined in Subsection 3.3.4. This library tries to mitigate bugs related to not having schema and instances synchronized. It provides factory methods creating conceptual transformations which return transformations for both schema and instances so that after applying these transformations in both schema and instances, they are in sync.

4.3 Servers

In this section we describe the applications that implement the runtime containers defined in Subsection 3.2.5 that represent backend servers apart from analyzers and recommenders that are described in their own sections. The servers are typically HTTP servers communicating with each other using HTTP.

4.3.1 Analysis Store

Analysis Store application is a server implementing the Analyses Store container (Subsection 3.2.5). Its main purpose is to store analyses and provide access to them by their id or their type. All analyses are currently internally stored in MongoDB. It has the following endpoints.

Upload Analyses Endpoint that accepts an array of analyses to store (see Subsection 4.2.1 for more information about analyses). This endpoint is typically used by an analyzer creating analysis. Analyzer Store on receiving such request replaces all analyses in the database for the given combination of dataset and analyzer.

Get Analysis by ID Endpoint that accepts analysis ID in path and returns the given analysis.

Get Analysis by Type HTTP Get endpoint that return all stored analyses of given types.

4.3.2 Analyzer Manager

Analyzer Manager application is a HTTP server implementing Analyzer Manager container (Subsection 3.2.5); therefore, it provides an endpoint for uploading datasets which are then forwarded to analyzer queues. All analyzer queues are implemented using Redis queues.

The request data must be in multipart/form-data format containing an array RDF files and an array of notification requirements. Any DCAT datasets are retrieved and paired with the notification requirements and pushed to all analyzer queues.

4.3.3 Catalog

Catalog application implements the Catalog container (see Subsection 3.2.5). Therefore, its main responsibility is to manage storing and uploading datasets

for analysis. Catalog uses a Virtuoso triplestore database [61] to store uploaded datasets along with received analysis provenance from analyzers.

The catalog supports dataset upload by SPARQL Graph Store HTTP Protocol [62]. The protocol is implemented by creating a proxy using http-proxy-middleware library [63] forwarding requests to Virtuoso which supports the protocol. If Virtuoso responds with success, the uploaded RDF is sent to Analyzer Manager along with a notification request for sending analysis provenance to Catalog.

4.3.4 Recommender Manager

Recommender Manager project implements a server representing Recommender Manager. The server has a single HTTP GET endpoint for recommending based on Editor data. It accepts Schema and Instances which are subsequently forwarded to all configured recommenders. Then, it collects individual recommendations from recommenders and sends them back as a response.

4.4 Editor

Editor is an interactive frontend react single page editor application for semi-automatically transforming the structured data to RDF. Editor heavily relies on Schema, Instances, Transform and Parse packages (Section 4.2) for working with structured data. For the best idea of what features Editor provides, see the example of transforming of food data to RDF in Editor.

Since Editor is an app with rich user interface, we first provide a brief overview of the interface to show the main editor functionality and what kind of visual elements there are. We also list the available actions a user can do in Editor. Then we describe how its implemented.

4.4.1 User Interface

In this subsection we provide an overview of Editor user interface and list any actions users can perform using the user interface. Editor consists of four main visual elements: Diagram, Manual Actions Pane, Manual Actions Header and Recommendations Pane. They are shown in Figure 4.1 which shows the interface after importing food data similar to the data in Motivating example (Section 2.1) but with more food products. Diagram shows the schema of the data (schema is described briefly in Subsection 4.2.5 or in more detail in Subsection 3.3.3) in an interactive draggable graph (i.e. diagram). If a node is clicked, the entity detail is shown on the right in Manual Actions Pane as in the figure Manual Actions Header provides the main menu for what user can do manually. As can be seen in the figure, there are auto layouting options for Diagram, Undo/Redo, Import/Export of data and transformations option such as creating new property sets.

All detailed view for any actions is shown on the right in Manual Actions Pane. There is currently shown the detail of entity set for products. It can be seen that there are listed its property sets and it is possible to set their names or URIs as well as move them to a different entity set. Lastly, Recommendations Pane is shown on the left containing lists of suggested recommendations. The

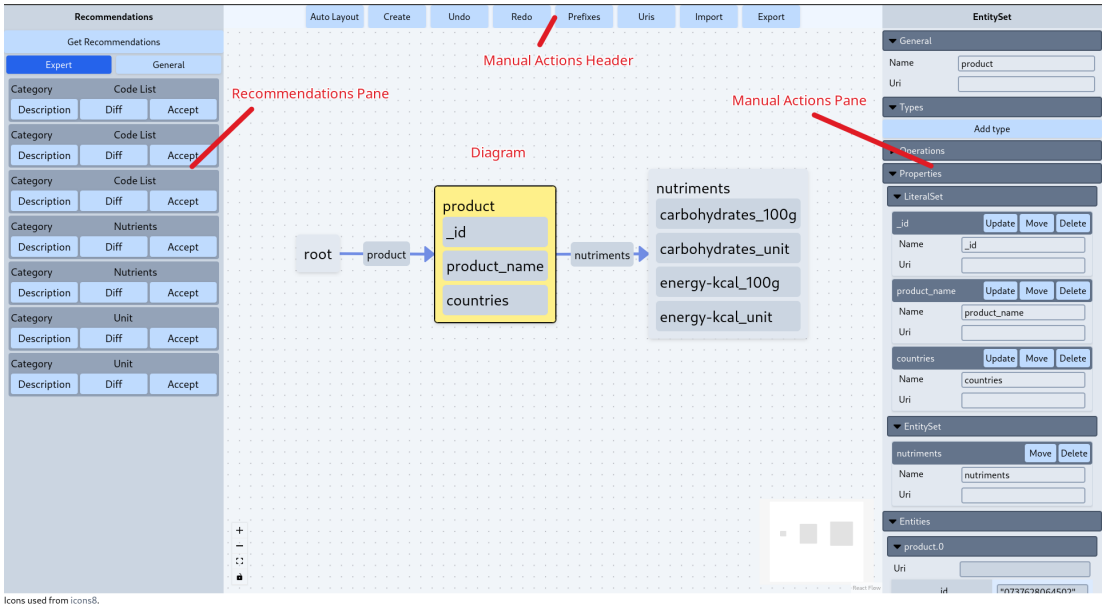

Figure 4.1 Entity Set Detail

Detail and Difference views can be shown upon clicking the corresponding buttons. Accepting recommendation means performing its transformation.

Figure 4.2 shows what the Difference View looks like for a recommendation that recommends code list values for Europa country code list. It contains two diagrams, one for the current state and one for the state if the recommendation is applied. In both cases, the user can click on nodes, their content and edges in the diagram to show how the values change. The recommendation suggests transforming the literal values of countries to code list URIs. On to top, there are shown the literal values of countries literal property set and the new countries entity set pointing to new entity sets containing the code list URIs (Editor supports prefixes; therefore, we created a prefix countries for the URI base for convenience).

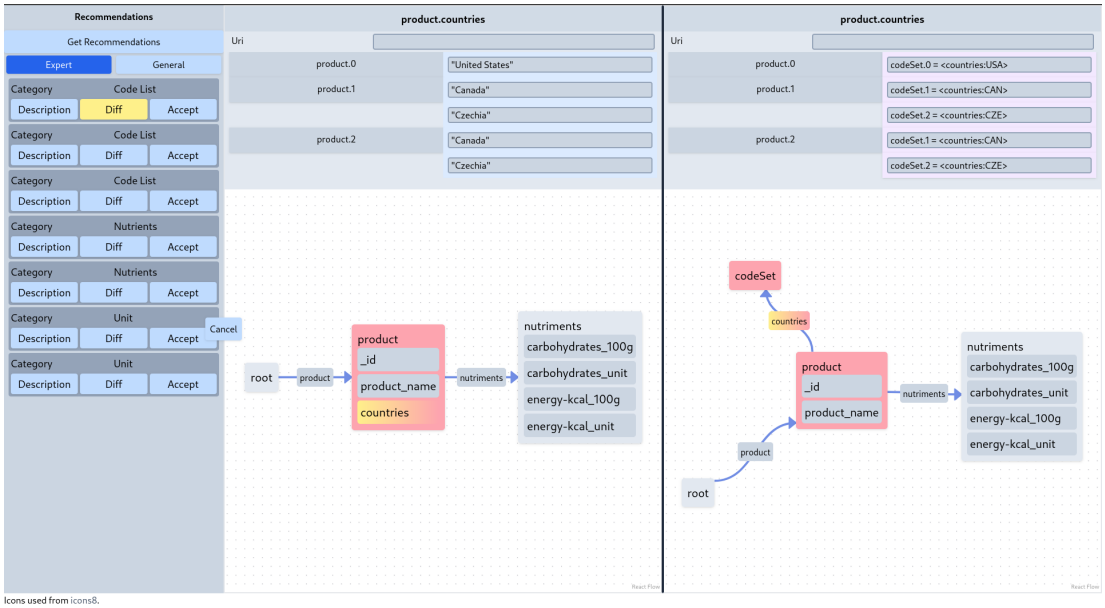

Figure 4.2 Code List Recommendation Difference View

Available User Actions In User Interface

We also include a list of actions that user can do in Editor.

- Move Property Set - for both literal and entity property sets
- Create Property Set - for both literal and entity property sets
- Create Entity Set
- Set instances URIs based on pattern
- Auto layout diagram - vertical, horizontal, radial and force layouts
- Undo/Redo any operation done by user and applied recommendation
- Set prefixes that appear when showing or setting URIs
- Import structured data
- Export all transformations done on the last imported structured data
- Import transformations to be done on the data
- Export Instances, Schema in RDF
- Delete entity sets and property sets
- Update literal values of property sets
- Set type or language to literals
- Set URIs, names to entity sets and property sets, Set type to entity sets
- Browse schema and instances
- Browse recommendations
  - View diagram schema difference view and browse entity and property recommended changes
  - View recommendation descriptions

State Management

Having described the user interface and listed the available user actions, we describe what data are stored as a state and how the state is managed. A state is a data structure that describes the application at any point in time and serves as a data source for the application. Editor stores the following data as a state.

- Schema and Instances created from imported structure data
- Diagram graph created and updated based on Schema
- Information about what operation is open in Manual Actions Pane
- Suggested recommendations

- History of changes to provide undo/redo
- Information about shown help box

It is quite important to keep the mentioned state data consistent with each other. The data can be partially dependent on one another and changing only one can put other data into an inconsistent state. Moreover, it is also important from which UI components are the data used and changed, since that components also need to update the data dependent on the changed data; therefore, they need access to them.

We briefly list the dependencies. If Schema changes, then Diagram must be updated. If a user updates either Schema and Instances manually in Manual Actions Pane, then again Schema, Instances and Diagram must be updated and the pane might be closed. Clicking on the diagram node triggers an entity set detail pane open in Manual Actions Pane. If some transformation operation is open in the pane, the entity set detail must not be opened. There are also options for displaying a help window when more complex interactions are required from a user. Moreover, history needs to know the valid synchronized data states. Only recommendations are used from a single UI component - Recommendations Pane. Still, recommendations are dependent on Schema, Instances and Diagram which are needed to be shown in a recommendation Difference View. Moreover, the recommendation logic transforms them when a recommendation is accepted (i.e. applied).

Not only need the main visual components the data but they need to trigger complex operations on top of them. An example is a Diagram auto layout or a structured data import both provided by Manual Actions Header. Therefore, Schema, Instances, Diagram, History, and Information about Manual Operations data and transformation operations on top of them need to be accessible to practically all visual components.

All the data apart from the recommendations are manipulated in one custom global hook `useEditor` providing custom conceptual functions so that no visual components need to handle any state synchronization. The hook consists of a composition of smaller hooks for each area of functionality (e.g. Diagram). The provided functions are split by their domain but all of them respect the global state. The functions are available to the rest of the application using React Context. The simplified signature of `useEditor` is shown below.

```
useEditor: () => {
    history: { undo, redo },
    schema: Schema,
    instances: Instances,
    diagram: {
        nodes: SchemaNode[],
        edges: SchemaEdges[],
        nodePositioning: { layoutNodes }
    },
    manualActions: { showMoveProperty, showEntitySetDetail, hide },
    help: { showHelp, hideHelp },
    updateSchemaAndInstances
}
```

Since the recommendations are not used in the manual part of Editor, they are stored in a separate state. There is also a custom hook `useRecommendations`

that provides any logic related to fetching recommendations, computing data for Difference View and other recommendation related things. It is again passed to the components working with recommendations by React Context.

4.4.2 Technology Stack

We list the main libraries the editor uses. It also uses the general libraries mentioned in the technology stack of this chapter (Section 4.1).

Vite Editor is bundled using Vite [64] which is a build tool that provides bundling and a development server with hot module replacement.

Tailwind CSS Any layouting and styling is done by using library Tailwind CSS [65].

Reactflow Reactflow [66] is a diagramming library providing diagram functionality a customizable React component. We utilize it to create all diagrams and their interactive capabilities such as dragging.

Elkjs Elkjs [67] is a port of Java Elk library [68] which provides graph layouting algorithms. It is used for the layouting of the main schema diagram.

React Error Boundary React Error Boundary [69] library implements an error boundary component that catches rendering errors and enable handling the errors, for example, with custom UI components informing user that an error happened and providing solutions to handling the errors (e.g. undo the last step) that user can choose from.

React Viewport List React Viewport List [70] library provides a list component that renders only list items that would be actually visible. When user scrolls, the behaviour is the same as having rendered all list items. The main feature of the library in comparison to similar virtual list libraries is that it supports dynamic sizes of items. Instances can be quite large and when rendering just hundreds of items next to a diagram makes the diagram drag operation lag. This situation happens quite often for example in entity set detail or when browsing recommendation difference view; therefore, we employ this library in these places.

4.5 Analyzers

In this subsection, we list all implemented analyzers (i.e. applications implementing Analyzer containers from Subsection 3.2.5) and provide a brief description of how the implemented analyzers work. Any analyzer consumes its analysis job queue to receive analysis jobs. Each analysis job includes a DCAT dataset to analyze and a list of notification requirements. The analyzer inspects the dataset for suitable a DCAT distribution and gets the data. It then analyzes the data and produces analysis which it stores to Analyzer Store. It also needs to send any notifications from the received list of notification requirements such as analysis provenance.

Basically the only thing that is different for each analyzer is the logic for creating analyses based on RDF data. The rest is the same and the Analyzer package (Subsection 4.2.1) can be used to handle it.

4.5.1 SKOS Codelist Analyzer

SKOS Codelist Analyzer detects code lists in dataset data and creates a code list analysis for each detected code list. The detection is done using SPARQL. A code list means a SKOS concept scheme of concepts that contain literal codes at least using `skos:notation` and link to the scheme using `skos:inScheme`.

Each produced analysis typically contains a code list IRI, a code list name and then a list of codes (i.e. SKOS concepts). Each code contains its IRI, its name and an array of literal values that represent the code. These values can be any standard code notations, labels or any derived strings related to the code.

4.5.2 SKOS Concept Scheme Analyzer

SKOS Concept Scheme Analyzer is similar to SKOS Codelist Analyzer but it focuses on finding general concepts schemes and produces analyses representing concept schemes.

It detects concepts schemes that contain concepts linked to the scheme using `skos:inScheme` but no `skos:notation` is required. Other SKOS terms such as `skos:broader` are also detected.

Each produced analysis then contains a scheme with a list of concepts that have fields for hierarchical SKOS properties such as `skos:broader`.

It uses SPARQL and it is very slow; therefore, it is turned off by default.

4.5.3 Elasticsearch Triple Analyzer

Elasticsearch Triple Analyzer combined with Elasticsearch Triple Recommender (Subsection 4.6.3) implement recommending based on a full-text index in Elasticsearch. It is inspired by the search methods mentioned in the analysis chapter (Chapter 2) - namely by the manuscript about using Elasticsearch for search in linked data [26], Swoogle [22] and Falcons [24].

The analyzer part builds the index. Based on [26], the index indexes dataset triples as virtual documents. Each virtual document contains fields for IRI, local names of all IRIs of the triple and a field for literal value if triple's object is literal. Moreover, `rdfs:comment` value of the subject and object of a triple are added to two fields as well. We employ a custom tokenization and filtering on these fields such as using n-gram for local names based on the mentioned manuscripts.

If the same dataset is uploaded multiple times, the virtual documents of the dataset from previous analysis are overwritten. Each produced analysis contains an index name where the corresponding dataset data were loaded into.

4.5.4 Type Map Analyzer

Type Map Analyzer creates for each dataset an analysis representing a map of resources (i.e. IRI) to their RDF types.

4.5.5 RDFS Vocabulary Analyzer

RDFS Vocabulary Analyzer recognizes RDFS vocabulary terms (`rdfs:Class`, `rdf:Property`) in dataset data and creates vocabulary analyses. A vocabulary analysis contains vocabulary class and property terms as well as their hierarchy (i.e. `rdfs:subClassOf`, `rdfs:subPropertyOf`).

4.5.6 Simple OWL Vocabulary Analyzer

Simple OWL Vocabulary recognizes some basic OWL vocabulary terms (i.e. `owl:Class`, `owl:DatatypeProperty`, `owl:ObjectProperty`) and creates the same type of analysis as RDFS Vocabulary Analyzer.

4.6 Recommenders

In this section we list all implemented recommenders (i.e. applications implementing Recommender containers from Subsection 3.2.5) and of how the implemented recommenders work. A recommender exposes a HTTP endpoint that accepts the Editor schema and instances and returns recommendations. It may fetch analyses from Analysis Store (Subsection 4.3.1) if it requires them.

Recommender package (Subsection 4.2.4) contains the data interfaces for recommendations, functionality for running who HTTP server and for retrieving datasets. The implemented recommenders heavily use the package and basically differ only in their logic that creates recommendations from the Editor data.

4.6.1 CodeList Recommender

Codelist Recommender creates recommendations that recommend linking to code list IRIs. It retrieves all code list analyses (currently only produced by SKOS Codelist Analyzer) which contain code list values as well as IRIs. The recommender tries to match the code values to literals in Editor instances. A recommendation transforming the literals to links to code IRIs is created for every match.

4.6.2 Czech Date Recommender

Czech Date Recommender tries to find czech date pattern `"DD.MM.YYYY"` and suggests updating the values to `"YYYY-MM-DD"` with type `xsd:dateTime`.

4.6.3 Elasticsearch Triple Recommender

Elasticsearch Triple Recommender recommends based on analyses from Elasticsearch Triple Analyzer, Type Map Analyzer, RDFS Vocabulary Analyzer and Simple OWL Vocabulary Analyzer (see Section 4.5 for more details about the analyses and analyzers). Elasticsearch Triple Analyzer builds a full-text index on RDF triples. This recommender tries to match search sources to triples in the index. Supported search sources are currently entity sets and property sets. The

matching is done using Elasticsearch `cross_field`[2] query with boosted fields for objects based on [26].

The matches only give matching triples that need to be transformed into useful recommendations. The matched triples do not necessarily contain vocabulary terms or interesting IRIs such as code list codes' IRIs. Hence, we use type and vocabulary analyses to get vocabulary terms (classes, properties) based on which are created recommendations for setting RDF types or URIs.

4.6.4 Food Ontology Recommender

Food Ontology Recommender provides recommendations for the nutrient parts of Food Ontology [11]. The model of the vocabulary is explained in the motivating example in Subsection 2.1.1. First, It finds property sets modeling a nutrient value based on string comparison. For all found property sets, it creates a recommendation that transforms the found nutrient property set and nutrient values to the Food Ontology representation. This type of recommendation is used in Use Case (Chapter 5) for a better explanation.

4.6.5 Uncefact Unit Recommender

This recommender tries to find units (e.g. `kg`, `g`) in literals of Editor instances. For each such unit found, it suggests transforming the units to a standardized UN/CEFACT codes [14].

4.7 User Documentation

In this section we describe the user documentation. There are four main entrypoints to the system. The main one is the transformation of structured data to RDF in Editor (Section 4.4). We do not discuss the transformation process here since there is a detailed use case for transforming food product data to RDF in Chapter 5. The main user interface is also described directly in the editor section (Section 4.4).

The other entrypoints that are actually discussed in this section are uploading dataset for analysis in Subsection 4.7.1, adding a new analyzer in Subsection 4.7.2 and adding a new recommender in Subsection 4.7.3.

4.7.1 Uploading Datasets for Analysis

Uploading datasets for analysis is done through Catalog app (Subsection 4.3.3) which supports SPARQL Graph Store HTTP Protocol [62] on endpoint `/rdf-graph-store`. Since Catalog proxies all requests to Virtuoso [61], the support is as complete as Virtuoso's[3]. That, for example, means that some responses might have not spec compliant return codes. Catalog app typically runs on URL specified by `CATALOG_URL` environment variable.

[2] `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-multi-match-query.html`

[3] `https://vos.openlinksw.com/owiki/wiki/VOS/VirtGraphUpdateProtocol`

The protocol allows update and fetch RDF graph data in RDF Graph store using HTTP. We only highlight the main features that are used to upload, update and retrieve datasets.

Uploading a dataset can be done using HTTP POST or HTTP PUT request on `/rdf-graph-store` endpoint. Datasets are uploaded as a RDF files to either a default graph or a graph specified by URI. Using HTTP PUT request replaces the graph content while HTTP POST adds the data to the graph. Note that in our case specifying the default graph option results in Catalog generating a random graph URI that the RDF data are placed in. The generated graph URI is then returned in response header `Graph-URI`. Uploaded RDF data are sent to Analyzer Manager that retrieves DCAT datasets which are then analyzed.

The RDF data can sent in multipart/form-data under name `res-file` otherwise Virtuoso fails. Alternatively, it can be done using curl [71] program as shown below. Graph data can be retrieved using HTTP GET (also shown below).

```
# Add datasets to a random graph.
curl
  --verbose
  --url "http://{catalog-domain:port}/rdf-graph-store?default"
  -X POST
  -T {dcat-dataset-file}

# Add datasets to graph {URI}.
curl
  --verbose
  --url "http://{catalog-domain:port}/rdf-graph-store?graph={URI}"
  -X POST
  -T {dcat-dataset-file}

# Update graph {URI} with new datasets.
curl
  --verbose
  --url "http://{catalog-domain:port}/rdf-graph-store?graph={URI}"
  -X PUT
  -T {dcat-dataset-file}

# Get graph {URI}.
curl
  --url "http://{catalog-domain:port}/rdf-graph-store?graph={URI}"
```

If one dataset is submitted multiple times, its older analyses are overwritten with new ones.

Browsing Catalog Data

Virtuoso also provides a SPARQL endpoint outside the context of Catalog which administrator can use to browse datasets and related analysis provenance. Virtuoso's URL is set in `VIRTUOSO_URL` environment variable. The SPARQL endpoint would then be available at `{VIRTUOSO_URL}/sparql/`.

Analysis provenance is specified using the PROV ontology [49]. The provenance data represent an analysis as `prov:Entity`. Its IRI is dereferenceable and leads to the full analysis content. Analysis is connected to the `prov:Activity` that generated it using `prov:wasGeneratedBy`. The activity is linked to the analyzed dataset using `prov:used` and to the analyzer that created the analysis using

`prov:wasAssociatedWith`.

4.7.2 Adding Analyzer

One of the main design goals was to make adding new analyzers straightforward. What an analyzer does is described, for example, in Subsection 3.2.5 or in Subsection 4.2.1. In this subsection, we provide instructions how to add a new TypeScript analyzer. We first describe the project setup, then how an analyzer can be built in code. Lastly, we show what needs to be set in the system configuration to run the new analyzer.

We recommend to use the TypeScript language for implementing new analyzers since much helper functionality needed by all analyzers is already provided by Analyzer library (Subsection 4.2.1). Alternatively, it is possible to create an analyzer in any language that implements consumption of a Redis queue and the (HTTP) communication between the analyzer and the other servers it has to communicate with. That mainly includes communication with Analysis Store (Subsection 4.3.1) and sending analysis provenance.

Project Setup

We need to create a new project in directory `apps/analyzers`. The best way is to copy one of the existing projects and change the project name. If that is not desired, there are three main things to set up. We need the following dependencies in `package.json`.

```
{
"dependencies": {
        "@klofan/analyzer": "*",
        "@klofan/config": "*",
    },
"devDependencies": {
        "@klofan/typescript-config": "*",
        "typescript": "^5.0.2"
    }
}
```

Package `@klofan/analyzer` (Subsection 4.2.1) contains analysis definition and functionality for building an analyzer server. Package `@klofan/config` provides checking that environment variables are correctly specified and access to them. Package `@klofan/typescript-config` contains TypeScript configuration to reference in `tsconfig.json` such as below.

```
{
    "extends": "@klofan/typescript-config/base.json",
    "compilerOptions": {
        "rootDir": "./src",
        "outDir": "./dist",
    },
    "include": ["src/**/*"],
}
```

Building Analyzer Server

With the new project set up, we start building the analyzer. We use the described function from Analyzer library in Subsection 4.2.1 that creates a full analyzer server. We can see a mock implementation of an analyzer below. It is only required to retrieve a queue name and a port from the system configuration and then to implement a function that accepts a DCAT dataset and produces internal analyses. An internal analysis differs from the standardly used term analysis in that it is not required to assign ID to the returned analysis or its provenance which is done for us by the `runAnalyzerServer` function.

```
import { DcatDataset,fetchRdfData } from '@klofan/analyzer/dataset';
import { runAnalyzerServer } from '@klofan/analyzer/communication';
import { createLogger } from '@klofan/config/logger';
import { Quad } from '@rdfjs/types';

const logger = createLogger();
const QUEUE_NAME = // <- Configuration
const PORT = // <- Configuration

const analyzeFunction =
  async (dataset: DcatDataset): Promise<InternalAnalysis[]> => {
    const quads: Quad[] = await fetchRdfData(dataset);
    const analyses: InternalAnalysis[] = // <- Quads
    return analyses;
}

// Run server only if QUEUE_NAME is set.
if (QUEUE_NAME) {
  runAnalyzerServer(analyzeFunction, {
    port: PORT,
    jobQueue: QUEUE_NAME,
    analyzerIri: 'http://example.com/analyzer',
    logger: logger,
  });
}
```

Note that although an analyzer does not need to be a server, `runAnalyzerServer` for convenience creates additionaly a simple HTTP server with a single endpoint that accepts datasets and returns analyses using the provided `analyzeFunction`.

Configuration

We now discuss what to add to the system configuration so that the analyzer can be run. A pair of environment variables for the analyzer queue and port must be set. The queue environment variable must have prefix `ANALYZERS_` and suffix `_QUEUE`. File `@klofan/config/src/env/server.ts` contains a Zod [52] schema for the valid environment variables. The new variables must be added - copy and rename how it is done for a different analyzer.

The environment variables can then be retrieved the following way and plugged in the code shown above.

```
import { SERVER_ENV } from '@klofan/config/env/server';

const QUEUE_NAME = SERVER_ENV.ANALYZERS_{NEW_ANALYZER_NAME}_QUEUE
const PORT = SERVER_ENV.ANALYZERS_{NEW_ANALYZER_NAME}_PORT
```

4.7.3 Adding Recommender

One of the main design goals was to make adding new recommenders straightforward. A recommender is a server with a single endpoint that accepts the Editor schema and instances and returns recommendations. In this subsection we provide instructions how to add a new TypeScript recommdender. We first describe the project setup, then how a recommender can be built in code. Lastly, we show what needs to be set in the system configuration to run the new recommender.

We recommend to use the TypeScript language for implementing new recommenders since much helper functionality needed by all recommenders is already provided by the Recommender library (Subsection 4.2.4). Alternatively, it is possible to create a recommender in any language that implements the (HTTP) communication between between the recommender and the other servers it needs to communicate with. This is mainly Recommender Manager (Subsection 4.3.4) that sends requests to recommenders' endpoints to get recommendations.

Project Setup

To add a new recommender, we need to create a new project in directory `apps/recommenders`. The best way is to copy an existing recommender project and change the project name. If the prefferred option is to create a project from scratch, there are two things to set up. The following dependencies in `package.json` are typically necessary for any recommender.

```
{
    "dependencies": {
        "@klofan/analyzer": "*",
        "@klofan/config": "*",
        "@klofan/instances": "*",
        "@klofan/transform": "*",
        "@klofan/recommender": "*",
        "@klofan/schema": "*",
        "@klofan/server-utils": "*",
    },
    "devDependencies": {
        "@klofan/typescript-config": "*"
        "typescript": "^5.0.2"
    }
}
```

Packages `@klofan/instances` (Subsection 4.2.2), `@klofan/transform` (Subsection 4.2.6) and `@klofan/schema` (Subsection 4.2.5) provide functionality for working with Editor data. Package `@klofan/config` validates configured environment variables and gives access to them. Package `@klofan/analyzer` (Subsection 4.2.1) contains analysis interfaces in case the new recommender is to recommend based on analyses. Package `@klofan/recommender` (Subsection 4.2.4) provides the recommendation interface and functionality for building a recommender server. The other important thing to set up is `tsconfig.json`. An example working setup is below.

```
{
    "extends": "@klofan/typescript-config/base.json",
    "compilerOptions": {
        "rootDir": "./src",
```

```
        "outDir": "./dist",
    },
    "include": ["src/**/*"],
}
```

Building Recommender Server

With the project setup, we can build a recommender. `@klofan/recommender` package (Subsection 4.2.4) provides a function that creates a full recommender server. There is a mock implementation of a recommender below. It is only required to retrieve a port and request limit from the system configuration and implement a function that accepts the editor schema and instances and returns recommendations.

```
import { runRecommenderServer } from '@klofan/recommender/server';
import { getAnalyses } from '@klofan/recommender/analysis';
import { Analysis } from '@klofan/analyzer/analysis';
import { createLogger } from '@klofan/config/logger';

const logger = createLogger();
const PORT = // <- Configuration
const REQUEST_LIMIT = // <- Configuration

const recommendFunction = async ({
  schema,
  instances,
}): Promise<Recommendation[]> => {
  const analyses: Analysis[] =
    await getAnalyses(['code-list-analysis'], { logger });
  const recommendations: Recommendation[] = // <- schema + instances
  return recommendations;
}

// Run server only if PORT is set
if (PORT) {
  runRecommenderServer(recommendFunction, {
    port: PORT,
    requestLimit: REQUEST_LIMIT,
    logger: logger,
  });
}
```

Configuration

We now discuss what to add to the system configuration so that the recommender can be run. A pair of environment variables for the recommender port and URL must be set. The URL environment serves for Recommender Manager (Subsection 4.3.4) to know how to send requests to the new recommender. The URL environment variables must have prefix `RECOMMENDERS_` and suffix `_URL`. File `@klofan/config/src/env/server.ts` contains a Zod [52] schema which validates set environment variables. These two new environment variables must be added to the schema. There are already ports and URLs for other recommenders; therefore, copy how it is done for one of them.

We also showed that request limit needs to be retrieved from the system configuration. It is already set under `RECOMMENDER_REQUEST_LIMIT`.

```
import { SERVER_ENV } from '@klofan/config/env/server';

const PORT = SERVER_ENV.RECOMMENDERS_{NEW_RECOMMENDER_NAME}_PORT;
const REQUEST_LIMIT = SERVER_ENV.RECOMMENDER_REQUEST_LIMIT;
```

4.8 Deployment

In this section, we discuss the development and production deployment. We first need to describe the configuration of the system. Then we discuss the production deployment and development deployment.

4.8.1 Configuration

The runtime configuration of the system is done using environment variables. The list of used environment variables can be found in `data/example/dev-env` for the development configuration and in `data/example/docker-env` for the production configuration. These files can be used as they are without any further configuration needed.

We still provide an overview of the environment variables for a possible customization. The environment variables can be split into three groups - general, analyzer and recommender environment variables.

General Environment Variables

Environment variables in the general group define ports and URLs of all databases and apps apart from analyzer and recommenders. Note that the port must match the port in the corresponding URL. It also possible to set size request limits for individual apps. The probably three most configurable environment variables are the following.

```
# Timeout for fetching dataset data in milliseconds
ANALYZER_GET_DATASET_DATA_TIMEOUT=5000
# Timout for sending notifications in milliseconds
# Value 0 means no timeout
NOTIFICATION_TIMEOUT=0
# Base when creating IRIs e.g. for provenance activity
BASE_IRI=http://example.com/
```

Analyzer Environment Variables

This group of environment variables specifies analyzer ports and analysis job queue names. The queue environment variable name must have prefix `ANALYZERS_` and suffix `_QUEUE`. By using this pattern Analyzer Manager knows the names of the queues to send analysis jobs to.

Recommender Environment Variables

This group of environment variables specifies recommender ports and URLs. The port and the port in the corresponding URL must be the same. The URL

environment variable name must have prefix `RECOMMENDERS_` and suffix `_URL`. By using this pattern Recommender Manager knows URLs of recommenders to which they forward recommendation requests.

4.8.2 Production

There is a docker compose file `compose.yaml` in the root of the repository. It contains all required databases and apps defined as services. The data of each database are persisted as a docker volume defined in its compose file in `databases/{db}/compose.yaml`. The following commands can be executed to run the system.

```
# Run all these commands from the repository root

# Copy configuration for production
#   - must be in the root dir and named '.docker-env'
cp ./data/example/docker-env ./.docker-env

# Build images
sudo docker compose --env-file .docker-env build --no-cache

# Run system
sudo docker compose --env-file .docker-env  up
```

Note that the environment file is passed to `docker compose` and also MUST be in the root of the repo with name `.docker-env`.

Running New Analyzer

Running a new analyzer in production requires adding a service into the `compose.yaml` file in the root of the repository.

The service needs to have a port and some additional environment variables to give to the specified dockerfile. Provide the configured port environment variable as port and add default port - take the highest number of analyzer default ports and add one. The additional environment variables are `APP_PROJECT_NAME` and `APP_DIRECTORY`. Paste the template below to the `compose.yaml` as a new service and fill out the squared brackets.

```
{new-analyzer-name}:
    build:
      context: ./
      dockerfile: backend-dockerfile.dockerfile
    restart: always
    environment:
      APP_PROJECT_NAME: {analyzer-project-name}
      APP_DIRECTORY: analyzers/{analyzer-project-directory}
    env_file: .docker-env
    ports:
      - "${{conf-port}:-{inc-port}}:${{conf-port}:-{inc-port}}"
```

Running New Recommender

Running a new recommender in production requires adding a service to the `compose.yaml` file in the root of the repository.

This new service must have set exposed port and a few environment variables for Dockerfile. The template of the service is shown below. When something is in curly brackets, that means to substitute its value. `conf-port` means the configured recommender port and `inc-port` is a default port that must be set as unique among the services. The convention for the value of the `inc-port` is to take the maximum default port of recommender services and increment it by one.

```
{new-recommender-name}:
    build:
      context: ./
      dockerfile: backend-dockerfile.dockerfile
    restart: always
    environment:
      APP_PROJECT_NAME: {recommender-project-name}
      APP_DIRECTORY: recommenders/{recommender-project-directory}
    env_file: .docker-env
    ports:
      - "${{conf-port}:-{inc-port}}:${{conf-port}:-{inc-port}}"
```

4.8.3 Development

The system is run in development using Turborepo [50] to run apps and docker compose to run databases. The following commands describe how to run the system in development.

```
# Run all these commands from the repository root

# Copy configuration for development
#   - must be in root dir and named '.env'
cp ./data/example/dev-env ./.env

# Run databases in using docker
sudo docker compose -f db-compose.yaml --env-file .env up

# Wait until databases are running

# Install dependencies
npm ci # or npm install

# Run services in dev mode
npm run dev
```

Note that the environment file is passed to `docker compose` and also MUST be in the root of the repo with name `.env`.

If new analyzers or recommenders are added, `npm install` must be run from the repository root to have Turporepo notice the new applications. It then runs them automatically if their configuration is set correctly.

5 Use Case

In this chapter we illustrate how the system is used to handle its main functionality - enabling the transformation of structured data to RDF in an interactive environment while suggesting transformation recommendations. We take the food data used in Motivating Example (Section 2.1) and showcase in the form of a tutorial how to transform them to RDF using our system, namely Editor application.

Before the tutorial begins, the system must be deployed locally. Therefore, in Section 5.1 we link deploy description and provide the datasets based on which recommendations are done and which; therefore, must be loaded into the system before any step in the tutorial is done.

Once the system is running, the tutorial can be done. It consists of three main parts described in separate sections. First, we describe the input data we later load to Editor in Section 5.2. Then, in Section 5.3 we show how Editor is used to transform the data to RDF using screenshots from Editor. Lastly, we provide the result RDF data in Section 5.4. We run Editor in Firefox.

5.1 Deploy System Locally and Upload Datasets

In this section we describe how the system can be locally deployed and which datasets should be loaded to it to be able to reproduce the steps in the upcoming tutorial.

5.1.1 Deploy System Locally

The system can be deployed using docker compose[1]. The exact details are discussed in Subsection 4.8.2.

5.1.2 Upload Datasets

The datasets needed for the tutorial are saved at `data/example/catalog.ttl` from the root of the repository. If Catalog is running on its default port `7000`, the datasets can be uploaded by running the following command (shorten it to one line or add `backslash` on line ends) from the repository root.

```
curl
  --verbose
  --url "http://localhost:7000/rdf-graph-store?default"
  -X POST
  -T data/example/catalog.ttl
```

Note that one of the datasets is a code list of countries that has around `2MB` and if the network is slow, it may fail to be fetched and analyzed due to a timeout. Therefore, if there is a different number of recommendations related to code lists than in the screenshots during the tutorial, run the command again. Running the command again makes any newly created analyses overwrite the ones created in

[1] `https://docs.docker.com/compose/`

previous runs, hence; there is no data duplication. If there are still no recommendations for the countries code list, set `ANALYZER_GET_DATASET_DATA_TIMEOUT` environment variables to a higher number than default `5000`. It sets the timeout in milliseconds for the request fetching dataset data.

5.2 Input Data

In this first tutorial section, we describe the input structured data to load to Editor. The data are the same food data as in Motivating Example (Section 2.1) with the exception of having not one food product but three in order to show how data with multiple instances (e.g. arrays with more than one element) can be manipulated and browsed in Editor.

The data are available at `data/example/food3.json` from the repository root and they are also shown below. There is a global array of some objects that contain a property `product` linking to a food product object. Each food product object has an identifier, name, list of countries where it is sold and its nutrient information. The data are from OpenFoodFacts [5] and we kept the original structure shown below.


```
[
  {
    "product": {
      "_id": "0737628064502",
      "product_name": "Thai peanut noodle kit",
      "countries": ["United States"],
      "nutriments": {
        "carbohydrates_100g": 71.15,
        "carbohydrates_unit": "g",
        "energy-kcal_100g": 385.0,
        "energy-kcal_unit": "kcal"
      }
    }
  }, {
    "product": {
      "_id": "0737628064503",
      "product_name": "Snickers bar",
      "countries": ["Canada", "Czechia"],
      "nutriments": {
        "carbohydrates_100g": 90.15,
        "carbohydrates_unit": "g",
        "energy-kcal_100g": 383.0,
        "energy-kcal_unit": "kcal"
      }
    }
  }, {
    "product": {
      "_id": "0737628064504",
      "product_name": "Milka chocolate",
      "countries": ["Canada", "Czechia"],
      "nutriments": {
        "carbohydrates_100g": 52.15,
        "carbohydrates_unit": "g",
        "energy-kcal_100g": 343.0,
        "energy-kcal_unit": "kcal"
      }
```

```
    }
  }
]
```

5.3 Transformation

In this section we present how the whole transformation of the food data to RDF can be done in Editor in the form of a work along tutorial. There are Editor screenshots showing the Editor states as we perform actions in Editor. They are listed chronologically from top to bottom. While the screenshots are referenced as figures by numbers, the reference typically points the closest figure above or below the reference point. The text on the following pages might end prematurely with blank space filling the rest of a page so that texts for multiple screenshots are not hoarded next to each other with figures following in the next pages.

We start by opening Editor on `localhost/index.html`. Figure 5.1 shows the initial state of Editor with no data after opening it.

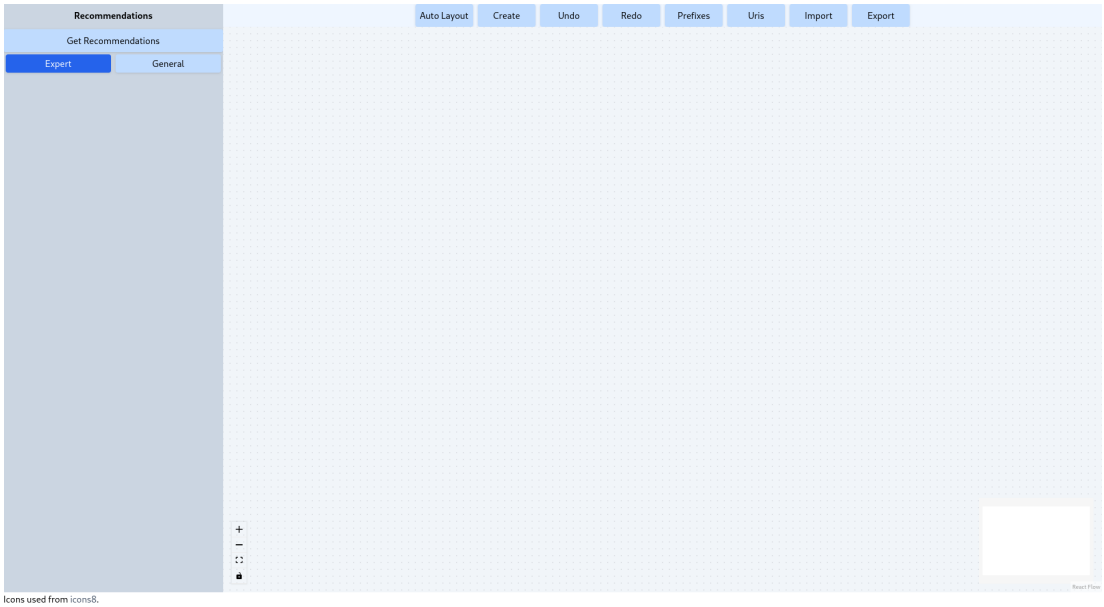

Figure 5.1 Initial State

We use the `Import` button in the header on the top of the page and upload the file with the food product data described in the previous section located at `{repository}/data/example/food3.json`. The import functionality currently supports uploading JSON and CSV files.

The import produces some boxes and arrows that are stacked on top of each other. We hover on the `Auto Layout` button and choose `horizontal layout` which layouts the boxes and arrows as shown in Figure 5.2 (the pane on the right should not be visible nor the yellow coloring). The rest of the options layout the boxes using different algorithms. Alternatively, we can manually drag and drop the boxes to create the desired layout.

Now each box and arrow is visible; therefore, we can explain what they mean and how to view the original data. Each box represents an object or an array of objects at a given position in the input JSON file. For example, the box

`product` represents the array of three products or the box `nutriments` represents the three nutrient objects linked to each product object using the `nutriments` property. If a food product object contained an array of two nutrient objects in the `nutriments` property, then the `nutrient` box would represent four objects. Note that the arrow between the boxes (from `product` to `nutriments`) has name `nutriments`. All of the arrows were created from properties to other objects. The other properties containing a literal value (e.g. string or number) are listed directly in the boxes such as `product` containing `product_name`.

The original data can be viewed by clicking on the boxes. For example, we click on `product` and a pane on the right should appear as in Figure 5.2. The pane has several drop-down sections that expand/shrink when clicked on. They are identified by the black triangles. We also close the `properties` drop-down by clicking it. There are a few terms such as `Entities` that will be explained in due time.

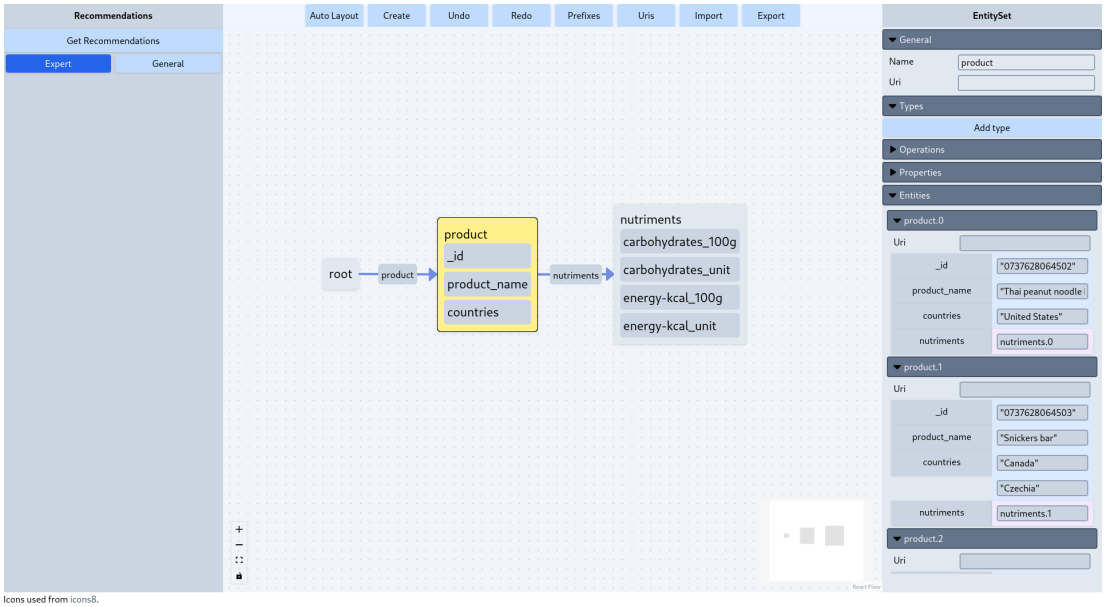

Figure 5.2 Product Detail

The original data are visible under the `Entities` drop-down. We can see the literal values of `product` objects in the read-only inputs with blue backgrounds. The inputs with pink backgrounds represent how the object references another one. For example, the visible input is for the `nutriments` property that points to the corresponding nutrient object in the original data. That is represented with value `nutriments.0` which means that the nutrient object is the first object (indexed at 0) in the array of nutrient objects represented by `nutriments` box.

To summarize, the diagram of boxes and arrows represents the structure of the data where boxes are arrays of objects (or single objects) and arrows represent their relationships. We provide a specialised terminology used throughout Editor so that we do not have to use terms such as boxes and arrows which represent mainly the visual representation. Terms Entity, Literal, Property represent original data. Terms Entity Set, Literal Set and Property Set describe the data structure.

Entity Entity corresponds to an object within a box of objects (i.e. an object in the original data). The content of the drop-down `Entities` is a visual

representation of entities. Each entity can have its specific URI set.

Literal Literal represents a literal value (e.g. string, number) - the input with a blue background.

Property Property is created from an object property from the original data. Each property points to either a literal or an entity.

Entity Set Entity Set corresponds to the described box. It represents a set of entities. An entity set also contains property sets. For example, `product` in the diagram is an entity set.

Literal Set Literal Set corresponds to a set of literals where the literals are values of properties with the same name for the entities of the same entity set. For example, a set of product names.

Property Set Property Set corresponds to the described arrow - a set of properties. The value of property set is either a literal set or an entity set. All of the properties in a property set point to literals or entities of the value of the property set. For example, both `nutriments` and `product_name` are property sets.

Entity sets and property sets are shown in the diagram. Entities, properties and literals can be viewed in the pane on the right in the `Entities` section. Any transformation operation a user can do is based on these terms. Note that even if we say that the defined terms represent the original data, the original uploaded data are converted to the described model and not preserved. Another terms used further in the tutorial are nodes which represent the diagram boxes (resp. entity sets) and edges which represent diagram arrows (resp. entity property sets).

Currently, if we exported the data to RDF, each entity would be represented by a blank node and some example property URIs based on property set names would be used to represent their properties. Therefore, one of the goals of the transformation is to assign URIs to both entities and properties. Setting a property set URI makes all its properties have the set URI when transformed to RDF. Entity URIs can be set in two ways. Either we can set an entity set URI which results in entities using the URI as a base to create their URIs or we can set entity URIs explicitly. We also typically want to set RDF types to entities which is done by adding a type to an entity set in the `EntitySet` pane (currently shown in Figure 5.2). Moreover, we typically want to change the structure of the data and add language tags or types to literals. How all of these manual actions are done along with some additional features is described in the rest of the section.

Now that we described how Editor works with data, we can start the actual transformation to RDF. Editor supports the concept of recommendations. A recommendation provides a suggestion on how to transform the data in Editor. For example, it can recommend to change the structure of the data represented in the diagram or change literal values. We try to see first if there are any recommendations available by clicking on the `Get Recommendations` button on the left pane called `Recommendations`. Note that this operation can take some time. We did not implement messaging the user that recommendations are being fetched since it is done instantly for us. The outcome is shown on Figure 5.3.

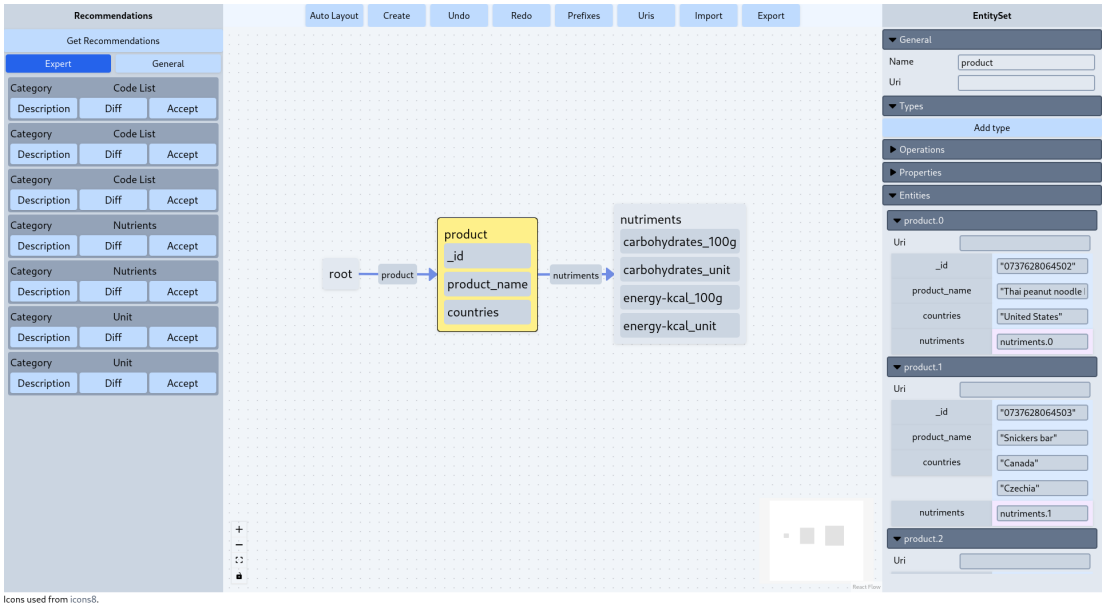

Figure 5.3 Recommendations Fetched

The Recommendations pane in Figure 5.3 shows a list of suggested transformation recommendations. The recommendations can either be expert or general each displayed when its tab is open. Both types work the same way in Editor. Expert recommendations should be more targeted to a use case while general recommendations are more general-purpose. Rather than describing recommendations and their features in text, we click on the `Description` button of the first recommendation to view what the recommendation does. The outcome is shown in Figure 5.4.

Figure 5.4 Country Code List Recommendation Description View

Figure 5.4 shows the description view layout and content on an example of a code list recommedation suggesting to use code URIs from the code list in `Related` section. Each recommandation description typically contains three sec-

tions: `Description`, `Recommended Terms` and `Related`. The first is `Description` which provides some information about what each recommendation suggests. `Recommended Terms` section contains a list of URIs that the recommendation suggests using. `Related` section contains links to concepts mentioned in `Description` and any related terms. All URIs are clickable for dereferencing.

The description view is meant to provide an overview about the recommendation and provide links to terms that the user can investigate and decide if they want to use them. For more detailed information about suggested data transformation, we click on the corresponding `Diff` button yellowed in Figure 5.5 to get a difference view of the recommendation.

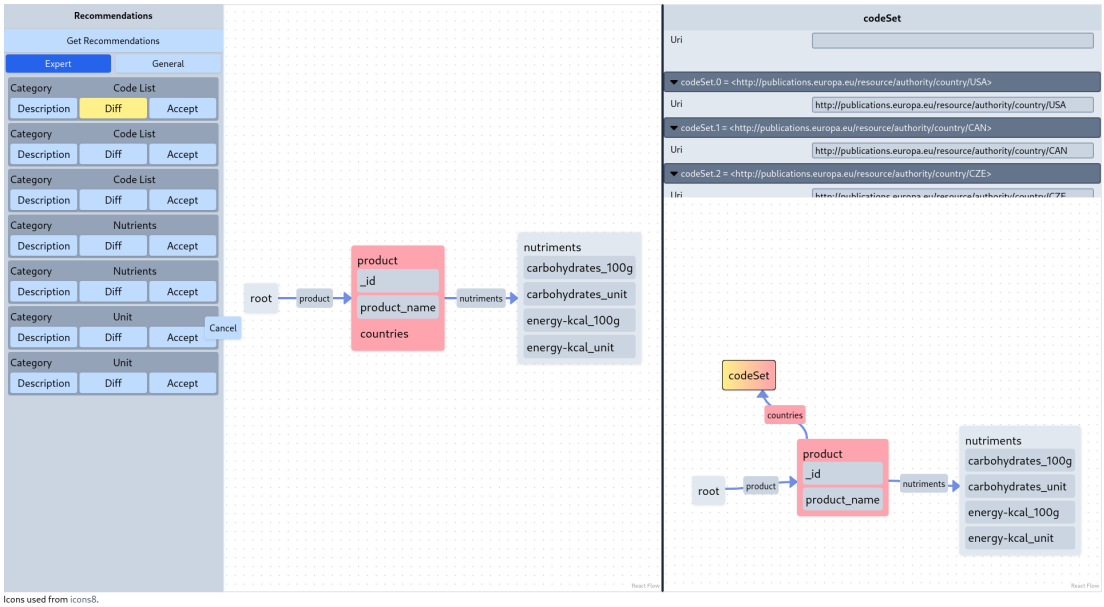

Figure 5.5 Code List Difference View - `codeSet` Entity Set Detail

Figure 5.5 shows the difference view of the recommendation. There are two diagrams with exactly the same meaning (i.e. showing data structure) as the main diagram from before. The left diagram shows the state of data as they are now. The right side diagram shows the state were we to transform the data according to the recommendation. The red entity sets and property sets represent changes between the two states. Any nodes and edges in the diagrams can be clicked on to view their data. A yellow color is used for the currently selected (i.e. clicked) diagram elements whose detail is shown above the corresponding diagram. In the figure, the `product` entity set contains changes as well as its property `countries`. We see that in the new state there is a new entity set `codeSet` and entity property set `countries` while the original literal property set `countries` is missing.

To get to the same screen as is portrayed in the figure, we click on `codeSet` which results in a pane appearing above the diagram which shows what the added entity set represents and what entities it contains. We see that its entities are countries and their URIs correspond to the URIs of Publications Europa country code list[2].

[2] `https://op.europa.eu/en/web/eu-vocabularies/dataset/-/resource?uri=http://publications.europa.eu/resource/dataset/country`

We investigate the changes further by clicking on `countries` in both diagrams (Figure 5.6).

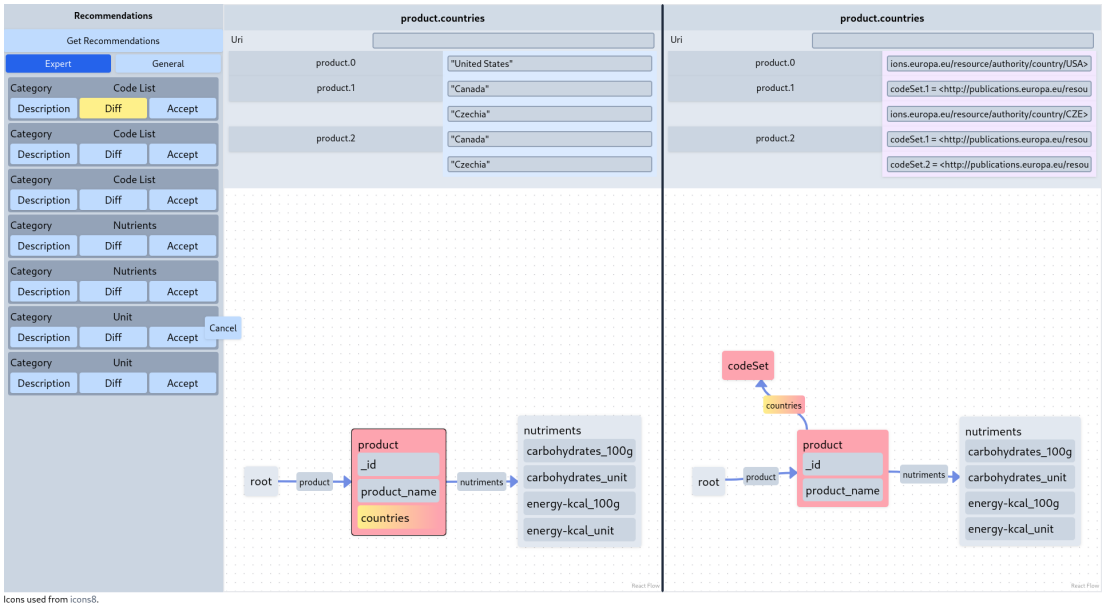

Figure 5.6 Code list Difference View - `countries` Property Sets

Figure 5.6 shows the difference between the original literal property set `countries` and the new entity property set `countries`. We can see that the countries represented originally by literals such as `"United States"` are represented by entities representing countries with well-known URIs such as `country:USA`[3].

We deem that representing countries using well-known URIs is better than using literals; therefore, we click on the `Accept` button of the recommendation which triggers the suggested transformation to be done on our data. With some manual layouting and by clicking on the newly created `codeSet` entity set node, we get the view presented by Figure 5.7. In the figure there are URIs represented using a prefix. It was not done automatically. Instead, we manually added a prefix for the code list after applying the recommendation.

[3] `https://publications.europa.eu/resource/authority/country/USA`

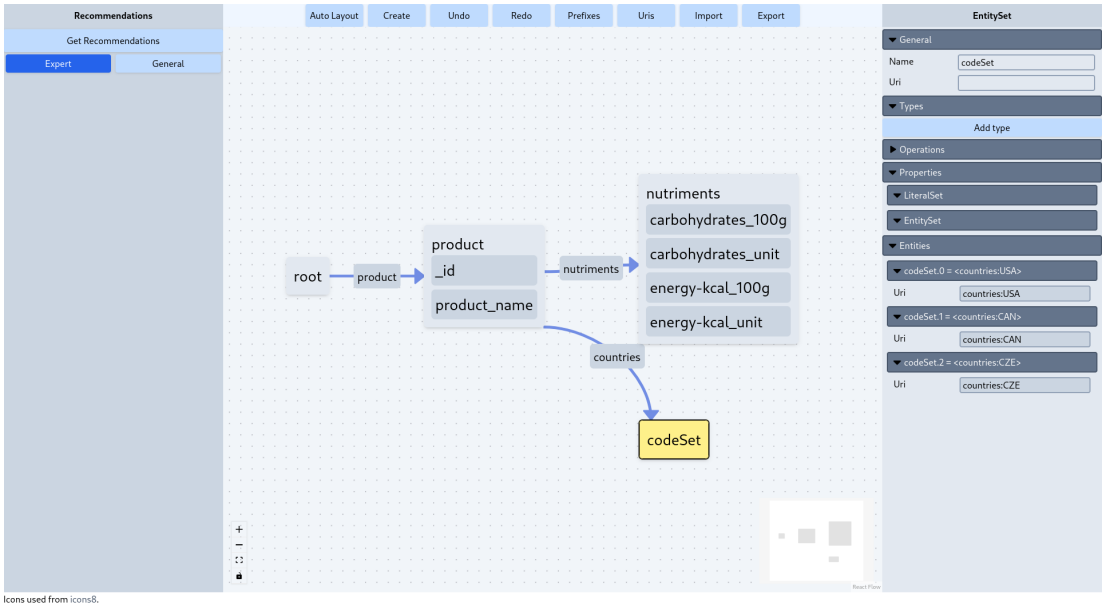

Figure 5.7 Applied Country Code List Recommendation

Adding a prefix can be done by clicking on the `Prefixes` button in the top header and filling out the `Add Prefix` form to get the view as in Figure 5.8. The prefixes are then used instead of the full URIs whenever any URI is shown. Moreover, when setting URIs manually in inputs, available prefixes are suggested in a drop-down and user can write the usual prefix syntax to set URIs. After creating the prefix, the Editor should be the same as in Figure 5.7 after clicking again on `codeSet` entity set node.

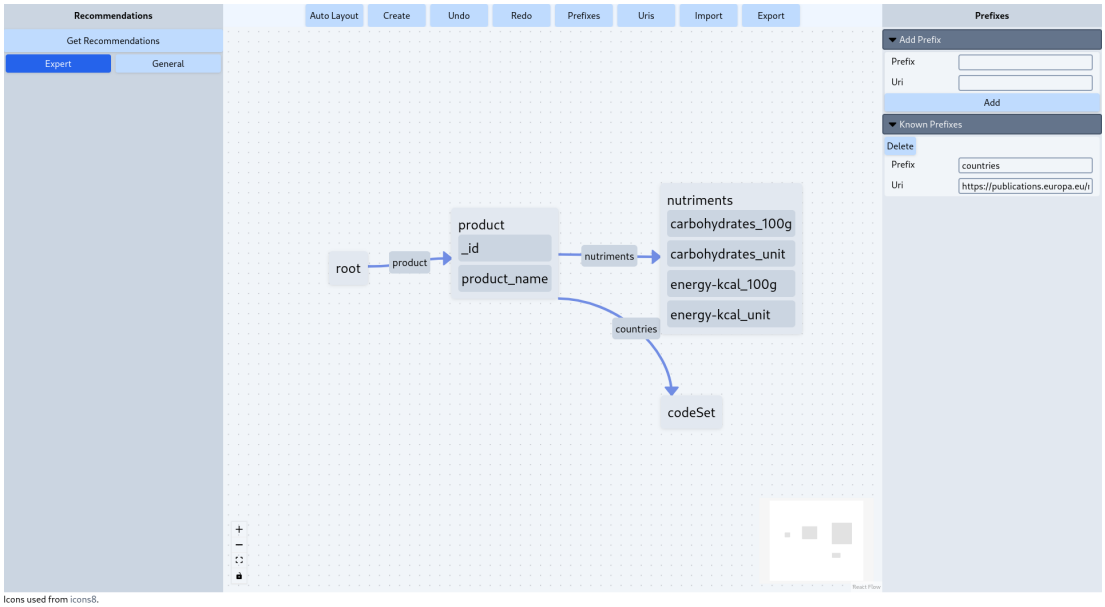

Figure 5.8 Country Code List Prefix Added

We continue with the transformation of data. We click on the button `Get Recommendations` again to get recommendations. Note that the code list recommendations are not present since our data not longer contain country literals. The topmost recommendation has category `Nutrients`. Since our data contain

nutrient information we investigate the recommendation and click on `Diff` to view its difference view (Figure 5.9 without the detail panes on the top).

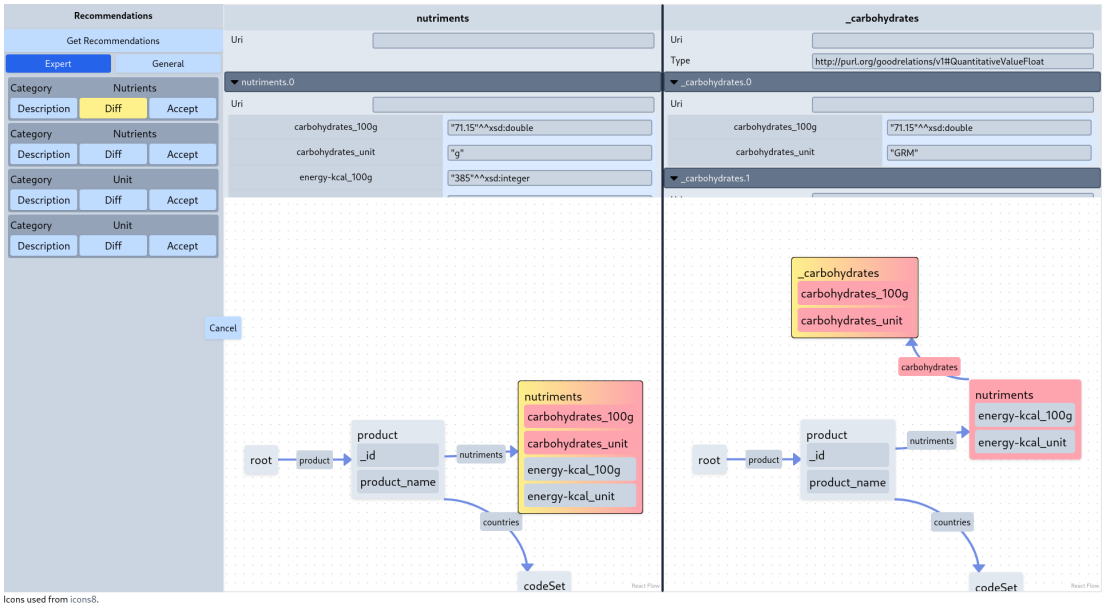

Figure 5.9 Carbohydrates Nutrient Recommendation Difference View

Figure 5.9 shows the difference view for the topmost recommendation that suggests how to represent the data about carbohydrates. We can see that the carbohydrates value and unit property sets are recommended to be moved to a separate entity set. We click on the new entity set with the carbohydrates property sets and on the `nutriments` node as shown in the figure (our Editor view should be the same now as in the figure).

By comparing the detail panes, we see that `_carbohydrates` entity set contains type `QuantitativeValueFloat`[4] from Good Relations [12] vocabulary and that the literal for unit changed from `"g"` to `"GRM"`.

When we click on the property `carbohydrates` that points to `_carbohydrates`, we can see that its URI is set to `carbohydratesPer100g`[5] from Food Ontology vocabulary [11]. Clicking on literal property sets of `_caborhydrates` shows that they also have URIs from Good Relations vocabulary.

Since we do not have to know these vocabularies, we investigate further by clicking on the `Description` button of the recommendation which results in Editor showing a description view shown in Figure 5.10.

[4] `http://purl.org/goodrelations/v1#QuantitativeValueFloat`
[5] `http://purl.org/foodontology#carbohydratesPer100g`

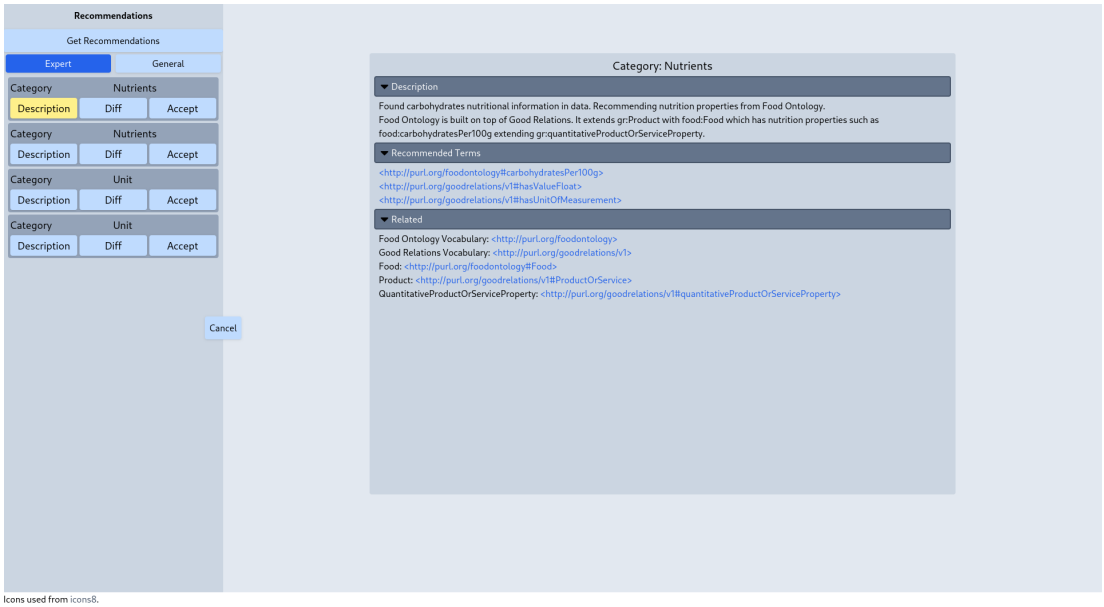

Figure 5.10 Carbohydrates Nutrient Recommendation Description

Figure 5.10 provides of a brief explanation of the recommendation for Food Ontology vocabulary and provides links for the recommended and related terms. We can either trust the recommendation and accept it or investigate the links. The investigation yields that Food Ontology (`food` prefix) is based on Good Relations (`gr` prefix) and that there is a type `food:Food` that is a subclass of `gr:ProductOrService`. Moreover, we find out that nutrition properties should be added to a resource with `food:Food` type. The model of Food Ontology is described in the Motivating Example (Subsection 2.1.1. We accept the recommendation and add prefixes for both vocabularies. The types of `_carbohydrate` and URIs of its properties are accessible in its detail pane that is shown when the node is clicked on (Figure 5.11).

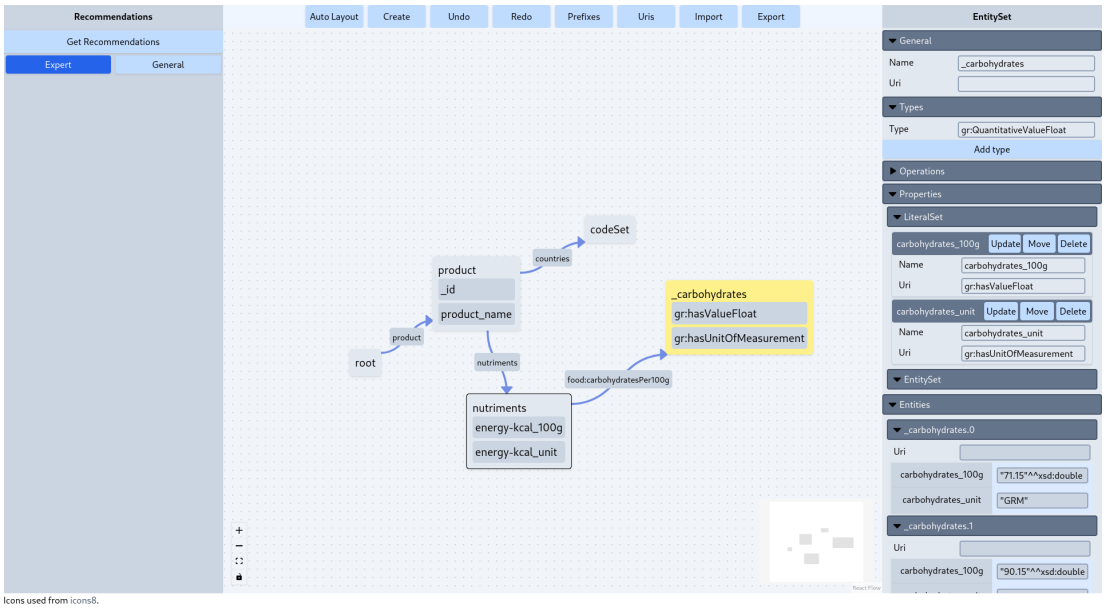

Figure 5.11 Editor After Carbohydrate Recommendation with Prefixes

We continue by again clicking on the `Get Recommendations` button and select the new topmost nutrient recommendation's difference view. If it does not look like in Figure 5.12, then use the other nutrient recommendation (there should be two). It is similar to the previous carbohydrate recommendation but for the other property sets related to energy.

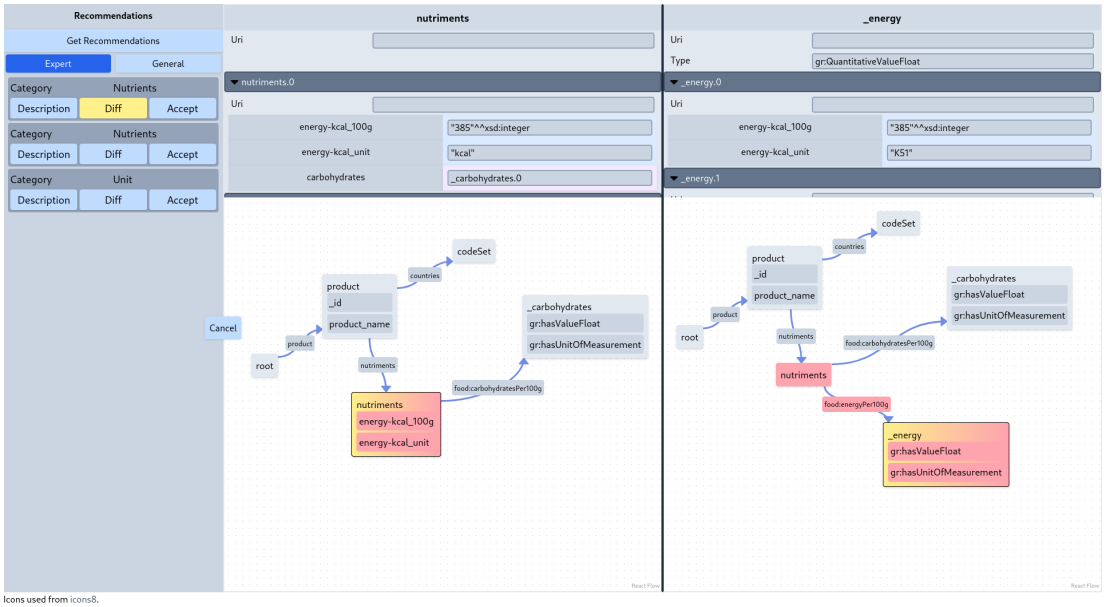

Figure 5.12 Editor Energy Nutrient Recommendation Difference View

We accept the recommendation the get the Editor diagram view as in Figure 5.13 without the right pane. Note that the diagram does not show property set names but rather prefixed URIs if property sets have URIs and the corresponding prefixes are set. Based on our investigation we know that the nutrient properties should origin in a food product entities. Currently, their origin is the `nutriments` entity set. Therefore, we want to move them to the `product` entity set. We start by clicking on `nutriments` showing its detail (Figure 5.13).

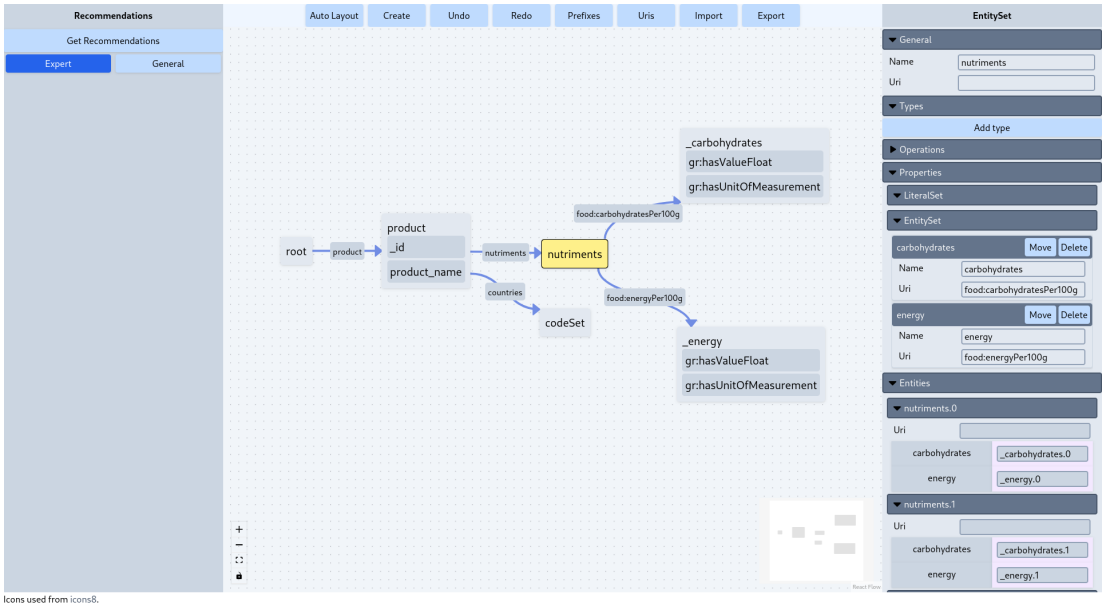

Figure 5.13 Editor after Nutrient Recommendations

Then, we click on the `Move` button under the section `Properties` for the property set (and its underlying properties) `carbohydrates`. The right pane changes and `Move PropertySet` pane appears. We click on the section `Original Mapping` and get to the same state as is shown in Figure 5.14 without the bottom diagram having any edges (i.e. arrows).

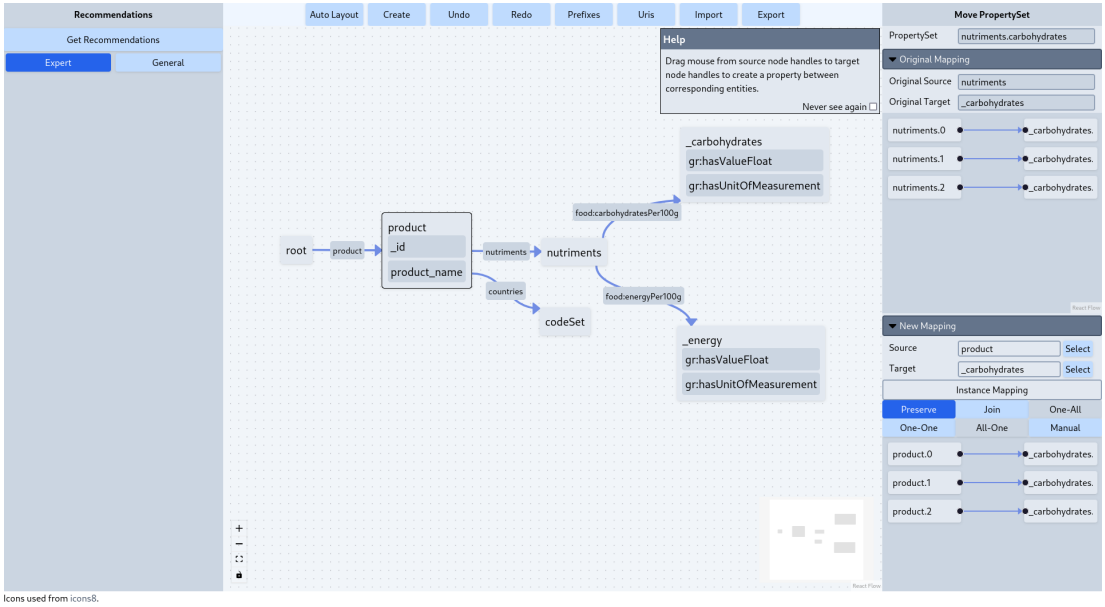

Figure 5.14 Move Carbohydrates Property Sets

Figure 5.14 shows how property sets and their properties are moved from one source or target entity set to another. The section `Original Mapping` represents how the properties between source and target entities are linked (mapped) which is represented by the diagram. The section `New Mapping` allows us to specify a new source or a new target or both. We click on the button `Select` next to the `Source` and click on the `product` node. By doing this the source is set to `product`

as it is shown in the figure. The target remains the same. We also need to specify how the properties between the selected source' and target' entities are mapped.

There are generally six options out of which only the four are available in this case. The `Manual` option lets us drag edges manually. However, if we only loaded some small example data and we potentially want to run the same transformations on larger data in future (executed transformations can be exported and imported), using the manual option is not feasible. The other options are suitable even for that purpose since they provide a rule based mapping to the underlying transformation algorithm. Mapping options `OneOne`, `OneAll`, `AllOne` are simple and work as one would expect. The `Join` mapping provides an option to map entities based on inner join of literal properties of the source and target. The option that we actually need in this scenario is named `Preserve`. The preserve option checks that the original and new sources have the same number of entities as well as that the original and new targets have the same number of entities. If so, it copies the mapping based on the original mapping shown in the diagram above. Both `nutriments` and `product` have both three entities and there is the property set `nutriments` between them which assigns each product one entity (i.e. the mapping is one to one). Therefore, we use the preserve option by clicking on the `Preserve` button and scroll down to click on the `Ok` button.

We also move the property set `food:energyPer100g` to `product` which gives us the diagram shown in Figure 5.15.

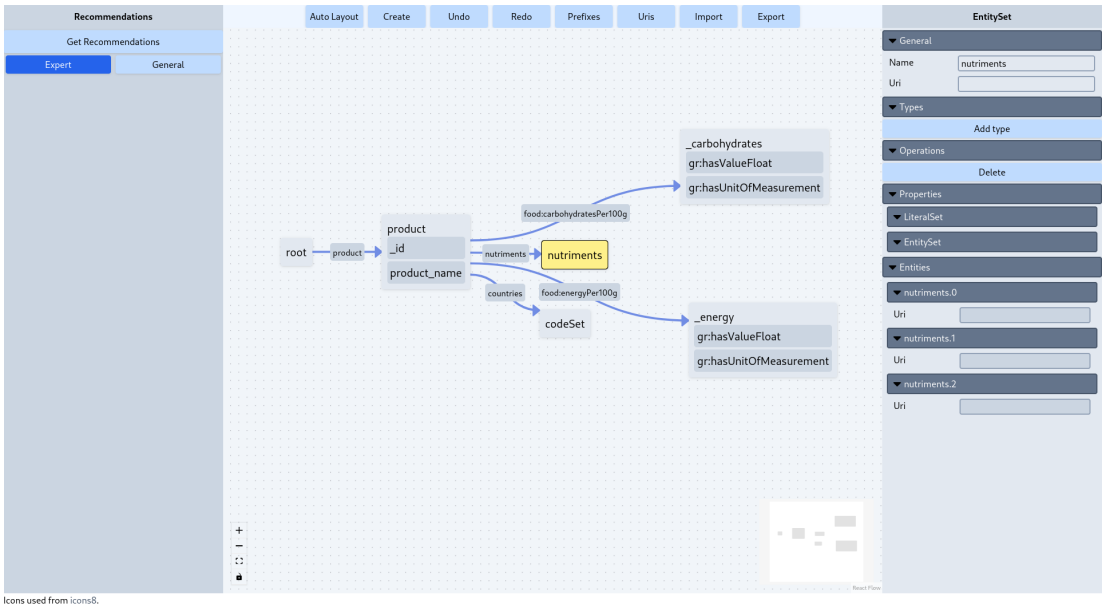

Figure 5.15 Editor After Move Carbs and Energy to Food

We now see that `nutriments` entity set is useless and we delete it. We click on it and click on the section `Operations` in the right pane. There should be a button `Delete` in the section `Operations` that is shown in Figure 5.15. We click on it and the `nutriments` entity set along with all associated property sets are deleted.

The nutrients properties (property sets) are now connected to the `product` entity set. We know that the sources of these properties should have types `food:Food`. So we click on `product` and click on `Add Type` and write `food:Food`

into the appeared input and hit enter. The end result is shown in Figure 5.16. Adding a type to an entity set means that all underlying entities will have the type in the final RDF.

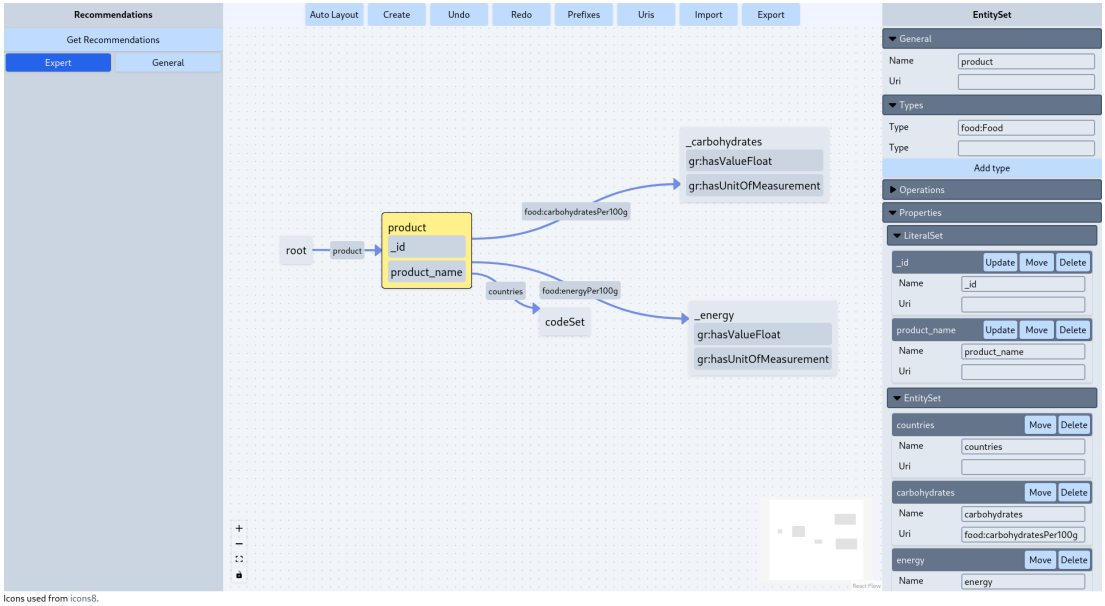

Figure 5.16 Added Food Type

We have yet to set URIs to the `product` literal property sets `_id` and `product_name`. We do it manually to show how it is done. When we did our investigation of Food Ontology while deciding whether to apply the carbohydrates recommendation, we found that `gr:ProductOrService` can have a property `gr:name` defining a product name and that `gr:ProductOrService` is the same as `schema:Product` which has a `schema:ProductID` property for defining a product ID.

Therefore, we click on `product` to open its detail and write the URIs in the corresponding property sets' inputs. The end result is shown in Figure 5.17.

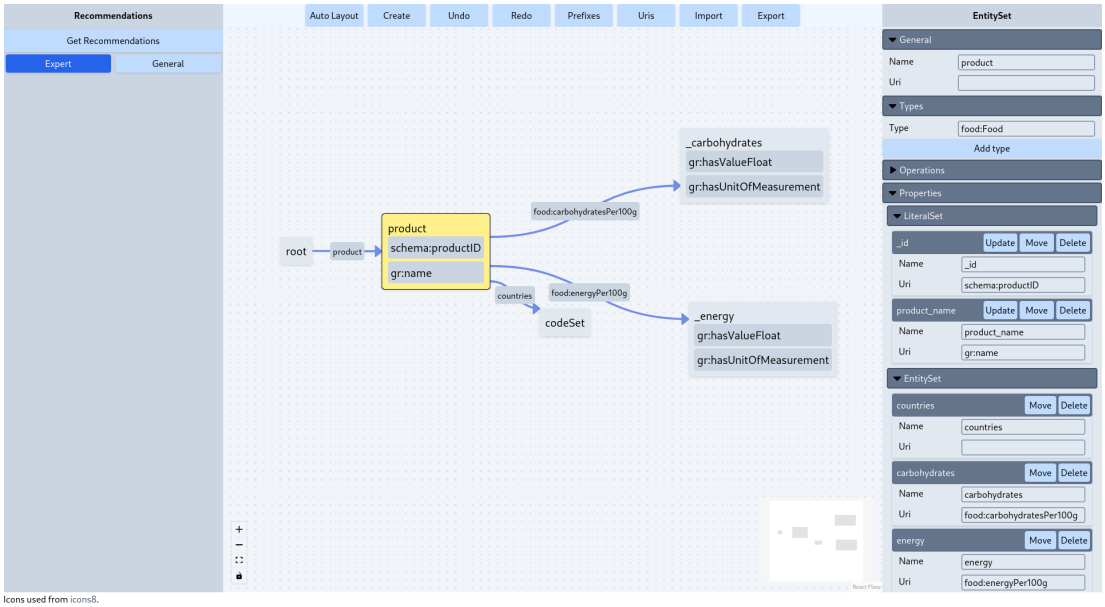

Figure 5.17 Editor Add Schema ProductID

We are almost done. We delete `root`. Then, we need to assign URIs to the `product` entities, assign language tags and set the URI of the `countries` property set. We start by setting URIs of `product` properties. We click on the `Uris` button in the header and the right pane should change to `Update Entity Uris`. There we click on `Select` and then on `product`. We can now set the URI pattern that is applied to all entities. We can compose it of text strings and literal property values. The inputs can be dragged over each other to change the ordering. We fill out the pattern according to how it is done in Figure 5.18. Below, we can see the entities' URIs. Then we click on `Ok` to set the URIs.

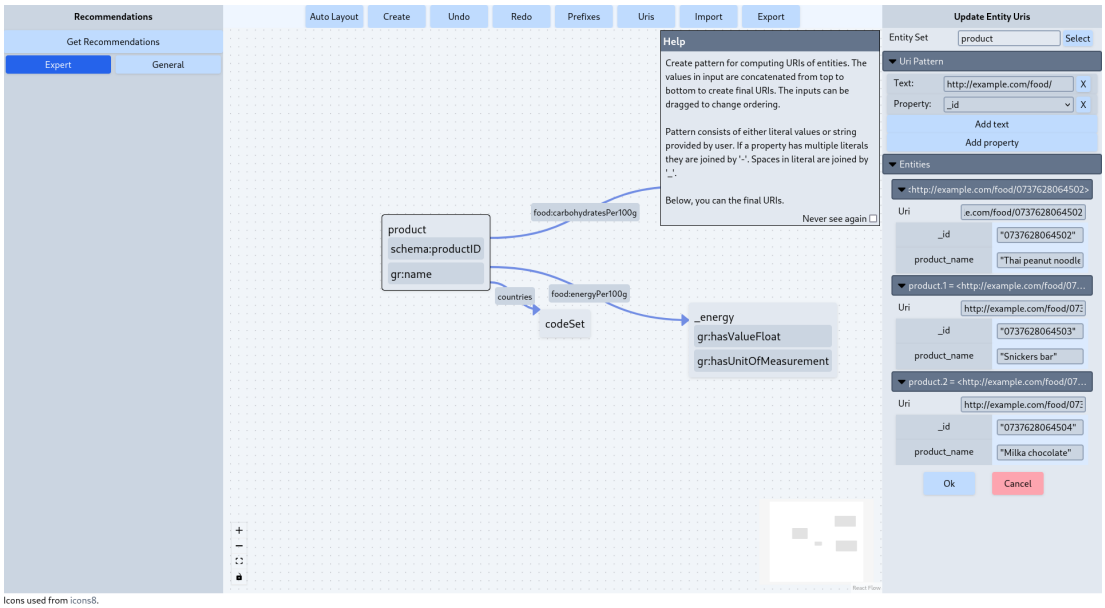

Figure 5.18 Set Food Product URIs

We continue by assigning language tags. The only literals that need a language tag set are literals of the property set `gr:name`. We click on `product` again and

click on the `Update` button next to the property set `gr:name`. We should be able to see the pane `Update Literals` (Figure 5.19). We write `en` to the input for `Language` and click on `Ok` to set the language tag `en` to all literals.

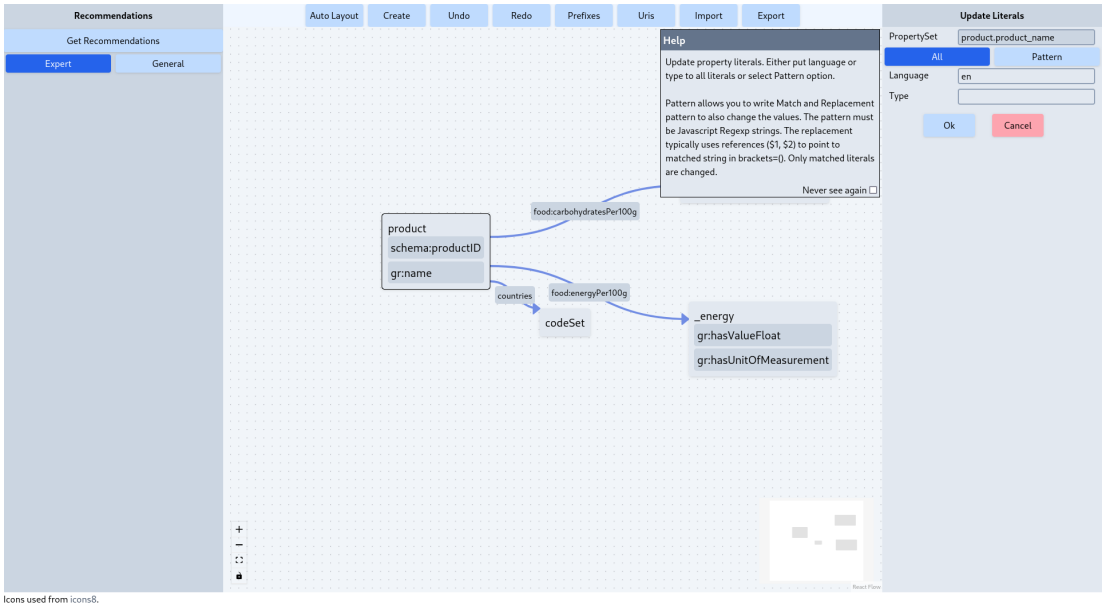

Figure 5.19 Set Language Tag for Product Names

Finally, we need to set the URI of the `countries` properties. We click on `product` and fill in `http://example.com/soldInCountries` to the URI input for the property set `countries` (Figure 5.20). We can scroll the pane to check that language tags, URIs and the URI for `countries` are set correctly.

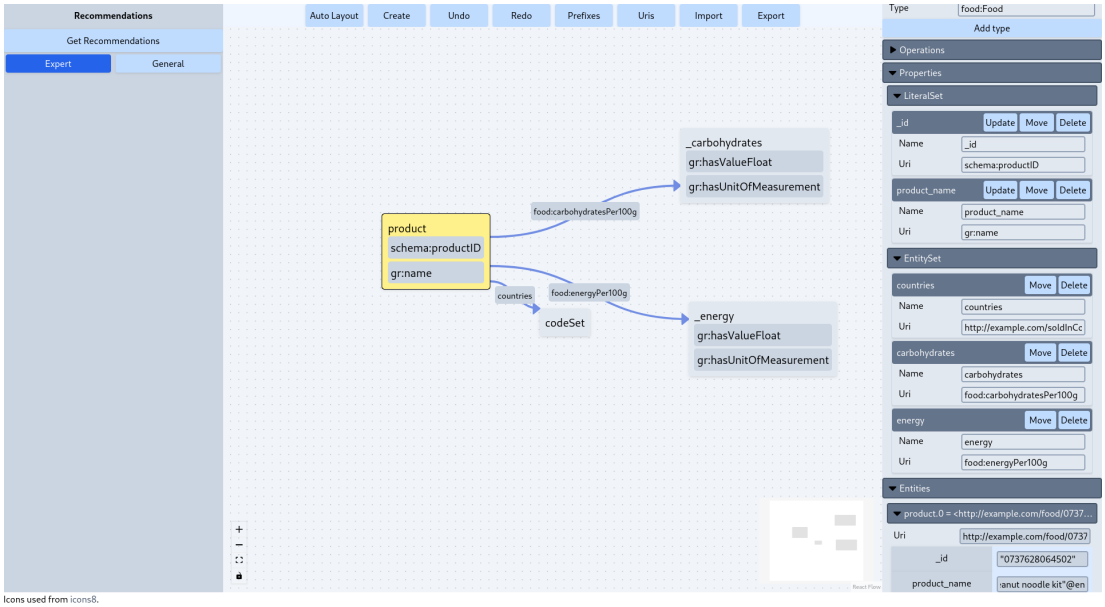

Figure 5.20 Set Countries Property

The only thing left to do now is to export the data to RDF. We click on `Export` and choose `Instances`. The `Export Instances` pane should appear (Figure 5.21). It validates all URIs and provides options to set URIs for all entity sets. Since the

URIs for `product` and `codeSet` are set directly on entities and the other entities should be blank nodes which is the default, we can click on `Ok`. A TTL file with the RDF is downloaded to our file system.

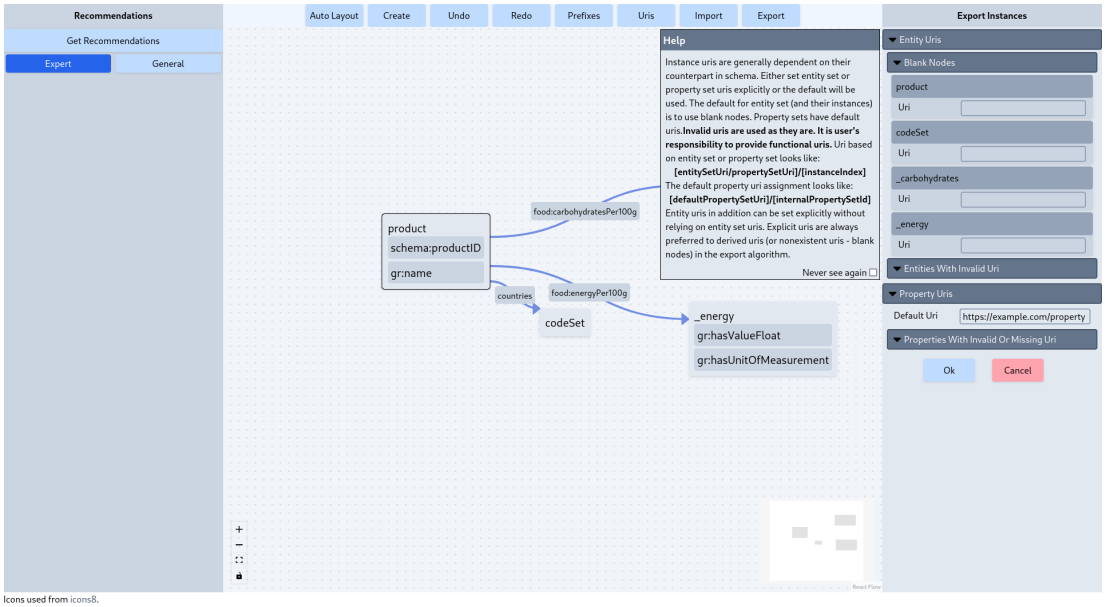

Figure 5.21 Export to RDF

We are at the end of the tutorial. Editor also supports `Undo` and `Redo` operations for convenience.

5.4 Final RDF Data

The produced RDF is shown below. We include a manually prettified version and only the first food product.

```
@prefix schema: <https://schema.org/> .
@prefix food: <http://purl.org/foodontology#> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.com/> .
@prefix ex-food: <http://example.com/food> .

ex-food:0737628064502 a food:Food ;
  schema:productID "0737628064502" ;
  gr:name "Thai peanut noodle kit"@en ;
  ex:soldInCountries
    <http://publications.europa.eu/resource/authority/country/USA> ;
  food:carbohydratesPer100g [
    a gr:QuantitativeValueFloat ;
    gr:hasValueFloat "71.15"^^xsd:double ;
    gr:hasUnitOfMeasurement "GRM"
  ];
  food:energyPer100g [
    a gr:QuantitativeValueFloat ;
    gr:hasValueFloat 385 ;
    gr:hasUnitOfMeasurement "K51"
  ].
```

6 Conclusion

The goal of the thesis was to propose a semiautomatic approach which combines search for vocabulary terms based on user data with the actual data transformation to help user represent structured data in RDF and implement its proof-of-concept implementation.

We first presented the issues encountered when representing structured data in RDF on a food product example from Open Food Facts [5]. We then analyzed the existing methods for recommendation and search of vocabulary terms. We also surveyed the existing software for vocabulary search and transformation of structured data to RDF. Based on the analysis, we identified the three challenges of the contemporary approaches and proposed our own approach to help users represent structured data in RDF. The approach is based on creating a model from the input structured that represents their RDF representation and user interactively transforming the model either manually or using recommendations to create the desired RDF representation. The recommendations suggest transformations of the model. They are primarily generated by recommenders with a built-in domain knowledge that recommending for specific domains. However, the approach also supports recommending based on general vocabulary term recommendation and search methods.

Afterwards, we identified the main components in the approach based on which we designed a system architecture. We also designed the model representation. We then implemented the system based on the approach and the design decisions made.

Lastly, we showed that the system can be successfully used to transform the food product data from example to RDF. However, the limitation of the approach is the number of supported domains which recommendation are created for. If a large part of user data is outside the supported domain, then the user must transform the data manually or rely on recommendations from the more general-purpose methods. However, that is also why the significant part of the design was focused on the addition of new expert recommenders being easy.

Bibliography

1. *The Linked Open Data Cloud* [https://lod-cloud.net/]. Accessed: 2024-03-29.
2. *Linked Data: Design Issues* [https://www.w3.org/DesignIssues/LinkedData.html]. Accessed: 2024-05-02.
3. *Linked Open Vocabularies* [https://lov.linkeddata.es/dataset/lov/]. Accessed: 2024-03-29.
4. *Bio Portal* [https://bioportal.bioontology.org/]. Accessed: 2024-03-29.
5. *Open Food Facts* [https://world.openfoodfacts.org/]. Accessed: 2024-04-01.
6. *SmartProducts Food Domain Model* [http://projects.kmi.open.ac.uk/smartproducts/ontology.html]. Accessed: 2024-04-01.
7. *Food Ontology in OWL* [http://www.w3.org/TR/2003/PR-owl-guide-20031215/food]. Accessed: 2024-04-01.
8. *Food Ontology* [http://data.lirmm.fr/ontologies/food]. Accessed: 2024-04-01.
9. *FoodOn: A farm to fork ontology* [https://foodon.org/]. Accessed: 2024-04-01.
10. *AGROVOC Multilingual Thesaurus* [https://agrovoc.fao.org/browse/agrovoc/en/]. Accessed: 2024-04-01.
11. *The FoodOntology Vocabulary for Food Products Description* [https://raw.githubusercontent.com/ailabitmo/food-ontology/master/food.owl]. Accessed: 2024-04-01.
12. *Good Relations: The Web Vocabulary for E-Commerce* [https://www.heppnetz.de/projects/goodrelations/]. Accessed: 2024-04-01.
13. *Schema.org* [https://schema.org/]. Accessed: 2024-04-01.
14. *UN/CEFACT Common Codes* [http://wiki.goodrelations-vocabulary.org/Documentation/UN/CEFACT_Common_Codes]. Accessed: 2024-04-01.
15. *Easygen* [https://github.com/go-easygen/easygen]. Accessed: 2024-04-02.
16. Schaible, Johann; Gottron, Thomas; Scherp, Ansgar. TermPicker: Enabling the reuse of vocabulary terms by exploiting data from the Linked Open Data cloud. In: *The Semantic Web. Latest Advances and New Domains: 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29–June 2, 2016, Proceedings 13*. Springer, 2016, pp. 101–117.
17. Liu, Tie-Yan et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*. 2009, vol. 3, no. 3, pp. 225–331.
18. Knoblock, Craig A; Szekely, Pedro; Ambite, José Luis; Goel, Aman; Gupta, Shubham; Lerman, Kristina; Muslea, Maria; Taheriyan, Mohsen; Mallick, Parag. Semi-automatically mapping structured sources into the semantic web. In: *Extended semantic web conference*. Springer, 2012, pp. 375–390.

19. Gupta, Shubham; Szekely, Pedro; Knoblock, Craig A; Goel, Aman; Taheriyan, Mohsen; Muslea, Maria. Karma: A system for mapping structured sources into the semantic web. In: *Extended Semantic Web Conference*. Springer, 2012, pp. 430–434.

20. Lafferty, John; McCallum, Andrew; Pereira, Fernando, et al. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: *Icml*. Williamstown, MA, 2001, vol. 1, p. 3. No. 2.

21. Kou, Lawrence; Markowsky, George; Berman, Leonard. A fast algorithm for Steiner trees. *Acta informatica*. 1981, vol. 15, pp. 141–145.

22. Ding, Li; Finin, Tim; Joshi, Anupam; Pan, Rong; Cost, R Scott; Peng, Yun; Reddivari, Pavan; Doshi, Vishal; Sachs, Joel. Swoogle: a search and metadata engine for the semantic web. In: *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 2004, pp. 652–659.

23. Brin, Sergey; Page, Lawrence. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*. 1998, vol. 30, no. 1-7, pp. 107–117.

24. Cheng, Gong; Qu, Yuzhong. Searching linked objects with falcons: Approach, implementation and evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*. 2009, vol. 5, no. 3, pp. 49–70.

25. *Apache Lucene* [`https://lucene.apache.org/`]. Accessed: 2024-04-03.

26. Kadilierakis, Giorgos; Fafalios, Pavlos; Papadakos, Panagiotis; Tzitzikas, Yannis. Keyword search over RDF using document-centric information retrieval systems. In: *The Semantic Web: 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31–June 4, 2020, Proceedings 17*. Springer, 2020, pp. 121–137.

27. *Elasticsearch* [`https://www.elastic.co/elasticsearch`]. Accessed: 2024-04-06.

28. Hasibi, Faegheh; Nikolaev, Fedor; Xiong, Chenyan; Balog, Krisztian; Bratsberg, Svein Erik; Kotov, Alexander; Callan, Jamie. DBpedia-entity v2: a test collection for entity search. In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2017, pp. 1265–1268.

29. Azad, Hiteshwar kumar; Deepak, Akshay; Azad, Amisha. LOD search engine: A semantic search over linked data. *Journal of Intelligent Information Systems*. 2022, vol. 59, no. 1, pp. 71–91.

30. Brill, Eric. Penn treebank tagger. *Copyright by MIT and University of Pennsylvania*. 1992.

31. Whetzel, Patricia L; Noy, Natalya F; Shah, Nigam H; Alexander, Paul R; Nyulas, Csongor; Tudorache, Tania; Musen, Mark A. BioPortal: enhanced functionality via new Web services from the National Center for Biomedical Ontology to access and use ontologies in software applications. *Nucleic acids research*. 2011, vol. 39, no. suppl_2, W541–W545.

32. Martínez-Romero, Marcos; Jonquet, Clement; O'connor, Martin J; Graybeal, John; Pazos, Alejandro; Musen, Mark A. NCBO Ontology Recommender 2.0: an enhanced approach for biomedical ontology recommendation. *Journal of biomedical semantics*. 2017, vol. 8, pp. 1–22.
33. *EMBL-EBI Ontology Lookup Service* [https://www.ebi.ac.uk/ols4/]. Accessed: 2024-04-09.
34. *Czech Open data portal* [https://data.gov.cz/]. Accessed: 2024-04-09.
35. *DCAT-AP-CZ* [https://ofn.gov.cz/rozhran%C3%AD-katalog%C5%AF-otev%C5%99en%C3%BDch-dat/2021-01-11/]. Accessed: 2024-04-10.
36. *European Data Portal* [https://data.europa.eu/en]. Accessed: 2024-04-10.
37. *Karma: A Data Integration Tool* [https://usc-isi-i2.github.io/karma/]. Accessed: 2024-03-30.
38. *Silk: The Linked Data Integration Framework* [http://silkframework.org/]. Accessed: 2024-03-30.
39. *Linked Pipes ETL* [https://etl.linkedpipes.com/]. Accessed: 2024-04-11.
40. *R2RML* [https://www.w3.org/TR/r2rml/]. Accessed: 2024-04-10.
41. *Direct Mapping* [https://www.w3.org/TR/rdb-direct-mapping/]. Accessed: 2024-04-11.
42. *Tarql: SPARQL for Tables* [https://github.com/tarql/tarql]. Accessed: 2024-04-11.
43. *CSV to RDF* [https://www.w3.org/TR/csv2rdf/]. Accessed: 2024-04-11.
44. *JSON-LD* [https://json-ld.org/]. Accessed: 2024-04-11.
45. *XSLT* [https://www.w3.org/TR/xslt-30/]. Accessed: 2024-04-11.
46. *DBpedia* [https://www.dbpedia.org/]. Accessed: 2024-04-13.
47. *The RDF Data Cube Vocabulary* [https://www.w3.org/TR/vocab-data-cube/]. Accessed: 2024-04-25.
48. *C4 model* [https://c4model.com/]. Accessed: 2024-04-19.
49. *The PROV Ontology* [https://www.w3.org/TR/prov-o/]. Accessed: 2024-04-25.
50. *Turborepo* [https://turbo.build/repo]. Accessed: 2024-04-25.
51. *Express* [https://expressjs.com/]. Accessed: 2024-04-25.
52. *Zod* [https://zod.dev/]. Accessed: 2024-04-25.
53. *Lodash* [https://lodash.com/]. Accessed: 2024-04-25.
54. *Formidable* [https://github.com/node-formidable/formidable]. Accessed: 2024-04-25.
55. *Axios* [https://axios-http.com/]. Accessed: 2024-04-25.
56. *Winston* [https://github.com/winstonjs/winston]. Accessed: 2024-04-25.
57. *N3* [https://github.com/rdfjs/N3.js]. Accessed: 2024-04-25.
58. *RDF.js* [http://rdf.js.org/]. Accessed: 2024-05-02.

59. *Communica Framework* [https://comunica.dev/]. Accessed: 2024-04-25.
60. *Jest* [https://jestjs.io/]. Accessed: 2024-04-25.
61. *Virtuoso* [https://github.com/openlink/virtuoso-opensource]. Accessed: 2024-04-25.
62. *SPARQL 1.1 Graph Store HTTP Protocol* [https://www.w3.org/TR/sparql11-http-rdf-update/]. Accessed: 2024-04-24.
63. *HTTP Proxy Middleware* [https://github.com/chimurai/http-proxy-middleware]. Accessed: 2024-04-24.
64. *Vite* [https://vitejs.dev/]. Accessed: 2024-04-25.
65. *Tailwind CSS* [https://tailwindcss.com/]. Accessed: 2024-04-25.
66. *Reactflow* [https://reactflow.dev/]. Accessed: 2024-04-25.
67. *Elkjs* [https://github.com/kieler/elkjs]. Accessed: 2024-04-25.
68. *Eclipse Layout Kernel* [https://eclipse.dev/elk/]. Accessed: 2024-04-25.
69. *React Error Boundary* [https://github.com/bvaughn/react-error-boundary]. Accessed: 2024-04-25.
70. *React Viewport List* [https://github.com/oleggrishechkin/react-viewport-list]. Accessed: 2024-04-25.
71. *Curl* [https://curl.se/]. Accessed: 2024-04-25.

List of Figures