FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Mahran Emeiri

A tool for configuring knowledge graph visual browser

Department of Software and Data Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Software and Data Engineering

Specialization: Software Engineering

Prague 2021

I declare that I carried out this master thesis independently, 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 Coll., 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 paragraph 1 of the Copyright Act.

In ..... date ..... Mahran Emeiri

This thesis is without a doubt the result of the hard work and support of many people that have been by my side through this journey.

It has been a tough two years and I could not have made it without my family who believed in me and kept supporting despite the long distances.

I would also like to express my gratitude to my supervisor Mgr. Martin Nečaský, Ph.D. for his guidance and patience over the last few months.

Title: A tool for configuring knowledge graph visual browser

Author: Mahran Emeiri

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D, Department of Software Engineering

Abstract: The main aim of this research is to provide a tool that helps users in creating, managing and validating configuration files visually, then compiles the user input into a valid RDF representation of the configuration that can be published as a linked open data resource, these configurations then can be used as an input to a Knowledge Graph Browser to be visualized as an interactive Knowledge Graph.

Keywords: knowledge graphs, visual browser, configuration, linked data

Contents

1. Introduction

The basic principle of the knowledge graph visual browser is enabling users to discover different knowledge graphs through different views defined by various browsing configurations. Real knowledge graphs are often too complex for human users, at the same time generic tools for knowledge graph visualisation and visual exploration are quite hard to use, and setting up the configuration for exploring these graphs could be very tedious and time consuming.

The basic idea was to allow knowledge graph experts to configure a set of simpler views, styles, data sets and vocabulary, so then it can be consumed by the knowledge graph visual browser, which will use these configuration to generate an interactive visualized knowledge graph.

This thesis presents the implementation of a tool called Knowledge Graph Browser Configuration Tool which helps to manage and create Knowledge Graph Browser configurations. This tool allows users to create new configuration files, store them locally on the user's device or remotely on the hosting server, and visualize some components.

At the moment of writing of the thesis, the majority of the stated tool features were developed but given the aim to primarily fulfill the needs of knowledge graph experts, there is a room for improvements and integrating new features as required.

1.1 Related work

This thesis combines several research topics in the current software engineering landscape including linked data visualization, knowledge graph management and knowledge Graph Visual Browser. The following sections provide an overview of recent work on these concepts.

1.1.1 Linked data visualization

As the Web becomes ever more enmeshed with our daily lives, there is a growing desire for direct access to raw data not currently available on the Web or bound up in hypertext documents. [1] Linked data provide the basis for knowledge to be distributed, networked, and shared. The term Linked Open Data (LOD) refers to a set of best practices for publishing and interlinking structured data on the Web. Creating a connection between data and its contexts could lead to the development of intelligent search engines which could explore the Web, moving from a keyword-based approach to a meaning-based approach. [2]

The term Linked Data was coined in 2006 from one of the creators of the Web, Sir Tim BernersLee. At the same time, he published a note[1] listing four rules for publishing LOD.

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF[2], SPARQL[3])
4. Include links to other URIs. so that they can discover more things.

Given the wide availability of LOD sources, it is crucial to provide intuitive tools enabling users without a technological background to explore, analyze, and interact with increasingly large data sets. LOD visualization aims to provide graphical representations of data sets with the aim to facilitate their analysis and the generation of insights out of complex interconnected information. [3]

[1] `https://www.w3.org/DesignIssues/LinkedData.html`
[2] `https://www.w3.org/RDF/`
[3] `https://www.w3.org/TR/sparql11-query/`

```
SELECT distinct ?cl_sub ?prop ?cl_obj
WHERE {
      ?cl_sub ?prop ?cl_obj.
      ?cl_sub rdf:type ?class.
      ?cl_obj rdf:type ?class.
}
```

Figure 1.1: Query for listing the Wikipathways dataset

As shown in (Figure 1.1) [4], visualization of complex LOD structures is a bit difficult. The authors [3] demonstrate this on Wikipathways [5]. Assuming that the user wants to know the contents of the data set, he/she could formulate a SPARQL query to extract the classes and relations and then analyze the results, as shown in (Figure 1.2).

sub	prep	obj
http://vocabularies.wikipathways.org/wp#Catalysis	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#TranscriptionTranslation	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#Inhibition	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#Binding	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#Stimulation	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#ComplexBinding	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#DirectedInteraction	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#Interaction
http://vocabularies.wikipathways.org/wp#GeneProduct	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DataNode
http://vocabularies.wikipathways.org/wp#Rna	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DataNode
http://vocabularies.wikipathways.org/wp#Metabolite	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DataNode
http://vocabularies.wikipathways.org/wp#Complex	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DataNode
http://vocabularies.wikipathways.org/wp#Protein	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DataNode
http://vocabularies.wikipathways.org/wp#Conversion	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#DirectedInteraction
http://vocabularies.wikipathways.org/wp#pathwayOntologyTag	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#ontologyTag
http://vocabularies.wikipathways.org/wp#diseaseOntologyTag	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#ontologyTag
http://vocabularies.wikipathways.org/wp#celltypeOntologyTag	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#ontologyTag
http://vocabularies.wikipathways.org/wp#target	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#participants
http://vocabularies.wikipathways.org/wp#source	http://www.w3.org/2000/01/rdf-schema#subClassOf	http://vocabularies.wikipathways.org/wp#participants

Figure 1.2: The results of the listing query

Adopting a graphical visualization, instead, can simplify a lot the analysis of the results. For example, the previous information can be obtained through a visualization tool (Figure 1.3). As it can be seen, displaying the same information as a graph, makes it easier to understand the connections and paths among the classes.

[4] http://www.linkeddatavisualization.com/resources/book_sample.pdf
[5] https://www.wikipathways.org/index.php/Portal:Semantic_Web

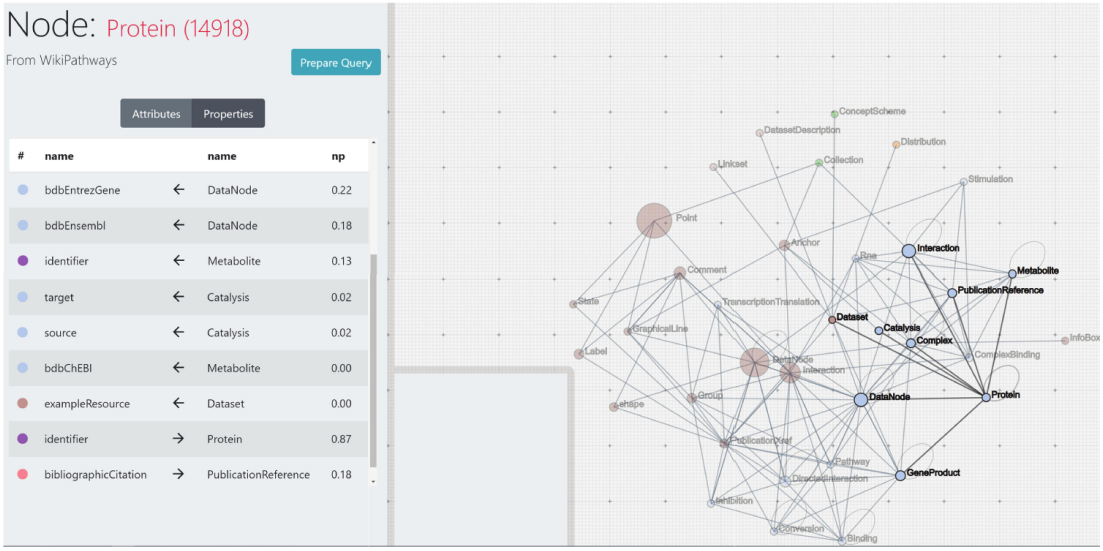

Figure 1.3: Visualizing the results with a visualization tool

1.1.2 Knowledge graph management

Knowledge graph is a visual representation of interlinked entities of a given domain , which captures the information about these entities and forges connections between them. That is why using a knowledge graph is a good method of representing entities that are published as LOD resources, and it might give an easier solution for tracing the connection between different entities.

The idea of browsing LOD has been taken up in several research works and resulted in the development of several browser applications for LOD such as the research prototypes Tabulator [4] and LENA [5].

Users can explore interlinked data with these browsers, however, most of these browsers do not support the users deciding which paths to follow during their exploration of the available data. Users are required to investigate displayed information manually and virtually scanning presented information from top to bottom, in order to spot links that are most relevant, meaningful, and interesting. [6]

In the recent years, knowledge graph technologies established a solid position in the enterprise world, serving as a central element in the organizational data management infrastructure. Knowledge graphs are becoming both the reposi-

tory for organization-wide master data (ontological schema and static reference knowledge) as well as the integration hub for various legacy data sources: e.g., relational databases or data streams. [7] The authors of metaphactory created a platform which covers the whole life cycle of knowledge graph applications from data extraction and integration, storage, and querying to visualization and data authoring.

1.1.3 Knowledge Graph Visual Browser

Knowledge graphs are a popular term to denote structured data represented in a form of graphs [8][9][11],They can be represented in different data models[11]. RDF [10]is a popular data model for sharing, publishing and integration of knowledge graphs.

Based on these facts an experimental tool called Knowledge Graph Visual Browser (KGBrowser)[6] was created by the department of Software and Data Engineering at Faculty of Math and Physics, it consumes a configuration which is expressed in RDF and published as Linked Open Data resource, and interprets visual configurations, provides a defined behaviour and enables users to visually explore knowledge graphs using the defined views.[7]

The configuration is a group of components expressed in a machine readable format and written in RDF, which is also an instance of an ontology that was defined for the Knowledge Graph Browser input.

1.2 Comparison with other tools

Two surveys of linked data visualization tools [12, 13] and a survey of linked data consumption tools [14] have been published. These surveys categorise the tools in the fields of visualizing LOD and LOD consumption into groups, the

[6]https://kgbrowser.opendata.cz
[7]https://dspace.cuni.cz/handle/20.500.11956/121022

first group is the generic graph-based knowledge graph visualization tools such as Ontodia[8], LodLive[9], RDF4U[10], RelFinder[11], RDFshape[12] and VisGraph[13]. The second group contains the tools which do not visualize knowledge graphs as nodes and edges but present nodes to users as interlinked web pages. There is also a group for the tools that provide an overview of a knowledge graph content in a form of statistics, classes used in the knowledge graph and navigation to individual instances.

Each on of these tools accepts a set of different configurations, some of them allow the users to customize these configurations, while others have specific set of configurations. in this section we will have a look at some of the generic graph-based knowledge graph visualization tools and their configurations and the level of customization they allow.

```tsx
/* template.tsx */

class CustomElementTemplate extends React.Component {
    render() {
        return (
            <div className='example-template'
                style={{borderColor: this.props.color}}>
                <div className={this.props.icon} />
                <div className='example-label'>{this.props.label}</div>
            </div>
        );
    }
}
```

```css
/* template.css */

.template-1-first-class {
  border-radius: 100px;
  background-color: white;
  border: 1px solid black;
  width: 200px;
  height: 200px;
}
```

Figure 1.4: Sample of style customization of ontodia

[8]https://github.com/metaphacts/ontodia
[9]http://en.lodlive.it/
[10]http://rathachai.github.io/rdf4u/
[11]http://www.visualdataweb.org/relfinder.php
[12]https://rdfshape.weso.es/
[13]https://visgraph3.github.io/

Starting with Ontodia, Ontodia is a JavaScript library that allows to visualize, navigate and explore data in the form of an interactive graph based on underlying data sources built using reactjs[14], this library allows users to change the visual style of any element. Not only color or icon can be changed, but also any style or template of any element [15]. These styles and templates are defined as reactjs components, it requires users to manually write the configurations (Figure 1.4).

Tools such as LodLive, RDF4U and VisGraph do not allow custom configurations, while LodLive gives some sort of interactivity when building the graph, there is no method for customizing the visual styles of the components. As for RDF4U, it gives the ability to show and hide some visual components, and there is no such support in VisGraph.

On the other hand, both RelFinder and RDFshape accept user configurations, while RelFinder uses XML[16] for its configurations , RDFshape accepts RDF configurations (Figure 1.5), but both of them do not allow a high level of customization.

```
<?xml version="1.0" encoding="utf-8" ?>
<data>
<proxy>
<url>http://www.visualdataweb.org/relfinder/proxy.php</url>
</proxy>
<endpoints>
<endpoint>
<name>DBpedia</name>
<abbreviation>dbp</abbreviation>
<description>Linked Data version of Wikipedia.</description>
<endpointURI>http://dbpedia.org</endpointURI>
<dontAppendSPARQL>true</dontAppendSPARQL>
<defaultGraphURI>http://dbpedia.org</defaultGraphURI>
<isVirtuoso>true</isVirtuoso>
<useProxy>false</useProxy>
<method>POST</method>
<autocompleteURIs>
<autocompleteURI>http://www.w3.org/2000/01/rdf-schema#label</autocompleteURI>
...
</autocompleteURIs>
<autocompleteLanguage>en</autocompleteLanguage>
<ignoredProperties>
<ignoredProperty>http://www.w3.org/1999/02/22-rdf-syntax-
ns#type</ignoredProperty>
...
</ignoredProperties>
<abstractURIs>
<abstractURI>http://dbpedia.org/property/abstract</abstractURI>
...
</abstractURIs>
<imageURIs>
<imageURI>http://dbpedia.org/ontology/thumbnail</imageURI>
...
</imageURIs>
<linkURIs>
<linkURI>http://purl.org/ontology/mo/wikipedia</linkURI>
...
</linkURIs>
<maxRelationLength>2</maxRelationLength>
</endpoint>
<endpoint>
...
</endpoint>
</endpoints>
```

RelFinder Configuration

```
@prefix :       <http://example.org/> .
@prefix schema: <http://schema.org/> .
@prefix item:   <http://data.europeana.eu/item/04802/> .
@prefix dbr:    <http://dbpedia.org/resource/> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix it:     <http://data.example.org/item/> .
@prefix wd:     <http://www.wikidata.org/entity/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .

:alice  a       foaf:Person .

:bob    a                   foaf:Person ;
        schema:birthDate    "1990-07-04"^^xsd:date ;
        foaf:knows          <http://example.org/alice#me> ;
        foaf:topic_interest  wd:Q12418 .

:carol  a                foaf:Person ;
        schema:birthDate  "unknown" .

wd:Q12418  dcterms:creator  dbr:Leonardo_da_Vinci ;
        dcterms:title    "Mona Lisa" .

it:243FA  dcterms:subject  wd:Q12418 ;
        dcterms:title    "La Joconde à Washington"@fr .
```

RDFShape Configuration

Figure 1.5: Sample configuration for both RelFinder and RDFshape

(Table 1.1) shows the comparison of KGBrowser with other Knowledge Graph

[14]https://reactjs.org/
[15]https://github.com/metaphacts/ontodia/wiki/Style-customization
[16]https://www.w3.org/XML/

browsing tools, form the point of the supported custom configurations where ○ denotes that this configuration is not supported, ● denotes that this configuration is supported and ◇ denotes that this configuration is partially supported.

	Ontodia	LodLive	RDF4U	VisGraph	RelFinder	RDFshape	KGbrowser
Visualization	●	●	●	●	●	●	●
Views	○	○	◇	○	○	○	●
Styles	●	○	○	○	○	◇	●
SPARQL endpoint	●	◇	○	◇	●	◇	●

Table 1.1: Comparison of supported configuration between different knowledge graph browsers

From the table, we can notice that all of the mentioned tools help visualizing different entities, but they vary in the level of customization. According to the developer documentation of Ontodia[17], this browser supports customizing both entities styles and the source of entities through a custom SPARQL endpoint, but there is no visible methods for customizing the view of an entity and the graph behaviour when interacting with an entity.

While on LodLive there is a partial support for a custom SPARQL endpoint as the tool gives the users a list of predefined SPARQL endpoint to select from, also there is a predefined set of interactions that can be used to interact with an entity, but there is no visible options for changing the styles of the entities.

RDF4U has a partial support for views as it contains a predefined set of features that can be turned off or on, and it also has a predefined set of entity behaviours, even thought there is no actual support of custom SPARQL endpoints, users can still form a json feed [18] and submit it to the tool to be visualized, but the users can not customize the visual representation of each entity.

In VisGraph users can not specify a SPARQL end point but they can specify an RDF file in one of two different ways, either by entering a URL or by uploading

[17]https://github.com/metaphacts/ontodia/wiki
[18]http://rathachai.github.io/rdf4u/data/lodac_bubo.json

it directly to the tool server, but these is no mention of a way for customizing the styles or the views and behaviour of an entity.

According to RelFinder documnetation[19], the tool can easily be configured to work with different RDF data sets, but we could not find a way to configure the views and behaviour of the graph or visual styles of the entities.

RDFshape allows a couple of methods for providing an input, either by providing a url or by entering an RDF text or by submitting a file of the configurations to the tool, and even though there is no visible methods for customizing the visual styles, the tool provides a set to predefined options for visualization.

Finally the KGBrowser, supports a user custom configurations represented in RDF format, it supports custom entity visual styles, custom views and behaviour and it accepts a custom SPARQL endpoint for each set of views.

1.3 Motivation

The main goal of this thesis is to provide a tool that helps users in creating, managing and validating configuration files visually, then compiles the user input into a valid RDF configuration that can be published as an LOD resource, and after that this customized configuration will be used as an input for the KGBrowser to be visualized as interactive Knowledge Graph.

The idea is to save time and efforts while creating these configurations, since a configuration file could contain hundreds of lines of RDF, with too many relations between individual components that could be easy to miss and hard to track manually, another motivation behind the tool is to reuse some of the published components.

[19]http://www.visualdataweb.org/relfinder.php

1.4 Thesis structure

To describe the development process that shows how the configuration tool was designed, built and tested over time, this thesis is split into the following chapters.

Chapter two: Defining requirements
In this chapter there is an overview of the initial requirements and the analysis and gathering processes, and the changes to those requirements after a few iterations.

Chapter three: System design
This chapter describes system design and incremental process of improvements, after collecting the requirements, and the design of the user interface based on the functionality.

Chapter four: Implementation overview
Here we go into the details of the implementation overview starting with the system architecture and its modules and the interaction between these modules.

Chapter five: Testing
In this chapter we highlight some usability testing scenarios and testing coverage for the source code.

Chapter six: Conclusions and future plans
In the final chapter, we shed the light on possible ways to improve this tool, with a conclusions drawn from the work done.

2. Defining requirements

The KGBrowser expects a configuration file in a specific format and structure, this configuration file may contain hundreds of lines and many interlinked sections, and that would increase the possibility of having errors and mistakes while creating this file manually.

The initial idea behind creating the tool is saving time and effort, plus providing a validation and visualization layer for some components while creating the configuration file. this tool should provide a template for creating this configuration file, automatically generate the interlinked sections, alert the users if they have any missing sections or incorrect linking between sections and provide the users with ability to preview their input.

2.1 Configuration file structure

Before going into the details of the requirements gathering, it is very important to understand the structure and the different sections the of configuration file, because it will have a huge impact on how the system was analysed, designed and built.

To support re-usability, an ontology was introduced based on a formal model which enables to express configurations in a machine readable form. It is called Knowledge Graph Visual Browser ontology. A configuration is expressed in RDF as an instance of the ontology (KGVB) .

The configuration and each one of its component are represented as RDF resources identified by their Internationalized Resource Identifiers (IRIs). This allows defining visual configurations by reusing components from other configurations. (Figure 2.1) shows the ontology as a Unified Modeling Language (UML) class diagram.

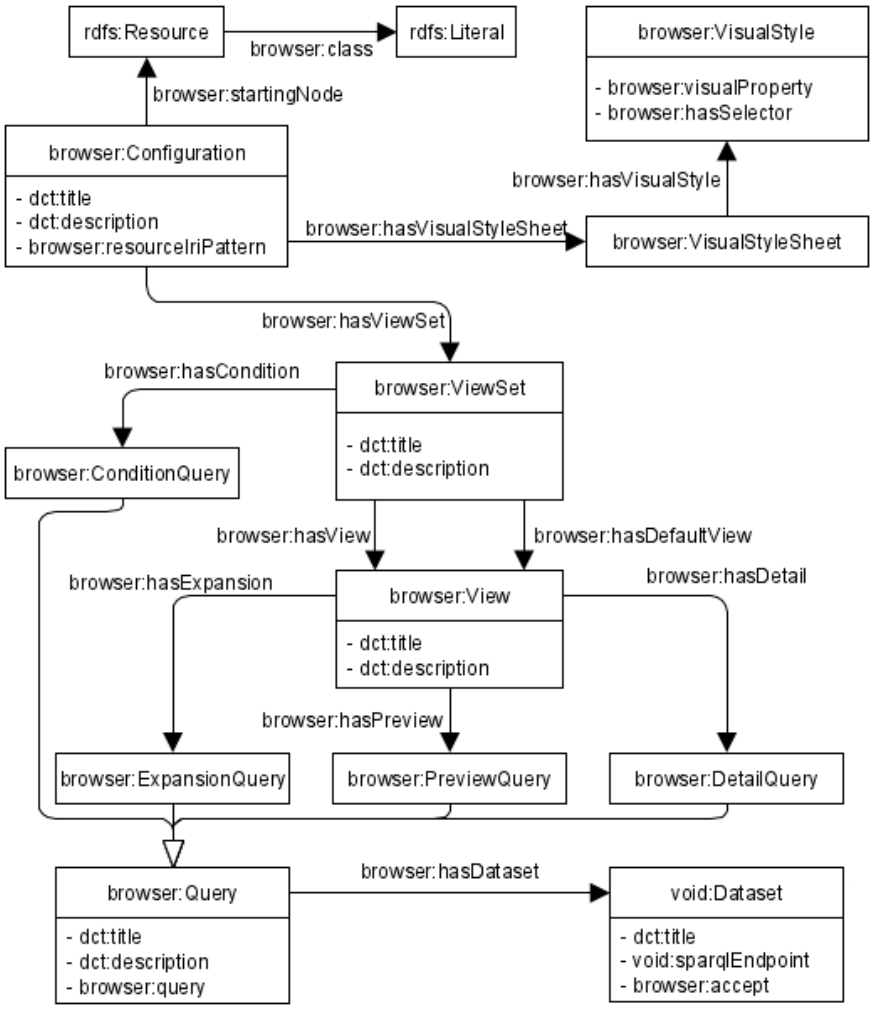

Figure 2.1: Ontology for defining visual configurations

We can break down the file into nine visible and interlinked components, each component contains a definitions of an RDF tuples. these components are listed as:

1. Configuration.
2. View sets.
3. Views.
4. Expansion queries.
5. Preview queries.
6. Details queries.

7. Data sets.

8. Visual style sheets.

9. Visual styles.

We can identify two more components in the file, they are used to provide some context to the configuration file, these components are:

1. Prefixes.

2. Vocabulary.

2.2 Analysing the configuration file

The configuration file consists of configuration components and each component is an RDF tuple which has some attributes.

Prefixes

This section of the configuration file includes the user defined prefixes which will be used in other configuration parts. Each prefix is annotated with "*@prefix*" [1] keyword then a prefix label followed by an IRI[2]. After analyzing a couple of files we noticed that there are some shared prefixes while other prefixes are domain specific. (Figure 2.2) Prefix section of configuration file.

```
@prefix browser: <https://linked.opendata.cz/ontology/knowledge-graph-browser/> .
@prefix wdab: <https://linked.opendata.cz/resource/vocabulary/knowledge-graph-browser/wikidata/animals/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix void: <http://rdfs.org/ns/void#> .
@prefix l: <https://slovník.gov.cz/legislativní/sbírka/90/1995/pojem/> .
@prefix a: <https://slovník.gov.cz/agendový/wikidata/animals/pojem/> .
```

Figure 2.2: Prefix section of configuration file

[1] https://www.w3.org/TR/turtle
[2] https://www.w3.org/TR/rdf11-concepts/

Configuration

A configuration is expressed as an instance of the *browser:Configuration*[3] class. it serves as the entry point to the configuration file and it has the following properties: (Figure 2.3) Sample of configuration section.

1. dct:title[4]: serves as human readable representation of the configuration.
2. dct:description: a human readable text that explains the purpose of the configuration.
3. browser:hasVisualStyleSheet: IRIs of the visual style sheets that are connected to this configuration.
4. browser:startingNode: IRIs of the actual entities that will serve as starting nodes of the interactive graph.
5. browser:resourceIriPattern: a regular expression that matches the IRIs of the graph entities.
6. browser:hasViewSet: IRIs of the view sets that are linked to this configuration.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/configuration/wikidata/animals> a browser:Configuration ;
  dct:title "Taxonomy of animals and plants"@en,
   "Taxonomie rostlin a živočichů"@cs;
  dct:description "Classification of all organisms, plants and animals, into taxa."@en,
   "Možnost procházet sítí taxonů všech zvířat a rostil jež jsou na Wikipedii."@cs;
  browser:hasVisualStyleSheet <https://linked.opendata.cz/resource/knowledge-graph-browser/wikidata/animals/style-sheet>;
  browser:startingNode <http://www.wikidata.org/entity/Q7377>,
   <http://www.wikidata.org/entity/Q192154>;
  browser:resourceIriPattern "^http://www\\.wikidata\\.org/entity/Q[1-9][0-9]*$";
  browser:hasViewSet <https://linked.opendata.cz/resource/knowledge-graph-browser/view-set/wikidata/animals/taxon> .
```

Figure 2.3: Sample of configuration section

View sets

View sets are defined as parts of visual configurations. each view set is an instance of the *browser:ViewSet* class and it has the following properties: (Figure 2.4) Sample of view set section.

[3]browser: is defined in the prefixes as https://linked.opendata.cz/ontology/knowledge-graph-browser/

[4]dct: is defined in the prefixed as http://purl.org/dc/terms/

1. dct:title: a human readable representation of the view set.
2. browser:hasView: IRIs of the visual views that are connected to this view set.
3. browser:hasDefaultView: IRI of the default view of this view set, it must be one of the IRIs defined in *browser:hasView*
4. browser:hasCondition: a condition query that will apply to all the views in this view set.
5. browser:hasDataset: IRIs of the data sets that are linked to this view set.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/view-set/wikidata/animals/taxon> a browser:ViewSet ;
  dct:title "Views of taxons"@en ;
  browser:hasView <https://linked.opendata.cz/resource/knowledge-graph-browser/view/wikidata/animals/taxon/broader>,
  <https://linked.opendata.cz/resource/knowledge-graph-browser/view/wikidata/animals/taxon/narrower> ;
  browser:hasDefaultView <https://linked.opendata.cz/resource/knowledge-graph-browser/view/wikidata/animals/taxon/broader> ;
  browser:hasCondition """PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
ASK {
  ?node wdt:P31 ?type .
  FILTER(?type IN (wd:Q16521, wd:Q713623))
}""" ;
  browser:hasDataset <https://linked.opendata.cz/resource/knowledge-graph-browser/dataset/wikidata> .
```

Figure 2.4: Sample of view set section

Views

Views are visual representations of the graph nodes. each view is an instance of the *browser:View* class and it has the following properties: (Figure 2.5) Sample of view section.

1. dct:title: a human readable representation of the view.
2. browser:hasExpansion: An IRI of an expansion query, which will define how to navigate to the neighbors of the current node.
3. browser:hasPreview: An IRI of a preview query, which fetches the properties and the visual classes of a certain node.
4. browser:hasDetail: An IRI of a detail query, which fetches the properties of a certain node.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/view/wikidata/animals/taxon/broader> a browser:View ;
  dct:title "Parent taxons"@en ;
  browser:hasExpansion <https://linked.opendata.cz/resource/knowledge-graph-browser/expansion-query/animals/taxon/broader> ;
  browser:hasPreview <https://linked.opendata.cz/resource/knowledge-graph-browser/preview-query/animals/taxon/basic> ;
  browser:hasDetail <https://linked.opendata.cz/resource/knowledge-graph-browser/detail-query/animals/taxon/basic> .
```

Figure 2.5: Sample of view section

Queries

There are three types of queries which are linked to a view, these queries are:

1. Expansion queries as an instance of *browser:ExpansionQuery* class: defines how to navigate to the neighbors of the current node.
2. Preview queries as an instance of *browser:PreviewQuery* class: fetches properties and visual classes of a certain node.
3. Detail queries as an instance of *browser:DetailQuery* class: fetches properties of a certain node

They share the same properties but the differ in the execution behaviour, each section contains the following properties: (Figure 2.6) Sample of detail query section.

1. dct:title: a human readable representation of the query.
2. browser:hasDataset: An IRI of the data set which the query will run against.
3. browser:query: a SPARQL query that defines the results of the query.

Data sets

This defines a SPARQL endpoint on which queries are executed. each data set is an instance of *browser:Dataset* class, it has the following properties: (Figure 2.7) Sample of data set section.

1. dct:title: a human readable representation of the data set.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/detail-query/animals/taxon/basic> a browser:DetailQuery ;
  dct:title "Taxon basic detail" ;
  browser:hasDataset <https://linked.opendata.cz/resource/knowledge-graph-browser/dataset/wikidata> ;
  browser:query """PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wdab: <https://linked.opendata.cz/resource/vocabulary/knowledge-graph-browser/wikidata/animals/>
PREFIX browser: <https://linked.opendata.cz/ontology/knowledge-graph-browser/>
CONSTRUCT {
  ?node rdfs:label ?nodeLabel ;
    wdt:P225 ?p225 ;
    wdt:P181 ?p181 ;
    wdt:P18 ?p18 .
} WHERE {
  ?node wdt:P31 wd:Q16521 .

  {
    ?node wdt:P225 ?p225 .
  } UNION {
    ?node wdt:P181 ?p181 .
  } UNION {
    ?node wdt:P18 ?p18 .
  }

  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }

}""" .
```

Figure 2.6: Sample of detail query section

2. void:sparqlEndpoint[5]: The IRI of the actual SPARQL endpoint that will execute the queries.

3. accept:query: defines the content type that will be used to communicate with the endpoint.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/dataset/wikidata> a void:Dataset ;
  dct:title "Wikidata SPARQL endpoint" ;
  void:sparqlEndpoint <https://query.wikidata.org/sparql> ;
  browser:accept "application/sparql-results+json" .
```

Figure 2.7: Sample of data set section

Vocabularies

This defines the vocabulary that are used in the view, each vocabulary can be an instance of one of the following classes *owl:Class*, *owl:ObjectProperty*, *owl:DataProperty* [6]. (Figure 2.8) Sample of vocabularies section.

[5]void: is defined in the prefixed as http://rdfs.org/ns/void#
[6]owl: is defined in the prefixed as http://www.w3.org/2002/07/owl#

```
<https://linked.opendata.cz/resource/vocabulary/knowledge-graph-browser/wikidata/animals/taxon> a owl:Class ;
  rdfs:label "taxon"@en .

<https://linked.opendata.cz/resource/vocabulary/knowledge-graph-browser/wikidata/animals/broader> a owl:ObjectProperty ;
  rdfs:label "parent taxon"@en ;
  rdfs:domain wdab:taxon ;
  rdfs:range wdab:taxon .
```

Figure 2.8: Sample of vocabularies section

Style sheets

This defines a group of visual styles that will be used in the configuration, each style sheet is an instance of *browser:VisualStyleSheet* class and it has one property *browser:hasVisualStyle* which is the IRIs of the visual styles of this configuration. (Figure 2.9) Sample of style sheet section.

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/wikidata/animals/style-sheet> a browser:VisualStyleSheet ;
  browser:hasVisualStyle <https://linked.opendata.cz/resource/knowledge-graph-browser/wikidata/animals/style/taxon>,
    <https://linked.opendata.cz/resource/knowledge-graph-browser/wikidata/animals/style/species>.
```

Figure 2.9: Sample of style sheet section

Styles

This defines a the visual properties of a given entity, each style is an instance of *browser:VisualStyle* class and it has wide range of proprieties, some proprieties are shared among all of the styles e.g. *browser:hasSelector* which is the Cascading Style Sheets (CSS) selector[7] of the graph nodes on which this style will be applied, other properties are dynamically added with different values, such as width, color and other user interface (UI) properties that are defined by the cytoscapejs[8], since the KGBrowser is using the cytoscapejs library for visualization. (Figure 2.10) Sample of style section.

[7] https://www.w3.org/Style/CSS/Overview.en.html
[8] https://js.cytoscape.org/#style/property-types

```
<https://linked.opendata.cz/resource/knowledge-graph-browser/wikidata/animals/style/taxon> a browser:VisualStyle ;
  browser:border-width "2px" ;
  browser:shape "octagon" ;
  browser:label "data(label)" ;
  browser:width "20px" ;
  browser:height "20px" ;
  browser:padding "2px" ;
  browser:text-halign "center" ;
  browser:text-valign "bottom" ;
  browser:text-margin-y "2px" ;
  browser:hasSelector ".taxon" .

<https://linked.opendata.cz/resource/knowledge-graph-browser/wikidata/animals/style/species> a browser:VisualStyle ;
  browser:background-color "#ff8000" ;
  browser:hasSelector ".species" .
```

Figure 2.10: Sample of style section

2.3 Requirements gathering

The requirements gathering process was done in multiple steps using agile methods [15], started with the initial requirements and the basic functionality needed from the tool then incrementally adding more complex and advanced features in each meeting with the stakeholder. This process can be divided into three major milestones.

Each one of these milestones added a new set of requirements, and of course each one had its impact on the software behaviour and the system design and architecture, these milestones are:

1. The initial requirements.
2. Data persistence.
3. User experience.

During the these iterations we identified two types of users who might be interested in using the tool based on their technical experience:

1. **Normal user:** users with no technical background but still interested in using the KGBrowser, or users who just need to experiment with existing configuration or do some adjustments on a predefined configuration file.

2. **Expert user:** users with good knowledge of how LOD works and how KGBrowser is configured, and they are able to use the tool to either modify existing an configuration file or create new configurations from scratch.

2.4 The initial requirements

After the first look on a couple of samples of the KGBrowser configuration files, we agreed on a couple of key requirements and functionalities that should be available in the tool, starting from the idea that it should user friendly and usable by none expert users, but at the same time it should provide some advanced features that can be used by expert users. and that lead us to defining the following requirements and use cases (Figure 2.11).

1. The user should be able to create a new configuration.
2. The user should be able to download the generated representation of configuration file.
3. The user should be able to modify configuration files.

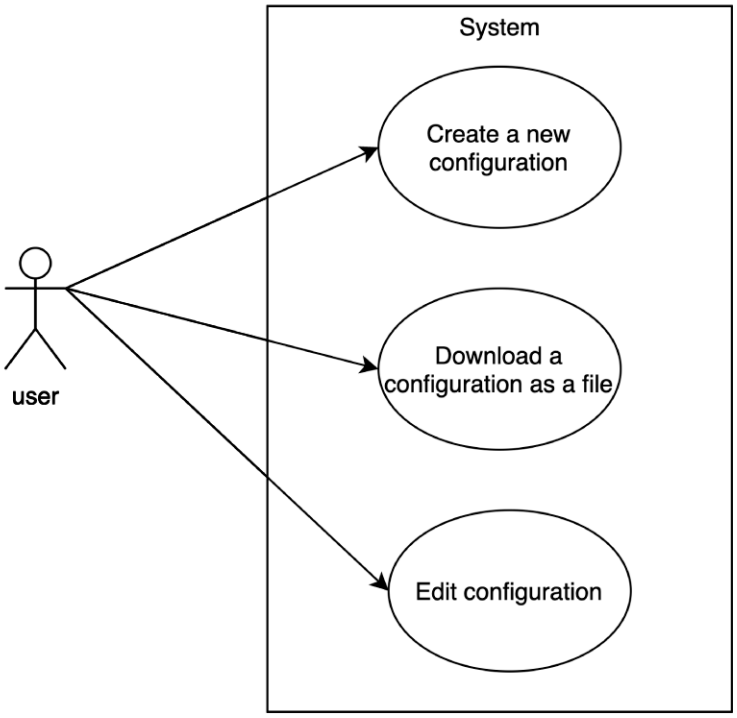

Figure 2.11: Use case diagram of the initial requirements

Some use case scenarios

Use case	Download a configuration as a file
Summary	A .ttl file should be generated by the system and saved to the user's device upon the user request
Normal flow	• The user starts a new blank configuration. • The user clicks on the download button. • The system validates the user input for missing components • The system generates an RDF turtle representation of the configuration • The system saves the generated configuration as file to the user's device
Pre-condition	The user must fill in the configuration components.
Exceptions	The system alerts the user for missing components.

Table 2.1: A use case scenario for downloading an RDF representation of the configuration.

2.5 Data persistence

In the second iteration we noticed that the configuration file takes too much time to be generated, and might be difficult to finalize the whole file at once, so based on that a new set of requirements was added (Figure 2.12).

1. The user should be able to save the configuration files remotely.

2. The user should be able to continue working on a configuration file at any specific point.

3. The user should be able to see a list of available configuration files.

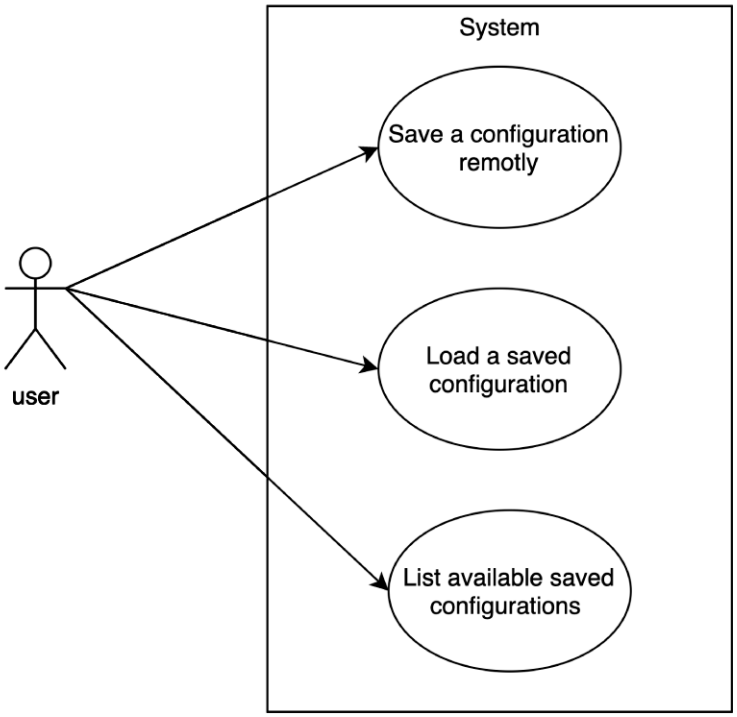

Figure 2.12: Use case diagram of the second iteration

Some use case scenarios

Use case	Load and edit a saved configuration
Summary	A user shall be able to continue working on a saved configuration
Normal flow	• The system loads a list of saved configurations • The user clicks on the selected configuration. • The system fetches the configuration from the database • The system populates the configuration component into the user interface
Pre-condition	The system must have saved configurations
Exceptions	The system could not load the selected configuration.

Table 2.2: A use case scenario for loading a saved configuration

2.6 User experience

In the third iteration we focused on improving the users experience while using the tool, in a way that can give them an estimate on how the KGBrowser will behave when using the written configuration file, another improvement was introduced, which is to give the users the ability to reuse or alter some parts of a configuration file that was published as an LOD resource, another thing was noticed from the analysis that most components are using the same base URI so it would make more sense if we define this URI once, and taking this in consideration we defined the following requirements (Figure 2.13).

1. The user should be able to load configurations from an LOD resource.
2. The user should be able to choose which sections to include from the loaded configuration.
3. The user should be able to run and preview the queries.
4. The user should be able to define the base uri for components.
5. The user should be able to use meta configuration as templates.
6. The user should be able to attach components from meta configuration.
7. The user should be able to assign the current configuration to a meta configuration group.

Some use case scenarios

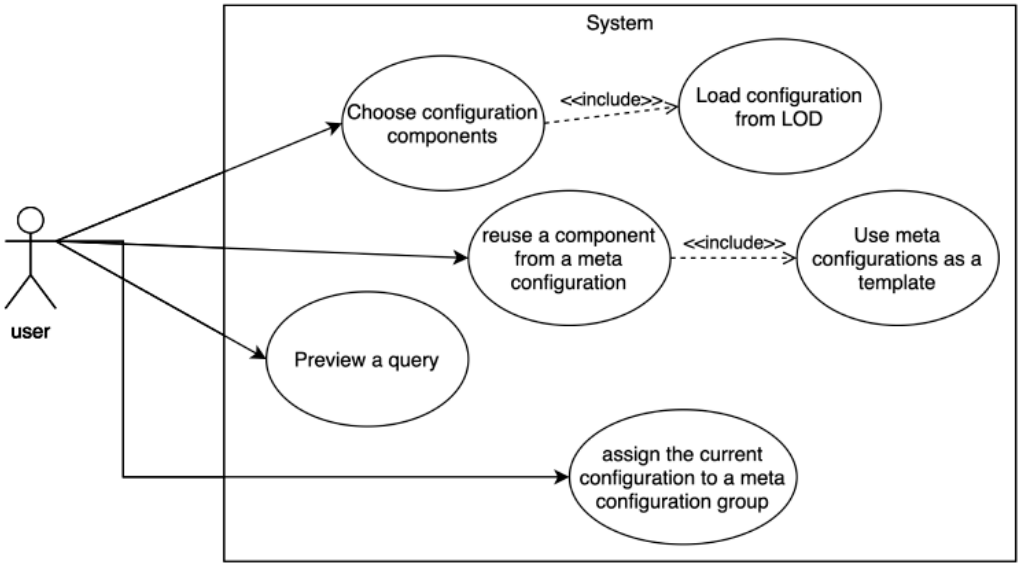

Figure 2.13: Use case diagram of the third iteration

Use case	Reuse components from a meta configuration
Summary	A user shall be able reuse some components from meta configuration group.
Normal flow	• The user starts a new blank configuration. • The user starts filling the configuration components. • The system suggests some components to reuse. • The user selects the needed component. • The system populate the selected component into the user interface.
Pre-condition	The user must use a meta configuration as a template.
Exceptions	The system could not load the selected meta configuration.

Table 2.3: A use case scenario for reusing components

Use case	Preview a query
Summary	A user shall be able see the visualization of a query results
Normal flow	• The user selects a query. • The user clicks the preview button. • The user selects a data set and a starting node. • The system executes the query and visualize the results.
Pre-condition	The user must fill in a query and data set information.
Exceptions	The query results are empty.

Table 2.4: A use case scenario for previewing a query

3. System analysis and architecture

After gathering the initial requirements and the analysis of the configuration file and its components, we started with a proof of concept of the tool, taking in consideration that the tool should be scalable and easily extendable since we are following an incremental development process.

3.1 The system design

The system architecture went through a couple of changes during the process due to the added requirements in each iteration, it was important from the beginning to keep that in mind so we can add to the system rather than redesigning it with every iteration.

At the early stages and after the gathering the initial requirements of the system and after a careful analysis we decided to use a web based application for the solution software, we also decided to encapsulate our software modules with docker[1], because basically docker containers encapsulate everything an application needs to run, they allow applications to be shuttled easily between environments, whether it's a development or a production environment.

The next step was choosing the front-end framework which will be used for creating the front-end application, and since we are using a web based application we had a variety of modern frameworks to choose from, we chose to use vuejs [2] as a framework for the front-end module of the system, also we integrated this framework with a couple of helper libraries and plugins such as vuex[3] for man-

[1]https://www.docker.com/
[2]https://vuejs.org/
[3]https://vuex.vuejs.org/

aging the app state, vue-router [4] for managing the url changes and vuetify[5] that provides reusable UI components. Based on those decisions we created the first Proof of concept (POC) of the application.

After the second iteration of the requirements, and based on the need of storing the configurations files remotely at any point during the creation process, so the users would be able to modify them later on, we needed to extend the system by adding two modules, the first module is a database service, and the second module will handle the communication between the front-end module and the database module and it would be called the back-end module.

We deiced to use nestjs [6] for the back-end module, which is an open-source Node.js [7] framework for creating and compelling back-end systems and follows the restful web service [8] principles .

As for database module we chose to use Mysql [9] database for storing the configuration files status, the database design is very straightforward, we have one table which contains the following columns:

1. name: which is human readable name of the configuration file, if not provided, the system will generate a name based on the current timestamp.
2. baseLink: the base URI for the configuration file components, if defined by the user.
3. metaGroup: the meta group that was chosen by the user (if needed) to assign this configuration file to.
4. body: the current state of the configuration file serialized as a string.
5. created_at: A time stamp that is inserted when file first stored in the database.

[4]https://router.vuejs.org/
[5]https://vuetifyjs.com/
[6]https://nestjs.com/
[7]https://nodejs.org/
[8]https://www.w3schools.in/restful-web-services/intro/
[9]https://www.mysql.com/

6. updated_at: A time stamp that is inserted when file last edited.

We had to extend the system one more time after third patch of requirements, as we needed the software solution to execute queries against a given data set or load a configuration file represented as an LOD resource using an HTTP call, and hence this feature was already developed inside the KGBrowser as separate module called kgserver[10], we deiced to get an instance of the kgserver and encapsulate it within a docker container, but since the endpoints that are provided by kgserver were expecting a certain input structure, we needed to modify these endpoints to serve our needs.

Another obstacle we faced is how connect the front-end module to the kgserver, we could configure the front-end app to communicate directly with the kgserver, but that would mean that the front-end will be aware of that module, plus it is already communicating with the back-end module, so the solution was to configure the back-end module to serve as a proxy for the kgserver, so it will listing to the front-end module then upon request it will forward this request to the kgserver, once it gets a response from the kgserver it will reply to the client with the response.

In this case the front-end module will not be aware of the existence of the kgserver, it will communicate with it thought the back-end module, which acts as abstraction layer, this architecture is shown in the Figure 3.1.

[10]https://github.com/martinnec/knowledge-graph-browser

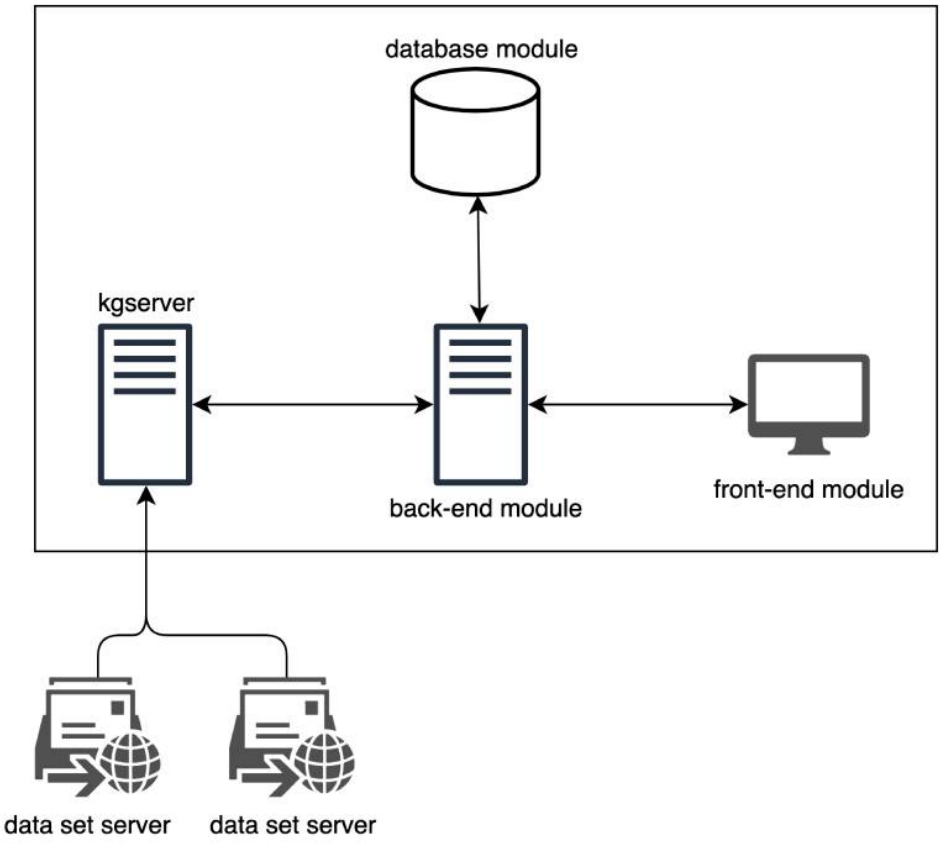

Figure 3.1: System modules and interactions with third party services

3.2 UI design

Based on the provided requirements, the UI was designed based on the following functionality categories:

1. Managing configuration files
2. Validating the user input
3. Visualizing the queries
4. Fetching configuration from LOD
5. Using meta configurations

3.2.1 Managing configuration files

The main functionality of the software is to create and manage configuration files for KGBrowser, in this case the application will provide a template for generating such a configuration. According to the analysis of a multiple configuration files, we noticed a hierarchical patterns between the the components of each configuration file, starting with the configuration components to view sets and style sheets, so we decided to use a top-bottom generating module, which mean that the user input in each section will impact how to generate the next level sections. As an example, each view sets should have a couple of views, so whenever the user assign these views to the view set, the software will automatically generate a placeholder in the views section.

However, there was an exception to this formula, which is the vocabulary section, after analysing a couple of files, we could not find a direct link between this section and another component, that we can used to automatically generate this part, so the ideal solution was to keep this part available for the user as a free input, yet that might not be entirely true, we prepared some predefined templates that represent three main vocabulary classes, *class*, *objectProprty* and *dataProperty*, then we left the freedom of using these templates to the user according to their needs, keeping in mind that the application will notify users if they try to export the configuration file without adding any vocabulary.

One of the interesting sections in the configuration file is the visual style component, all of the other components are subjected to a fixed template, let us take the view component as an example, the view component has four predefined properties, a title, an expansion query, a preview query and finally a detail query, on the other hand the style component can have a dynamic number of properties based on the user input, and each style component represents the visual styles of an entity or a group of entities (nodes or edges) in the knowledge graph.

Since the KGBrowser depends heavily on the *Cytoscapejs* library to visualize the graph, it was very important to analyse the visual proprieties that are accepted

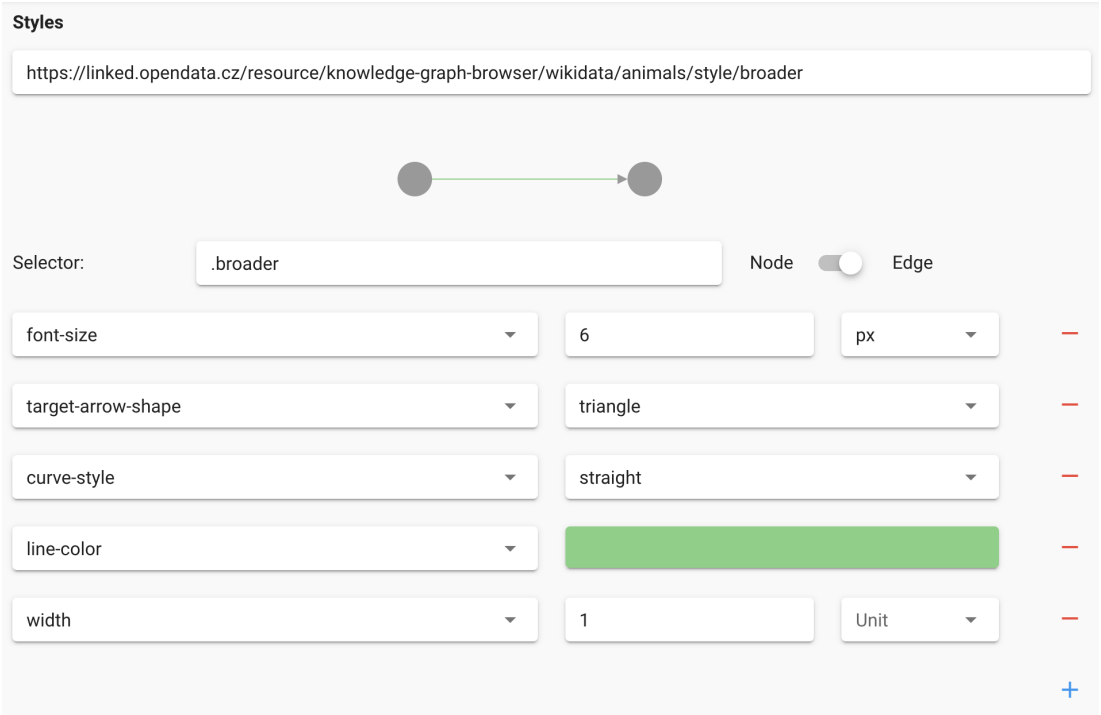

Figure 3.2: How the system is generating the style component

by this library. Basically the library accepts two types style groups, nodes style group and edge style group, each one of these group has a set of visual styles attribute that can apply to respective graph entity. The software will provide the users with a list of these attribute, based on their choice of entity type, in addition to that, the application will guide the users regarding the expected input type for each attribute. (Figure 3.2) How the system is generating the style component

Beside supporting the functionality of generating a configuration file, the software includes some extra enhancements that improve the user experience while creating the configuration file, one of these functionalities is the ability to save the file remotely at any given point of time so the user can resume the work on the file at any time, beside that, the user should be able to generate a turtle[11] preview of the configuration file and download it to the local machine (Figure 3.3).

.

Another thing worth mentioning here is the usage of base URI [12], we noticed that

[11]https://www.w3.org/TR/turtle/
[12]https://www.w3.org/TR/turtle/#relative-iri

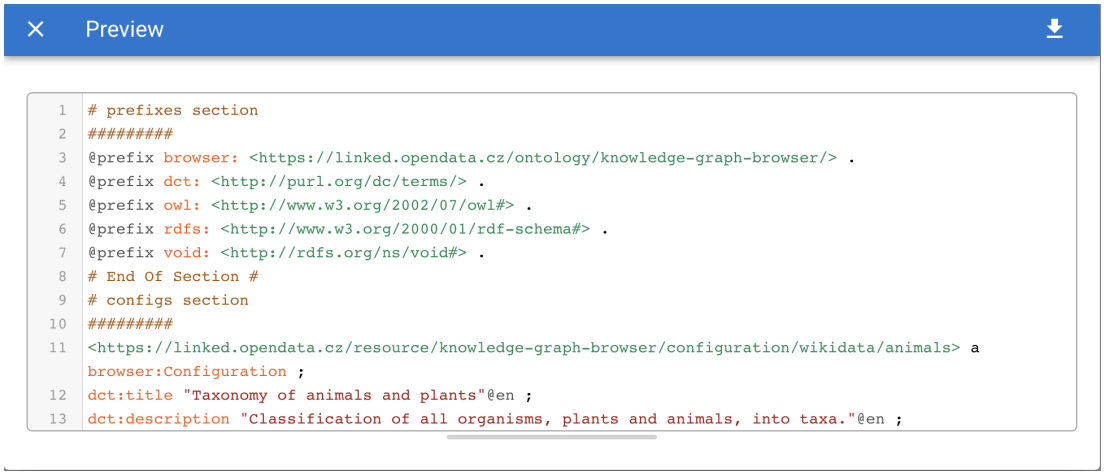

Figure 3.3: Turtle of the generated configuration

most of the componets share the same base URI so it would make sense that the users can define this base URI and then reuse it again, keeping the ability to use absolute URIs when needed. so the UI has an input field where the user can enter this shared base URI (Figure 3.4).

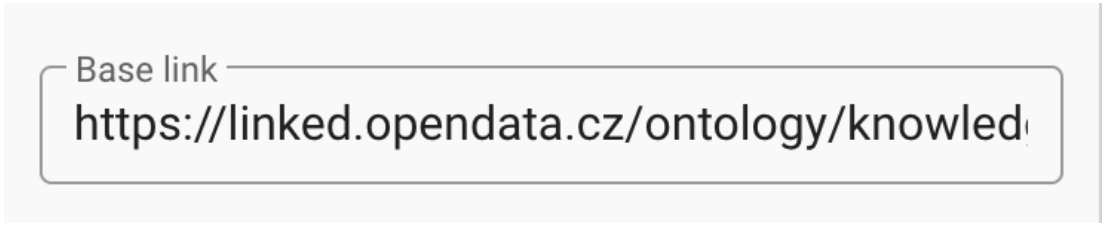

Figure 3.4: Defining a base uri

After defining the base URI, the users can use relative URIs based on RDF standards to achieve the correct behaviour (Figure 3.4), this will be reflected in the generated file as "*@base <base_uri>* ." and the relative uri will be generated as "*<#relative_uri> a class ;*".

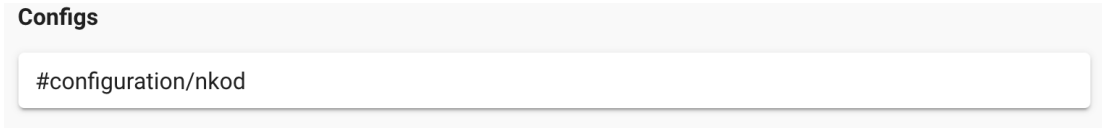

Figure 3.5: Using relative uri

3.2.2 Validating the user input

The configuration file may contain hundreds of lines, and that would make it very difficult to trace manually any missing sections or invalid values, thus we introduced the user input validation functionality.

The application is validating the user input on multiple levels, starting with the prefixes section, first we need to highlight that the application is providing the users with predefined prefixes, that were inferred from analysing a couple of configuration files, then the application is applying two types of validation, the first type of validation is that the user can not define a prefix with the same name of an already defined prefix (Figure 3.6), the second type of validation is that the prefix should have a valid IRI.

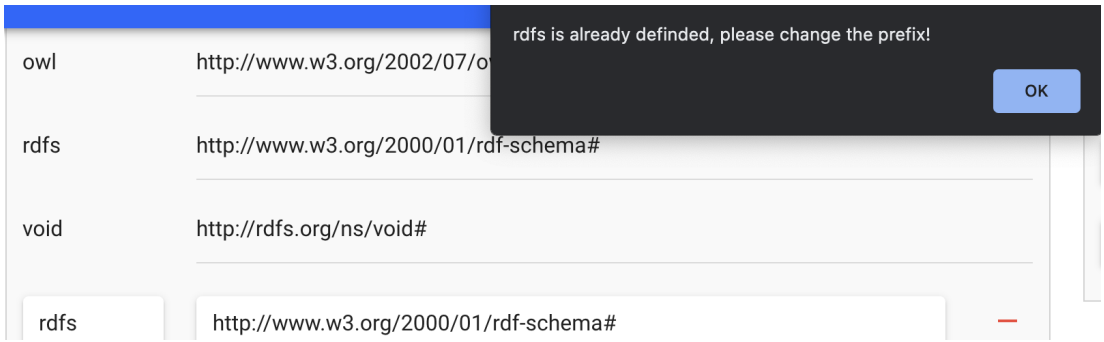

Figure 3.6: Validating duplicate prefix

Another level of validation is the validation of user input for each one of proprieties in the configuration file components, after analysing the configuration components we noticed that the proprieties could either a text literal or an IRI, but from UI point of view, that was a bit different, we were able to identify four types of user input, a text input, an IRI input, a drop down where the value of a property depends on another property, a SRARQL query and for that we used a SPARQL editor called yasgui[13]. the software is applying a couple of validation steps on the user input, such as an IRI input must have a valid structure (Figure 3.7), Text input must have a the language indicator if has multiple values, or any property can not have duplicate values of an input (Figure 3.7).

[13]https://triply.cc/docs/yasgui-api

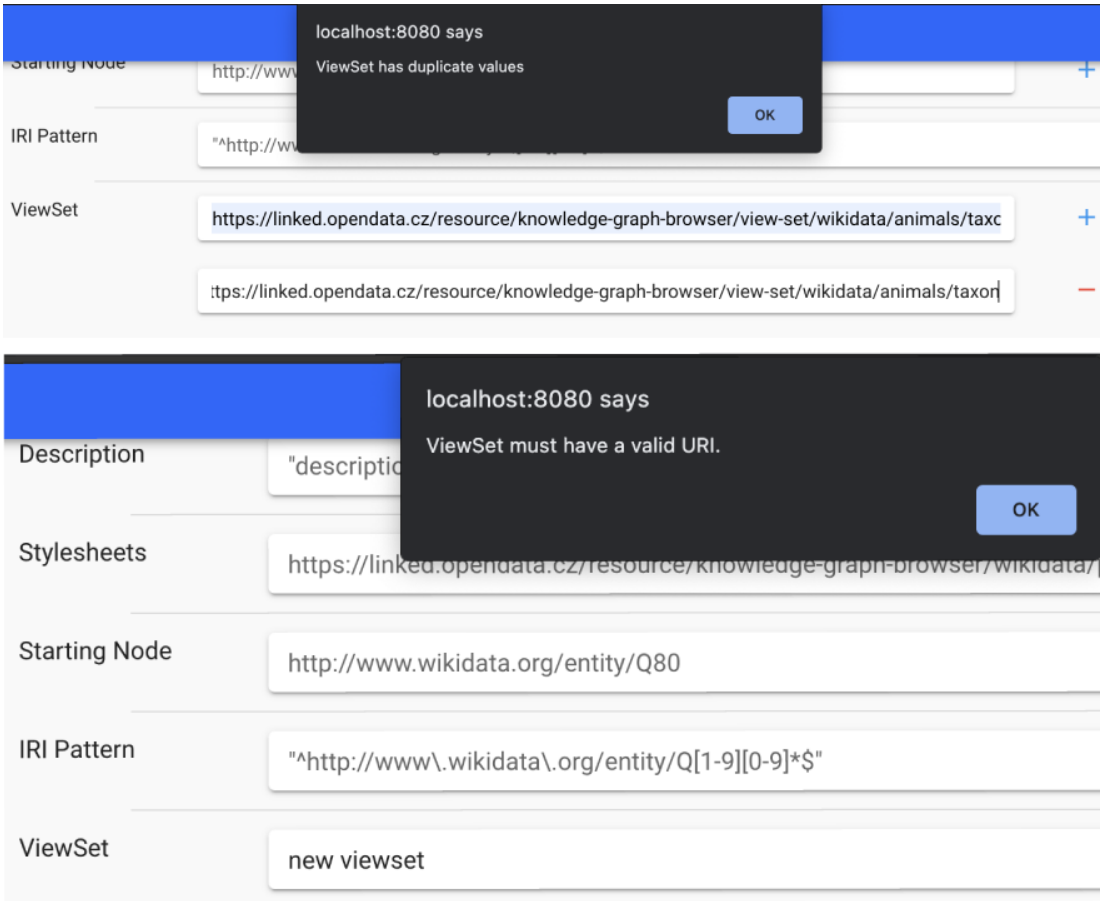

Figure 3.7: Validating duplicate value of an input and invalid IRI structure

3.2.3 Visualizing the queries

The queries are actually what give the KGBrowser its interactive properties, they define the graph behaviour as a response to a specific action, writing these queries might be a bit complicated especially for none expert users, it had been noted by a previous study [14] that these queries usually have a standard code structure, taking this point into consideration, we decided to include the sample structure as a placeholder for each query with instructions how to fill each section (Figure 3.8), which might be handy for both expert and none expert users .

There are three types of queries, expansion, preview and details queries, which can by split into two groups, queries that will fetch the properties of an entity and that include preview and detail queries, and queries that fetch the information about the neighbours of an entity and relations between them, and that includes

[14]https://dspace.cuni.cz/handle/20.500.11956/121022

```
#PREFIX pfx: <http://sample.uri/>
 ################# END OF PREFIXES #################
 CONSTRUCT {
   ##### CONSTRUCT OF EXPANTION NODES #####
   # ?expansionNode a <uri> ;
   #   rdfs:label ?expansionNodeLabel ;
   #   visualStyle:class ?expansionNodeClass .

   ##### CONSTRUCT OF EXPANTION EDGES #####
   # ?node has:edgeTo ?expansionNode .

   ##### CONSTRUCT VISUAL STYLES OF EDGES #####
   # has:edgeTo visualStyle:class "className" .

 ################# END OF CONSTRUCT #################
 } WHERE {

   ##### PATTERS FOR EXTRACTING THE NODES ####

 }
```

Figure 3.8: A sample of an expansion query template

the expansion query.

The deference between preview and detail query is that a preview query will get a summary of the entity information and its visual attributes, while the detail query will get the properties of an entity such as label, description and image if available, the software solution gives the users the ability to run the queries that they write against a data set they choose, with one of the configuration starting nodes considered as seed for query (Figure 3.9).

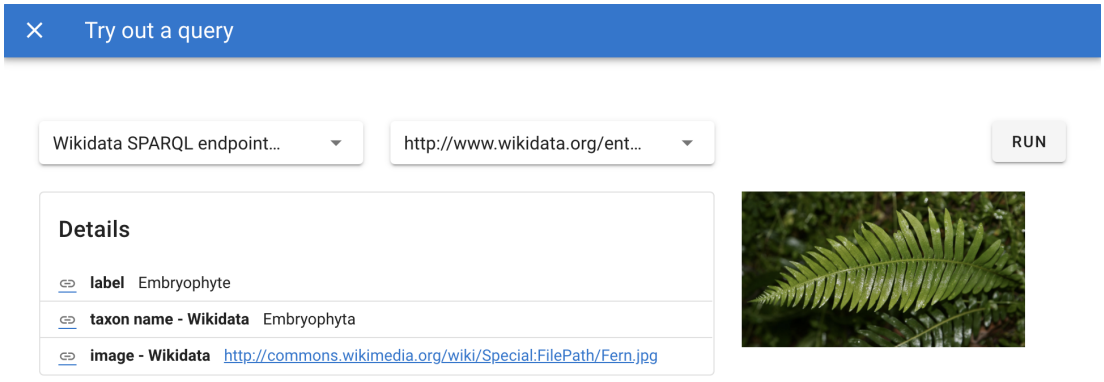

Figure 3.9: Visualizing a detail query against a selected data set

Visualizing the expansion query is a bit more complex, as the query gives information about the closest entities starting from a given node, and that requires

piloting the results as a graph, the idea here is not to replicate the KGBrowser functionality, but to give the users the ability to validate their queries and to make sure that they will get the intended results when these queries are executed in the KGBrowser as a part of the configuration (Figure 3.10).

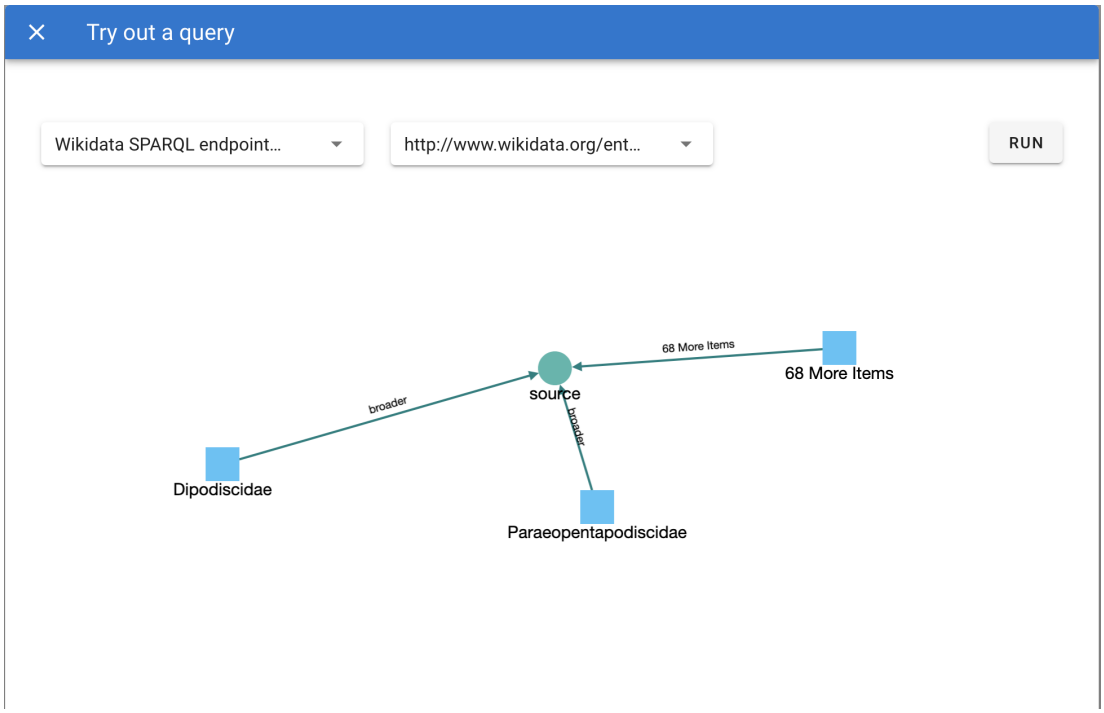

Figure 3.10: Visualizing a expansion query against a selected data set

3.2.4 Fetching configuration represented as an LOD resource

As we established, some none expert users might be interested in using the KGBrowser to visualize some data set, but they might not have the enough experience to write a configuration file from scratch, on the other hand expert users might also be interested in reusing a published configuration to examine how it behave, and it would be a bit absurd to re-write the whole configuration file from scratch just to change a couple of lines.

The solution was to give the users the possibility of loading a configuration file that was already published as an LOD resource, whenever the users initiate the

configuration creation process, they will be given an option to load this configuration from an LOD resource if it exists or from a predefined list of grouped and categorized meta configurations that were already published (Figure 3.11).

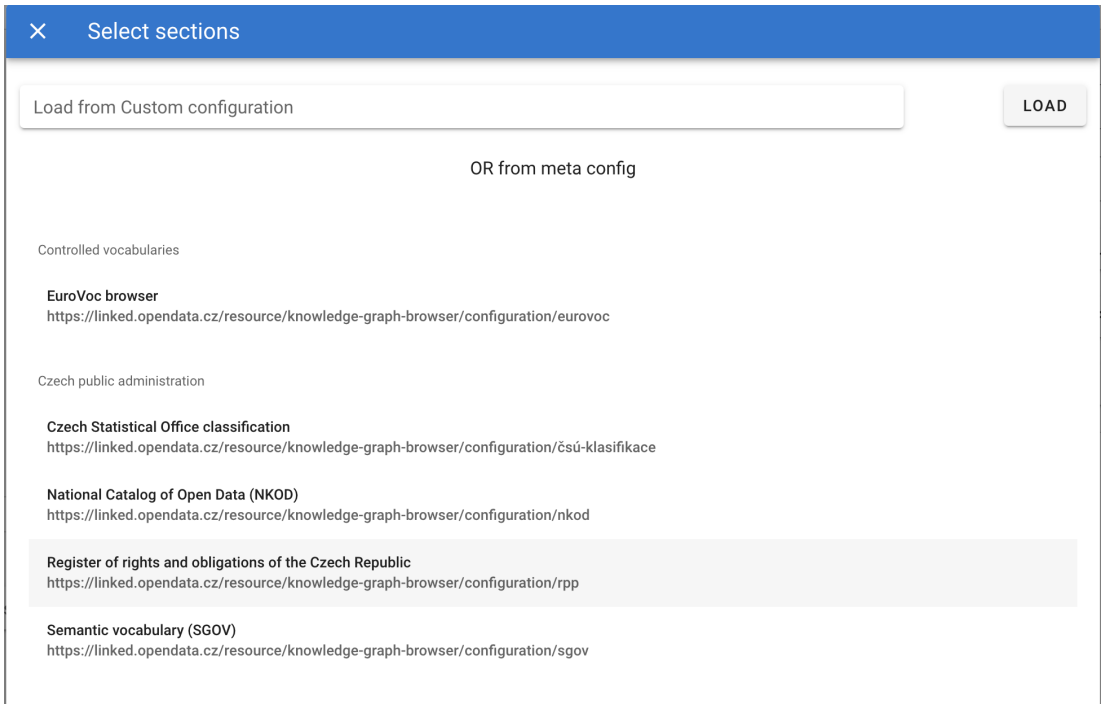

Figure 3.11: The option to fetch the configuration from LOD

If the resource does not exist or it is missing some information the user will alerted, otherwise the system will load all components of this configuration, then it will present the user a modal that contains the separate components of the configuration, so they can choose what to include and what to ignore, after clicking the save button the system will populate the template with the selected parts, and will give the users the ability to add and modify the needed sections (Figure 3.12).

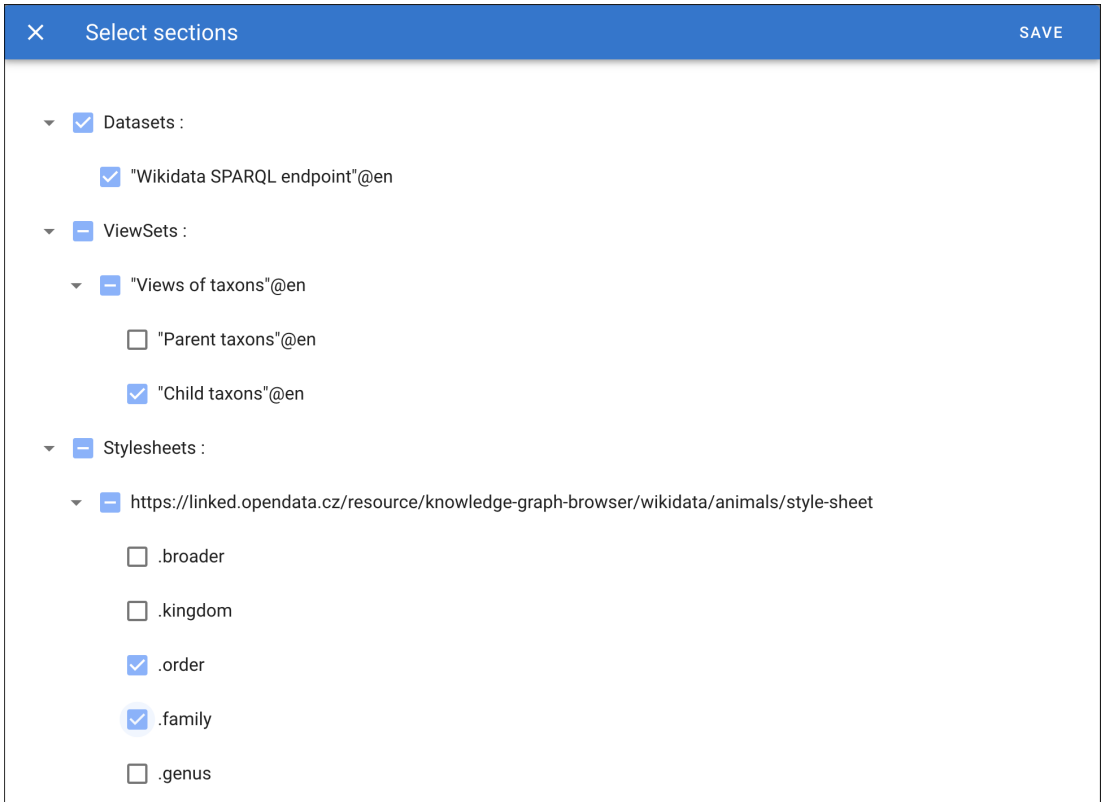

Figure 3.12: Choosing which components to include from a loaded configuration

3.2.5 Using meta configurations

The main purpose of this feature is to establish the re-usability of the configuration components. so the whole process of creating a new configuration is faster and more efficient, this feature is split into two main functionalities.

The first functionality is to give the users the ability to assign the configuration that they are working on into a meta configuration group, we have the meta configuration groups published as LOD resources and the tool will fetch them with an HTTP call and list them for the users to choose from, and this is an optional step. Users will click on *Assign to a meta config group* button, then choose a group from the list (Figure 3.13).

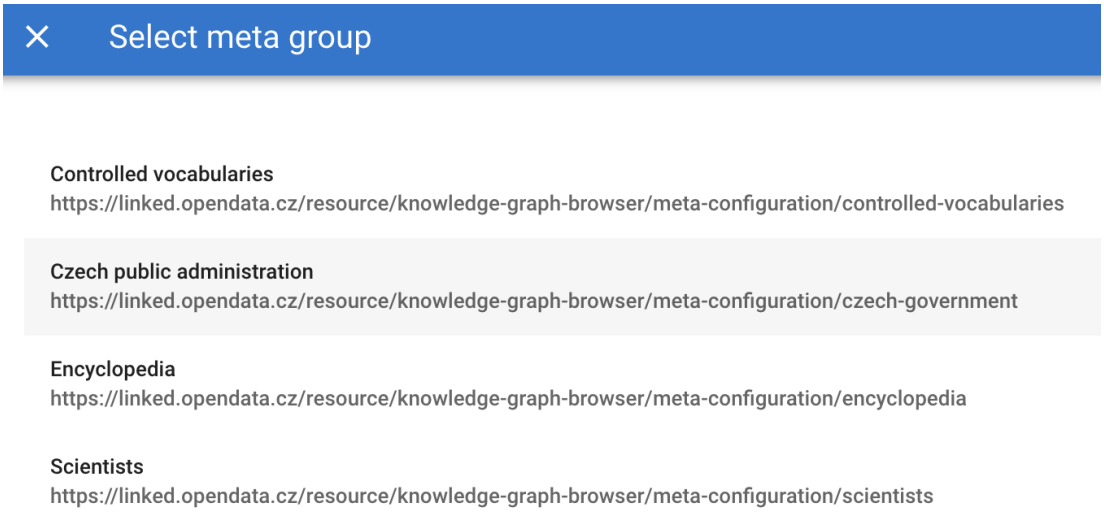

Figure 3.13: Choosing a meta group of be assigned to

Once a group is selected, the system will display this group to the users and they can remove or modify it at any time (Figure 3.13), if there is a group selected, the system will add "*<meta_configuration_group_uri> a browser:MetaConfiguration ; browser:hasMetaConficuration <configuration_file_uri>*." to the generated configuration file.

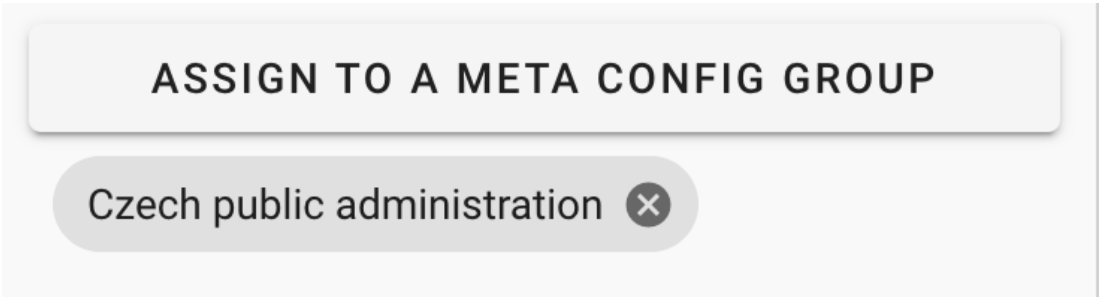

Figure 3.14: Managing the selected meta group

The second part of this feature is actually giving the users the ability to relay on published configurations as templates, as they might be interested in reusing some of the published components as an optional functionality. The user needs to click on *Use Meta Config as a template* button, this will list to the user all the available meta configurations groups with their assigned configurations files (Figure 3.15), users can select as many configurations as they need, then click the save button, then the system will load the list components for the selected configurations.

Figure 3.15: List meta groups and configurations

If the users provided the intended meta configurations and the system was able to load groups, then the users will be able to select components from a list of suggestions, for example if the user is adding a new view set he/she will have two options, either provide a new URI or choose the URI of a view set that was loaded from the selected meta group (Figure 3.16).

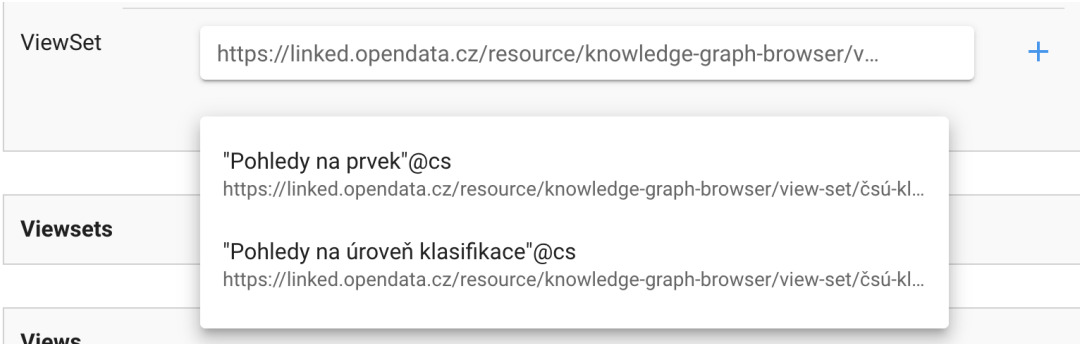

Figure 3.16: Suggestions of view sets

The list of suggestions consists of two main parts, the human readable title and the URI of the component, in some case such as style sheets where there is not title, we are providing only the URI (Figure 3.17).

Figure 3.17: Suggestions of style sheets

Upon selecting one of the suggestions, the system will load its properties and modify the needed parts of the current configurations file.

4. Implementation overview

We established that the system is split into four different modules encapsulated within docker containers for more efficient development and deployment, these modules are defined as follows: (Figure 4.1) System modules and interactions

1. Front-end module
2. KGserver
3. Database module
4. Back-end module

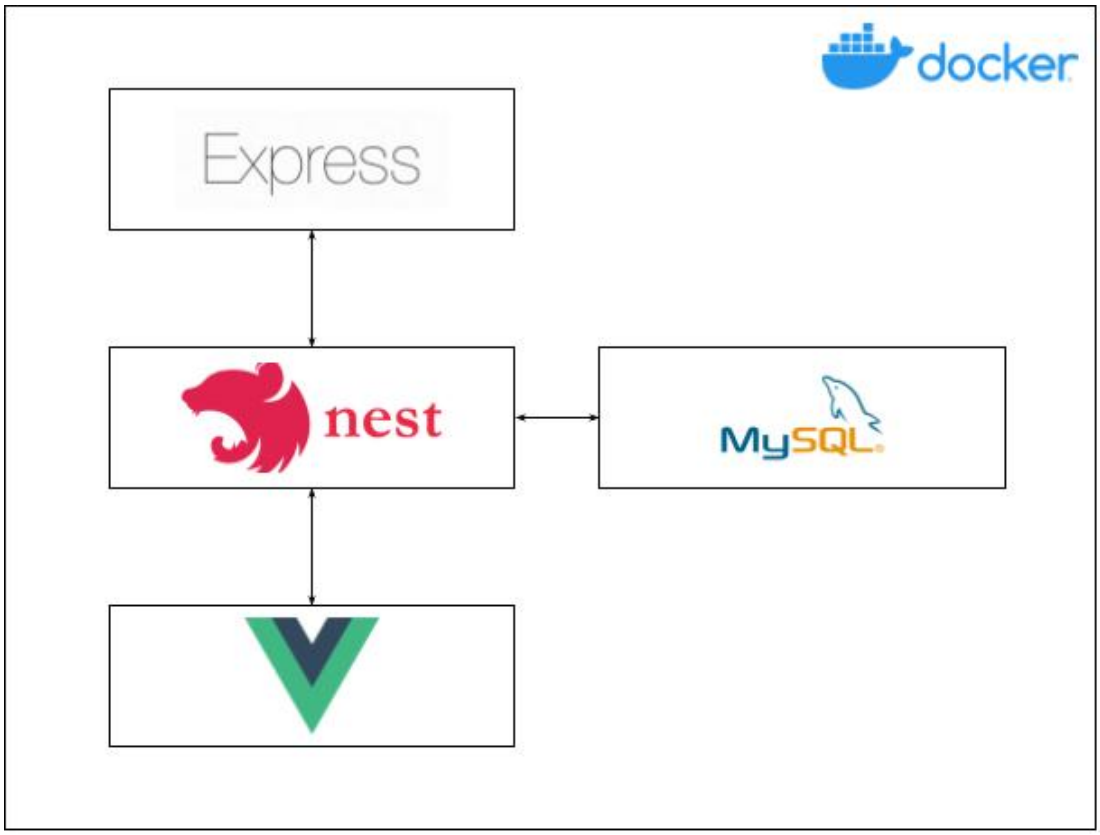

Figure 4.1: The system modules and interactions

4.1 Front-end module

The front end app is the user friendly interface, with visualization and input components, it is built using vuejs, which is a JavaScript[1] framework, it was very important to choose a modern tool for creating such an app, because that would save time and effort, vuejs offers the ability of creating reusable components, that would be good not only for code management and maintenance, but also for reusing the same components in places where that is feasible.

Vue component

The vue component is a reusable Vue instance with a name, it could be very simple and straight forward such as a button or an input field, or it could be very complex as a group of components with complex logic and features, each vue application has an entry point that serves as a parent for all other components. (Figure 4.2) Shows the vue application and components hierarchy [2]

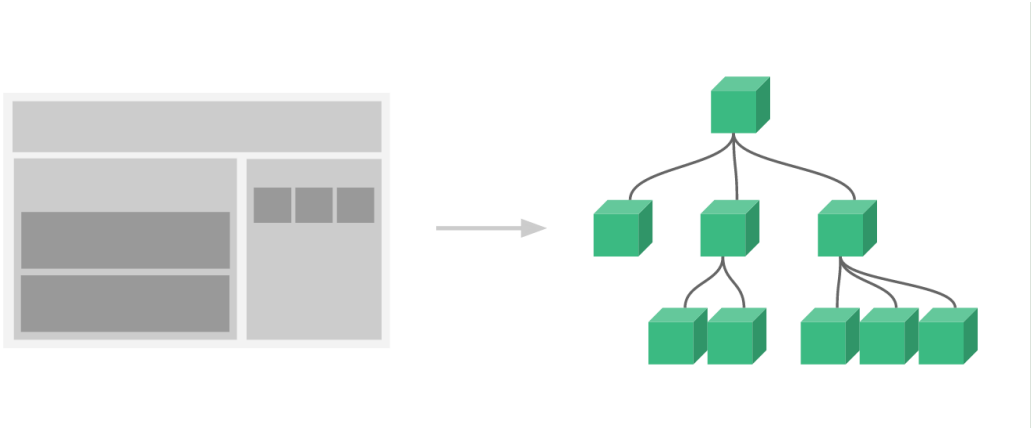

Figure 4.2: Vue application and components hierarchy

Each component contains three main parts, the first part is the template, which is the HTML [3] that will be rendered in the browser, then the script part, which contains the javascript code that is responsible for the logic of rendering the component and its functionality, and finally the style part where UI features are

[1] https://www.javascript.com/
[2] https://vuejs.org/v2/guide/components.html
[3] https://www.w3.org/html/

applied. (Figure 4.3) Shows the component structure

```
<template>
  <button @click="submit" class="button">Save</button>
</template>

<script>
export default {
  name: "SampleComponenet",
  methods: {
    submit() {
      alert("submitted");
    },
  },
};
</script>

<style>
.button {
  color: red;
}
</style>
```

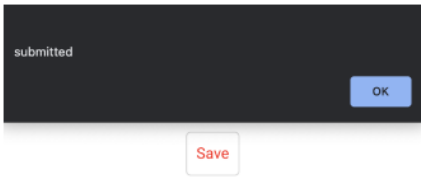

Figure 4.3: Vue component structure

Keeping that in mind, we started designing the components that are going to be used in the software solution, using a bottom up structure, beginning with shared components, in the case the user input, we called it a node, each node represents a property and it has its own attributes and features, such as the type of node (free text, a SPARQL editor, an IRI input or a drop down box), also each node can either accept one input value such as the queries or multiple values such as visual styles, all these configuration are stored in a template file, so in case we needed to add a new node type or modify an existing node type we just need to modify two places, the node component and node template.

The next component is the NodesSection component, each nodes section contains a group of nodes and it represents a configuration component of the configuration file, in the (Figure 4.4) you can see that each NodesSection represents a view component, and finally a we have a group of NodesSections that represents section of the configuration file.

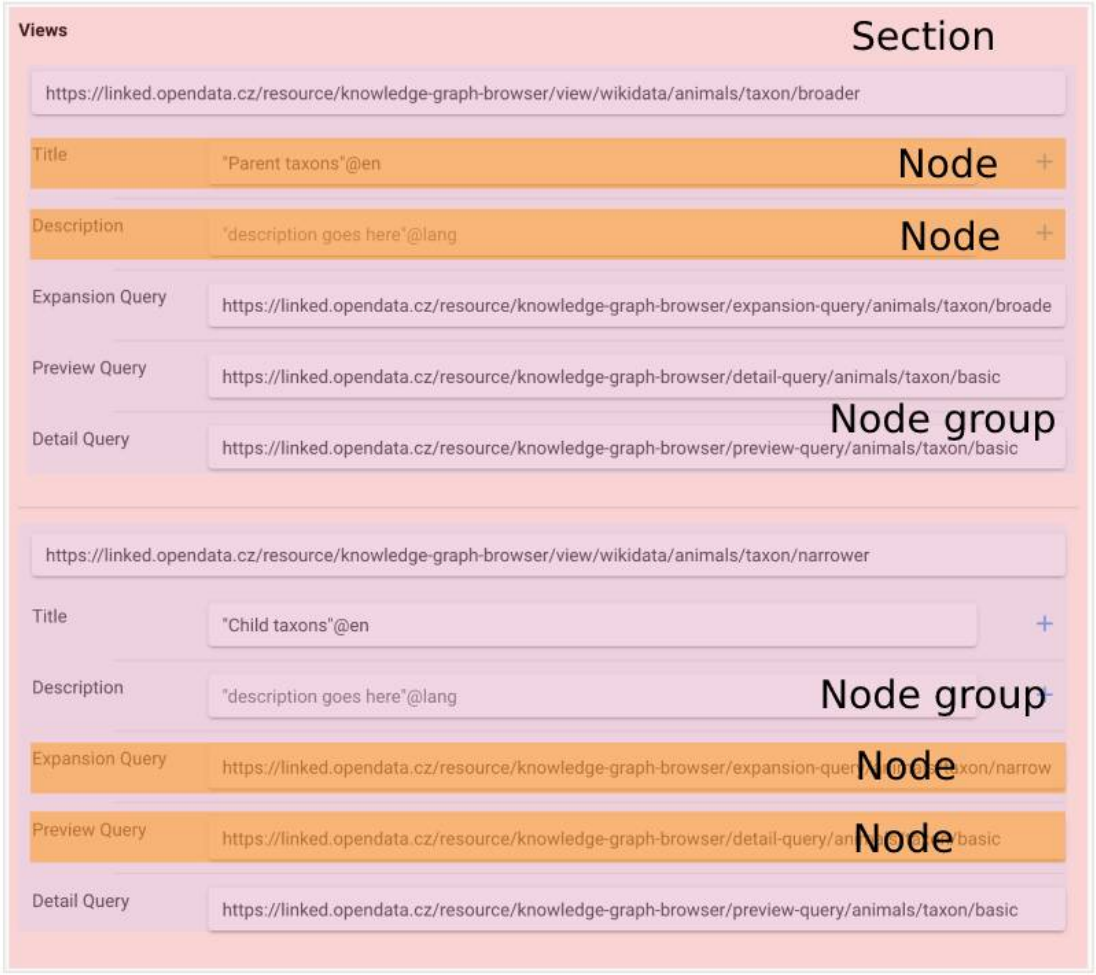

Figure 4.4: How the configuration file structure was represented as UI components

Communicating between vue components

As the configuration file contains multiple sections and these sections are interlinked and connected, their UI representation is also linked, for example when adding a view to a view set, it should be represented in the views section and if it does not exist it should be added, this type of communication between the nested vue components is a bit complex to achieve straightforward, that is why we needed a more efficient method of communication between the components using vuex.

Vuex is the central store for the application. this "store" is basically a container that holds the application state. its mission is to create a global state for the application, then components can listen to the changes in the state or parts of it by subscribing to the store, or they can dispatch an action to modify the state

of parts of it, and the store will handle notifying the subscribed components. (Figure 4.4) shows how to handle communication between vue components using vuex.

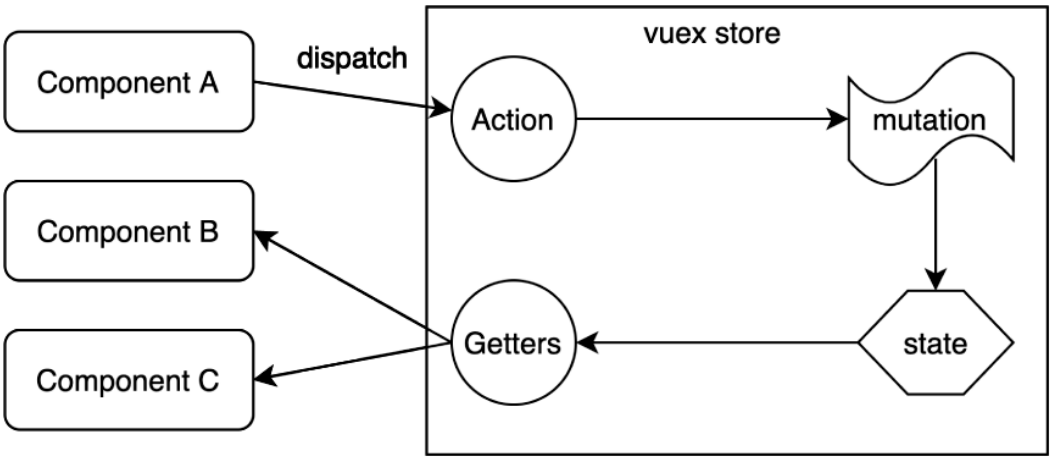

Figure 4.5: How to handle communication between components using vuex

Communicating with back-end module

The communication between the front-end the back-end modules is carried by asynchronous request/response pattern, using axios [4]. Axios is a promise [5] based HTTP client for the browser, which means whenever the request is sent to the back-end, it will run asynchronously without blocking the main thread that is responsible of handling the UI rendering.

After we get a response we have two cases, we either get a response with success code, then we update the application state which in return updates the components state, or we get an error code then we show an error message.

4.2 KGserver

KGserver is an open source application written in expressjs [6], it provides the main logic and functionality in the KGBrowser. The main idea behind the integration

[4]https://github.com/axios/axios
[5]https://javascript.info/promise-basics
[6]https://expressjs.com/

with the KGserver is that we needed two main functionalities provided by it.

The first thing, we needed to load meta configurations components that represented as a remote LOD resources, secondly we needed to run expansion, preview and detail queries against a specific data set. To achieve this level of integration we had two options, we either move the code from KGserver to the back-end module, or integrate the KGserver as a part of the system, we went with latter option as it seems to be a more logical choice, since we have a modular dockerized infrastructure.

After integrating this module in the system we faced a couple of obstacle, the endpoints that are responsible for running the queries will not work out of the box, the first issue we noticed, that these endpoints were expecting an IRI for a view that is is a part of a configuration file and published as LOD and from this view it will extract the data set information and intended query (expansion, preview or detail), along with view the endpoint is expecting to get the actual node that will be used as a seed for the query, so the solution was to modify these endpoints to accept the needed query as a text written by the user, plus the actual node that will be used as a seed for the query along side the data set information.

4.3 Database module

This module serves as a storage for the application, we are using mysql database to store the configuration file information and status, so the users can view or modify at any point of time, we are storing the file content, and to be more specific we are sorting the state of the front-end module, the application is serializing the state as string, sending it to the back-end to store in the database

While fetching the file information form the database, we reverse the previous information fetching the string content then parsing it as JSON[7] object so the

[7]https://www.json.org/json-en.html

front-end module will be able to understand it.

4.4 Back-end module

The back-end module serves as the skeleton of the system, it works as a bridge between the front-end module and the rest of the system, the front-end module can still provide some functionality without the need to connect to other modules, while a couple of features are being delivered by the back-end module.

We used nestjs for building the module, which is a nodejs framework for building server side application, the frameworks uses expressjs as an engine, and it uses typescript[8].

This module works on two different levels, the first level is a bridge between front-end module and the database module, and to achieve that we needed to integrate TypeOrm[9] with nestjs to be able to connect to the database, in typeorm each table in the database is represented by an entity class with the column names as properties, then typeomr will provide needed functionality for managing the table, for example adding an new property to the entity class will trigger typeorm sync operation and it will alter the table by adding a new column.

The second group of operations in the back-end module is that it servers as a proxy and reverse proxy between the front-end module and the KGserver, so instead of exposing the KGserver directly to the front-end module, in this way we keep the front-end module aware only of a single point of communication which is the back-end module.

At some point when the user needs to fetch the full information of a configuration which is published as an LOD resource, we needed to add some extra functionality to the back-end module. We can call each component URI using the HTTP protocol and specifying the HTTP header *Accept* as *'application/json'*, and this

[8]https://www.typescriptlang.org/
[9]https://typeorm.io/

will return a json formatted configuration. Using this fact, the back-end module will call each component URI to fetch the full information of a published configuration file, starting by the configuration IRI then we fetch the view sets and style sheets, after that for each view set we fetch its view and so on, once the whole structure is fetched we return the information to the front-end module.

(Figure 4.6) shows how to fetch configuration file components using the back-end module.

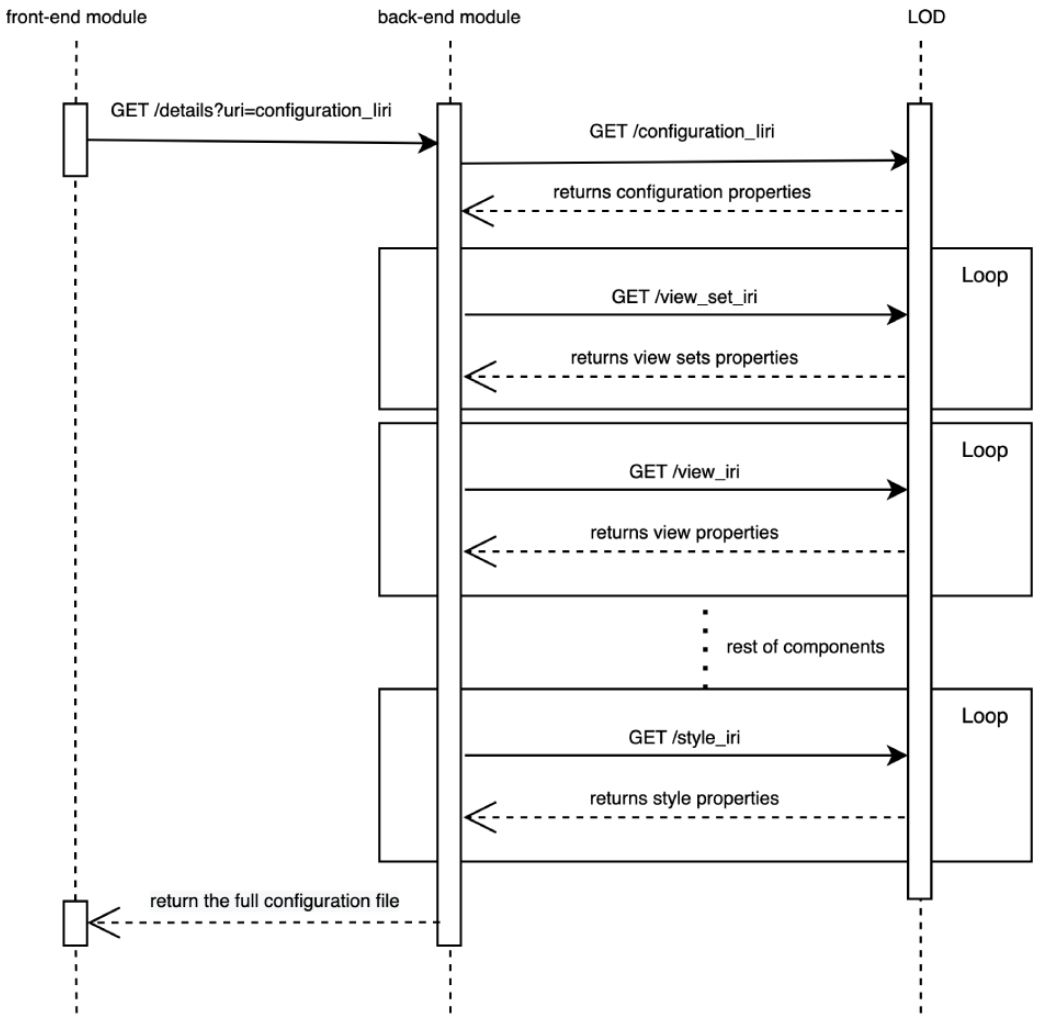

Figure 4.6: How to fetch configuration file components using the back-end module

A similar approach was used to load the meta configurations groups and details from the KGserver, the KGserver has an endpoint for fetching and parsing a meta configuration group and its meta configurations list, the back-end module will receive the meta group URI and it will call and parse recursively the configuration

uris until it fetches all their meta components. (Figure 4.7).

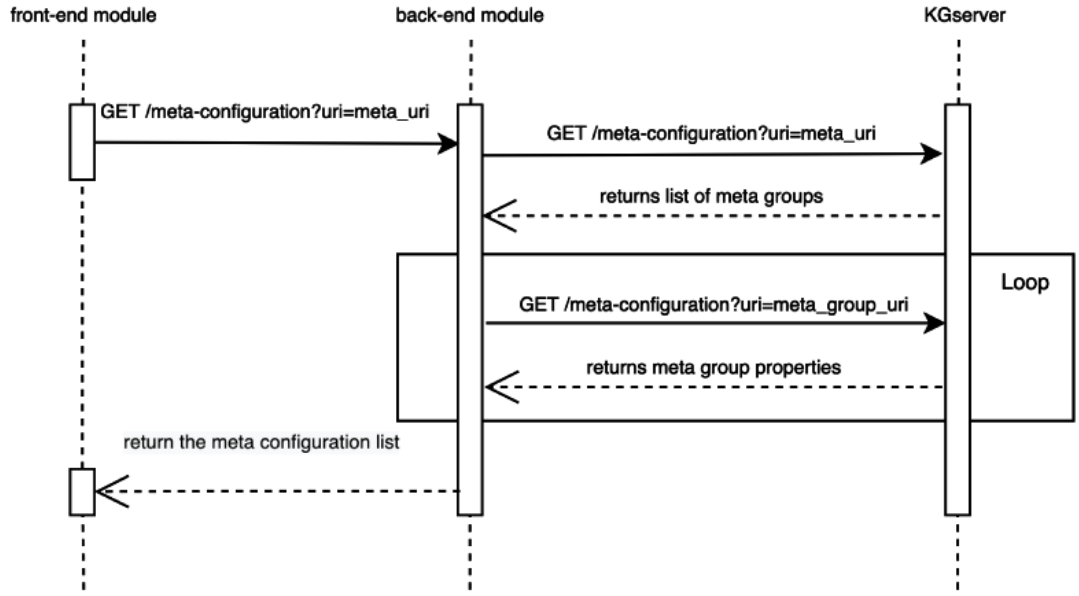

Figure 4.7: How to fetch meta configuration components from KGserver using back-end module

5. Testing

In order to test this tool, and to verify if it fulfills the desired requirements of the stakeholder, we performed couple of testing scenarios on the tool. the main idea behind the tool is to give the users a seamless experience while creating configuration files, with an advanced visualization.

And since this tool is an experimental tool and due to the situation during the last couple of months while writing this thesis, it was not easy to find users with a good level of experience in knowledge graphs who can help in testing, so I created a couple of testing scenarios and validated with my supervisor on our meetings.

First set

This testing set was conducted after the initial prototype, we went through a couple of scenarios, and here is a sample.

1. Start a new configuration file.
2. Create configuration section.
3. Fill in the generated components.
4. Generate an RDF turtle preview.
5. Download the generated file.

As an outcome of this test, we noticed that creating the configuration file might take a longer time than expected and that might require the user to keep the application running to return later and resume their work, so as solution we introduced the ability to save the progress and retrieve the saved work at any point.

Second set

The second round of testing was held after the gathering the second set of requirements, it pointed out some issues with the general user experience, as an example.

1. Start a new configuration file.
2. Fill in the components.
3. Download the generated file.
4. Publish it as an LOD resource.
5. Edit the file again.
6. Republish it as a LOD resource.

The main issue that we faced during this process was the idea that with every small change which is done on the configuration file, there was no way to know how it would perform on the KGBrowser, unless we publish it first, and that did not provide a good user experience, keeping that in mind we decided to give the users the ability get an estimated preview on how the KGBrowser will respond to the written queries, by visualizing the queries in the tool, without the need to publish as an LOD resource.

Third set

The final set of tests happened not far ago from the time when this thesis was written, it included a couple of testes to see the validity of the solution.

1. Start a new configuration file.
2. Click *FETCH FROM REMOTE RESOURCE* button.
3. A dialog should appear.
4. Enter the IRI of an already published configuration or select form the predefined list.

5. Select the required components.
6. The tool should populate the configuration template with the selected components.

the previous test will show the tool ability to fetch the configuration component from an LOD resource, while the following test will present how to visualize a query in the tool directly without the need to use the KGBrowser.

1. Start a new configuration file.
2. Enter values for configuration properties (view sets, starting nodes) in particular.
3. Add a view to the generated view set.
4. In the generated view fill in the IRIs of the expansion, detail and preview queries.
5. In each of the queries fill in the data set the query values.
6. Populate the data sets information.
7. Go to expansion query and click on the *RUN QUERY* button.
8. Choose a data set and seeding node.
9. Click on the *RUN* button.
10. The tool should show a preview of the query.

5.1 Automated tests

We used two types of tests in both the front-end module and the back-end module. In the front-end module we covered the shared components with unit testing using jest [1], as an example we tested the *SuggestionPanel.vue* component, which

[1] https://jestjs.io/

is responsible for showing a list of suggested components when the users select a meta configuration as template.

When a user selects suggestion from the list, this suggestion should not be present to the user again to avoid duplicate input, so in the following test (Figure 5.1) we are passing a list of three items and assuming that the user has already selected one item, so the component must render only two items.

```
it("renders menu with 2 items", () => {
  const items = [
    { id: "item-1", title: "title 1" },
    { id: "item-2", title: "title 2" },
    { id: "item-3", title: "title 3" },
  ];
  const execlude = ["item-3"];

  const wrapper = mount(SuggestionPanel, {
    localVue,
    vuetify,
    propsData: { items, execlude },
  });

  expect(wrapper.findAll(".v-list-item").length).toBe(
    items.length - execlude.length
  );
});
```

Figure 5.1: Unit testing example

In the back-end module we have applied another type of testing which is end to end testing, we covered all back-end endpoints with end to end tests for the main functionality scenarios, we are also using jest testing framework to preform these test.

In the following example (Figure 5.2), we are testing the endpoint that is responsible for visualizing the expansion query, we are preparing the input for the endpoint which consists of the SPARQL query, the starting node as the resource

and the data set information. The expected response code is 200 which means that the query was executed correctly, and we are matching the response structure as well.

```
it('/v1/queries/expand (POST)', () => {
  return request(app.getHttpServer())
    .post('/v1/queries/expand ')
    .send({
      dataset: {
        sparqlEndpoint: 'https://query.wikidata.org/sparql',
        accept: 'application/sparql-results+json',
      },
      resource: 'http://www.wikidata.org/entity/Q7377',
      sparql:
        'PREFIX wd: <http://www.wikidata.org/entity/> PREFIX wdt: <http://www.wikidata.org/prop/direct/> |
    })
    .expect(200)
    .then(res => {
      expect(res.body.edges.length).toBeGreaterThan(0);
      expect(res.body.nodes.length).toBeGreaterThan(0);
      expect(res.body.nodes.map(el => el.label).includes('Tetrapoda')).toBe(
        true,
      );
      expect(res.body.types.length).toBeGreaterThan(0);
    });
});
```

Figure 5.2: End to end testing example

Conclusions and future plans

This thesis describes the work on building a KGBrowser configuration tool from scratch, The main purpose is to improve the user experience while generating these configurations, and reduce the user struggle while doing so.

The process starts with analyzing the initial requirements and the samples of configuration files, then building a prototype based on the outcome, to see if we can put the idea into actual development. Furthermore, these requirements were modified and adapted in iterations and went through a few stages of testing in order to enhance the user experience.

After the first iteration the system grew by adding new modules to fit the needs and to adapt to the new requirements, which reflected on the system architecture, each module of this architecture was encapsulated in a separate unit using docker. The main focus of this thesis is the implementation of the tool, and it dives as well into details of all the system modules such as KGserver, and how it was used to achieve the main goal.

The work on this tool was done during the preparation of this thesis, however it is just a small step in a very long journey, which needs to be followed by future improvements, and here are some features that can be integrated with the tool.

1. At the moment the tool only provides the user with the ability to download the generated configurations, then users will have to publish the configurations manually as LOD resources, but it would be a good idea to enhance the tool by adding a feature that publishes the configurations automatically into an LOD repository of the users choice.

2. Another good to have feature is the ability to edit components that were loaded from an LOD resource, the current implementation allows the users to reuse components from an already published configurations without the ability to fully edit these components, so it would a good idea to allow users

to modify some parts of these components.

Bibliography

[1] Bizer, C.; and Heath, T. 2011. *Linked Data: Evolving the Web into a Global Data Space.*

[2] Bizer, C.; Heath, T.; and Berners-Lee, T. 2009.. *Linked data - the story so far.*

[3] Po, L.; Bikakis ,N.; Desimoni , F.; and Papastefanatos, G. 2020. *Linked Data Visualization: Techniques, Tools, and Big Data.*

[4] Berners-Lee, T.; Hollenbach, J.; Lu, K.; Presbrey, J.; Prud'hommeaux, E.; and Schraefel, M. 2007.. *Tabulator redux: Writing into the semantic web.*

[5] Koch, J.; Franz, T.; and Staab, S. 2008. *Lena - browsing rdf data more complex than foaf.*

[6] Koch, J.; Franz, T.; Staab, S.; Dividino, R. 2010. *LENA-TR : Browsing Linked Open Data Along Knowledge-Aspects.*

[7] Haase, P.; Herzig, D.; Kozlov, A.; Nikolov, A.; and Trame, J. 2019. *metaphactory: A Platform for Knowledge Graph Management.*

[8] Ji, S.; Pan, S.; Cambria, E.; Marttinen, P.; and Yu, P. S. 2020. *A survey on knowledge graphs: Representation, acquisition and applications.*

[9] Wang, Q.; Mao, Z.; Wang, B. and Guo, L. 2017. *Knowledge graph embedding: A survey of approaches and applications. IEEE trans. Knowl. Data Eng.*

[10] Wood, D.; Lanthaler, M. and Cyganiak, R. 2014. *RDF 1.1 concepts and abstract syntax.*

[11] Yan, J.; Wang, C.; Cheng, W.; Gao, M. and Zhou, A. 2018. *A retrospective of knowledge graphs. Frontiers Comput. Sci.*

[12] Desimoni, F.; Bikakis, N.; Po, L. and Papastefanatos, G. 2020. *A comparative study of state-of-the-art linked data visualization tools. In V. Ivanova, P. Lambrix, C. Pesquita, and V. Wiens, editors, Proceedings of the Fifth International Workshop on Visualization and Interaction for Ontologies and Linked Data co-located with the 19th International Semantic Web Conference (ISWC 2020), Virtual Conference (originally planned in Athens, Greece), November 02, 2020, volume 2778 of CEUR Workshop Proceedings.*

[13] Desimoni, F. and Po, L. 2020. *Empirical evaluation of linked data visualization tools. Future Generation Computer Systems.*

[14] Klímek, J.; Nečaský, M. and Škoda, P. 2019. *Survey of tools for linked data consumption. Semantic Web.*

[15] C.R, K.and Thomas, S. 2011. *Requirement Gathering for small Projects using Agile Methods .*

List of Figures

List of Tables

List of Abbreviations

LOD Linked Open Data

KGBrowser Knowledge Graph Browser

RDF Resource Description Framework

SPARQL SPARQL Protocol and RDF Query Language

KGVB Knowledge Graph Visual Browser

IRI Internationalized Resource Identifier

UML Unified Modeling Language

CSS Cascading Style Sheets

UI User Interface

POC Proof Of Concept

KGserver Knowledge Graph Server

Annex - developer handbook

The source code of this tool is available publicly on *https://gitlab.com/mahran-omairy/thesis-project/-/tree/master*, where you can clone it an run it locally, and a live demo is published on *https://kgbrowser-config.me/*.

The repository contains two main branches the **master** branch which contains latest version of the code, and the **production** branch which has the latest version of the code plus an apache server[2] container which is being used for deploying the application on a production environment. The steps needed for running the tool on a production environment are as follows:

1. Get the remote server ready by installing docker[3] and make sure that you have ports 80 and 443 exposed.
2. Clone the gitlab repository and checkout to the production branch.
3. In the *docker-compose.yml* file you need to modify the following variables.
    - MYSQL_USER and LOCAL_DB_USER should be assigned the same value for database user name (Optional).
    - MYSQL_PASSWORD and LOCAL_DB_PASS should be assigned the same value for database user password (Optional).
    - MYSQL_DATABASE and LOCAL_DB_NAME should be assigned the same value for database name (Optional).
    - MYSQL_USER and LOCAL_DB_USER should be assigned the same value for database user name (Optional).
    - MYSQL_ROOT_PASSWORD for mysql root user passowrd (Optional).
    - LETS_ENCRYPT_EMAIL a valid email for letsencrypt ssl certificate (Required).

[2]https://httpd.apache.org/
[3]https://www.docker.com/

- HOST_NAME_FROM the domain where the app will be served, the domain should be pointed to the server before deployment (Required).
- HOST_NAME_FROM_API the api domain for the back-end module, the domain should be pointed to the server before deployment (Required).

4. In *vue/src/services/api.js* file modify *baseUrl* variable to be as follows *https://your-api-domain/api/v1*, similar to the value of HOST_NAME_FROM_API (Required).

5. Once ready run the command **docker-compose up -d −−build** and the apache container will download the ssl certificate and the other container will build and run the modules.

To run the automated test for the front-end module, navigate to */vue* folder and run **npm install** then **npm run test** and the system will execute the defined set of tests, and to add a new test set navigate to the folder *vue/tests/unit* and add your test in a new file that matches the tested component name.

To run the automated test for the back-end module, first make sure that you are on **master** branch, and run the command **docker-compose up -d −−build** to make sure that the database module is running, then navigate to */nestjs* folder and run **npm install** then **npm run test** and the system will execute the defined set of tests, and to add a new test set navigate to the folder *nestjs/test* and add your test in a new file that matches the tested component name.

The back-end module is also integrated with a swagger[4] interface that can be helpful for manually testing an endpoint or reading more about the expected requests and responses from each endpoint, this swagger instance can be accessed locally on *http://localhost:5001/docs/* or on the current live domain *https://api.kgbrowser-config.me/docs/*, and if you deploy on a custom domain, swagger can be accessed on *https://your-api-domain/docs*.

[4]https://swagger.io/