



**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

**BACHELOR THESIS**

Jan Kuželík

**Automatic schema extraction from RDF  
Data**

Department of Software Engineering

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

Study programme: Computer Science

Study branch: Software and Data Engineering

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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

Thank you to my supervisor for your time, patience and guidance.

Thank you to those close to me for your continued support.

Thank you to the open source community for your priceless contributions.

Title: Automatic schema extraction from RDF Data

Author: Jan Kuželík

Department: Department of Software Engineering

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

Abstract: The Resource Description Framework (RDF) is a model for the representation of semantic data. RDF allows the storage of information without a fixed schema. This provides more flexibility but the lack of a fixed schema poses a significant entry barrier to the utilisation of the stored data. The SPARQL language is used for querying an RDF database.

Several works exist in the domain of schema extraction from SPARQL endpoints. Most tend to provide a visual representation of the schema, rather than an immediately usable output. Many of these solutions perform a very thorough and lengthy extraction unsuitable for a web application environment and some are not even available online.

This thesis introduces TypeSPARQ, an open-source web application for extracting schemata from SPARQL endpoints. TypeSPARQ creates a visualisation of the endpoint's schema and offers options for exporting it. TypeSPARQ integrates with LDKit, which provides type-safe access to SPARQL endpoints for TypeScript applications. These tools combined offer TypeScript developers a seamless process from endpoint exploration to integrating the endpoint within their projects.

Keywords: schema-extraction RDF SPARQL schema

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.0.1	Resource Description Framework . . . . .	3
1.0.2	SPARQL Protocol and RDF Query Language . . . . .	4
1.1	Motivation . . . . .	4
1.1.1	Schema extraction from RDF data . . . . .	5
1.2	Contribution . . . . .	6
<b>2</b>	<b>Related Work</b>	<b>7</b>
2.1	Enabling ad-hoc reuse of private data repositories through schema extraction . . . . .	7
2.2	HyperGraphQL . . . . .	9
2.3	TopBraid . . . . .	10
2.4	Stardog . . . . .	10
2.5	GraphQL-LD . . . . .	11
2.6	UltraGraphQL . . . . .	12
2.7	ViziQuer . . . . .	12
2.8	LODSight . . . . .	13
2.9	A Visual Aide for Understanding Endpoint Data . . . . .	14
2.10	LD-VOWL . . . . .	15
2.11	VizLOD . . . . .	16
2.12	SPARQLess . . . . .	16
2.13	Simplod . . . . .	17
2.14	Summary . . . . .	18
<b>3</b>	<b>Requirements</b>	<b>21</b>
3.1	Non-Functional Requirements . . . . .	21
3.2	Functional Requirements . . . . .	21
3.3	Application design . . . . .	22
3.3.1	Non-functional requirements ( $R1, R2$ ) . . . . .	22
3.3.2	Schema visualisation ( $R3$ ) . . . . .	23
3.3.3	Schema extraction ( $R4$ ) . . . . .	23
3.3.4	Querying rate ( $R5$ ) . . . . .	23
<b>4</b>	<b>External integration</b>	<b>25</b>
4.1	LDKit . . . . .	25
4.2	Schema Import . . . . .	25
4.3	Schema Export . . . . .	26
<b>5</b>	<b>Developer documentation</b>	<b>27</b>
5.1	Technologies and Frameworks . . . . .	27
5.1.1	Single Page Application . . . . .	27
5.1.2	Technologies used . . . . .	27
5.1.3	NPM . . . . .	28
5.1.4	TypeScript . . . . .	29
5.1.5	Vite . . . . .	29

5.1.6	Vue.JS	30
5.2	Dependencies	30
5.2.1	Tailwind CSS	30
5.2.2	Pinia	31
5.2.3	Vue Flow	31
5.2.4	Prism	31
5.2.5	Prettier	32
5.2.6	Zod	32
5.3	Project Structure	32
5.4	Vue Router	32
5.5	Components	33
5.5.1	CustomNode	33
5.5.2	NodeModal	33
5.5.3	ExportModal	34
5.5.4	ImportModal	34
5.6	Visual Design	35
5.7	Schema queries	35
5.7.1	QueryQueue	36
5.8	Stores	37
5.8.1	EndpointStore	37
5.8.2	VisStateStore	37
5.9	Data model	37
5.10	Assets	38
5.11	Tests	38
5.12	Points of Extension	38
5.12.1	Export formats	38
5.12.2	Import format	39
5.12.3	Visualisations	39
5.12.4	Flow layouts	40
<b>6</b>	<b>Administrator documentation</b>	<b>41</b>
6.1	Prerequisites	41
6.2	Getting started	41
6.3	Build	41
6.4	Hosting	42
<b>7</b>	<b>User documentation</b>	<b>43</b>
7.1	Using the web application	43
7.2	Using the default exported schema	44
<b>8</b>	<b>Evaluation</b>	<b>46</b>
8.1	Queries evaluation	46
8.1.1	Endpoint Evaluation	47
<b>9</b>	<b>Conclusion</b>	<b>51</b>
	<b>List of Abbreviations</b>	<b>52</b>
	<b>References</b>	<b>53</b>

# 1. Introduction

To get an understanding of the aim of this thesis, allow for an explanation of the concepts of the problem domain and its difficulties, which this thesis aims to resolve.

## 1.0.1 Resource Description Framework

RDF is a W3C Recommendation describing the abstract, machine-interpretable data model of information on the semantic Web. An RDF Graph consists of subject-predicate-object triples, through which the world is described. The resulting data is a directed knowledge graph with IRIs (Internationalised Resource Identifier), literals and blank nodes serving as vertices and predicates serving as edges. IRIs represent unique real-world entities, resources or concepts. Literals state literal values such as a number or a string. (W3C 2022d)

Because RDF is an abstract data model, it does not directly describe how its data is serialised. For this purpose, several serialisation techniques exist. The most common W3C-recommended formats are Turtle (W3C 2022h), N-Triples (W3C 2022b) and JSON-LD (W3C 2022a). The Listing below shows a simple example with three triples, meaning

`<http://example.org/person/Person1>` is a person called Alice with a given email address.

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/> .
2
3 <http://example.org/person/Person1> a foaf:Person ;
4   foaf:name "Alice" ;
5   foaf:mbox "alice@mail.com" .
```

Listing 1.1: Turtle serialisation example

To further facilitate the machine-interpretability of the data, RDF Schema (RDFS) (W3C 2022e) and the Web Ontology Language (OWL) (W3C 2022c) are used to describe relationships and schema of the RDF data.

RDFS is a vocabulary for describing properties and classes of RDF resources, with semantics for generalisation-hierarchies of such properties and classes (W3C 2022c).

OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality and more. An OWL representation of terms and their relationships is called an ontology (W3C 2022c).

Ontologies provide a unified schema for the data, whilst not enforcing it. This allows for different data providers to reuse the same concepts if they publish data in the same domains, but still allows them to add additional information if they desire. There is a multitude of openly available ontologies and vocabularies for many areas, which can be freely reused by any linked data provider (Group 2022).

A fundamental feature of an RDF document is its lack of a fixed schema, unlike, for example, an SQL data store. The schemata that can be described using RDFS or OWL are descriptive and not prescriptive by the nature of RDF. This allows for flexibility and the open-world assumption but complicates RDF's interoperability with more traditional rigid schema systems. The descriptive schemata

included in the RDF document can be imported from an online, well-defined and available schema, or be included directly in the data if requiring new custom definitions.

## 1.0.2 SPARQL Protocol and RDF Query Language

SPARQL is a W3C Recommendation focusing on querying and manipulating RDF data using triple patterns (W3C 2022g). It is reminiscent of SQL in its syntax. A SPARQL endpoint is accessible through HTTP and evaluates and answers received SPARQL queries. SPARQL allows queries that retrieve data and queries that modify or insert existing data. A SPARQL endpoint contains RDF and responds to data retrieval requests by providing either RDF data for CONSTRUCT queries or tabular data for SELECT queries.

For a basic idea of how a SPARQL SELECT query looks, observe Listing 1.2. The example query finds the top 20 organisations sorted by their employee count. Finding the employees is achieved in the `WHERE` clause, wherein all organisations are discovered according to their type. Subsequently, nodes that follow the property `foaf:Employs` and are also a `foaf:Person` are counted. Finally, the results are aggregated by the company and the employees are counted and ordered. As with other RDF documents, prefixes are available for visual clarity.

```
1 PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
2 PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 SELECT ?company (COUNT(?person) AS ?employeeCount)
5 WHERE {
6     ?person a foaf:Person .
7     ?company a foaf:Organization ;
8             foaf:Employs ?person .
9 }
10 GROUP BY ?company
11 ORDER BY DESC(?employeeCount)
12 LIMIT 20
```

Listing 1.2: Example SPARQL query

For a more complex understanding, please refer to the specification (W3C 2022g).

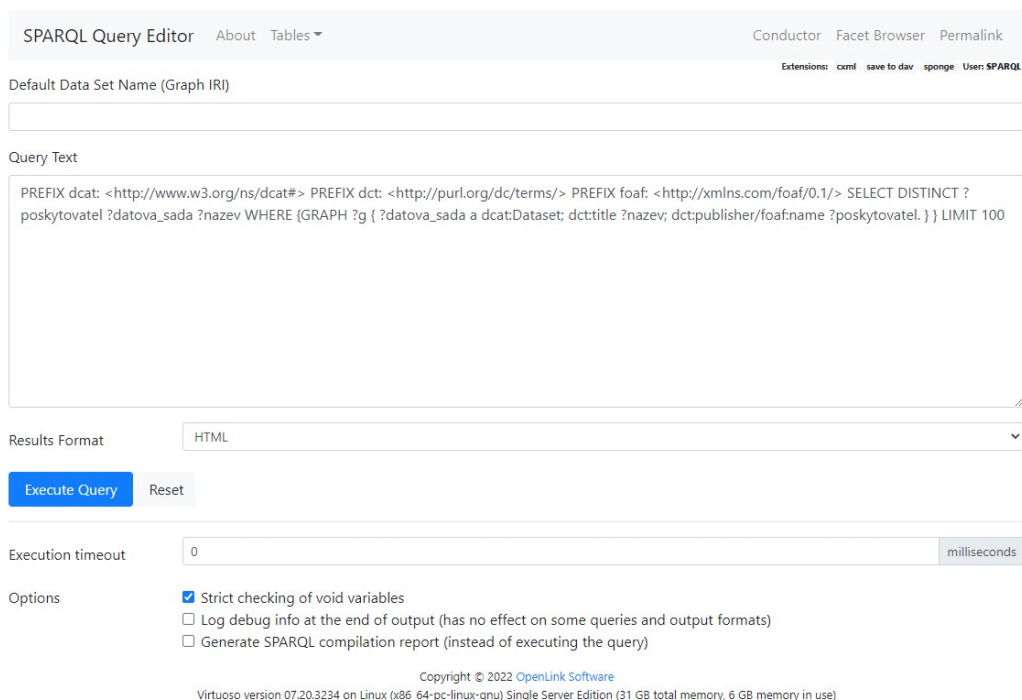
## 1.1 Motivation

Approaching a SPARQL endpoint can be somewhat intimidating for many people, especially those without extensive RDF and SPARQL knowledge. Yet these endpoints have the potential to be used by a multitude of people who may not yet have experience with them. A developer could use the endpoint for integrating external or public data into applications. Data analysts may look to utilise publicly available SPARQL endpoints for their models. Journalists may utilise them to look into state-published data. These people may have little to no experience with SPARQL and therefore great difficulty in exploring a SPARQL endpoint.

Endpoints such as the Czech Government, the Scottish Government or the US National Library of Medicine endpoints greet users with little more than a query input field and an execute button. Even for users with good SPARQL knowledge,



approaching an unknown endpoint will require them to initially execute a series of sampling queries to get an idea of its contents.



SPARQL Query Editor About Tables

Conductor Facet Browser Permalink

Extensions: cxml save to dav sponge User: SPARQL

Default Data Set Name (Graph IRI)

Query Text

```
PREFIX dcat: <http://www.w3.org/ns/dcat#> PREFIX dct: <http://purl.org/dc/terms/> PREFIX foaf: <http://xmlns.com/foaf/0.1/> SELECT DISTINCT ?poskytovatel ?datova_sada ?nazev WHERE {GRAPH ?g { ?datova_sada a dcat:Dataset; dct:title ?nazev; dct:publisher/foaf:name ?poskytovatel. }} LIMIT 100
```

Results Format HTML

Execute Query Reset

Execution timeout 0 milliseconds

Options

- Strict checking of void variables
- Log debug info at the end of output (has no effect on some queries and output formats)
- Generate SPARQL compilation report (instead of executing the query)

Copyright © 2022 OpenLink Software  
Virtuoso version 07.20.3234 on Linux (x86\_64-pc-linux-gnu) Single Server Edition (31 GB total memory, 6 GB memory in use)

Figure 1.1: data.gov.cz/sparql endpoint

Given the loose nature of RDF data, even if the endpoints are accompanied by a dataset explorer, those don't provide information about the frequency of properties for a given subject. For example a person may have a name and an email. Suppose you are looking for all people in an endpoint and get 100 results. Next, according to the explorer, you find that people can have emails. You query for people and their emails, but only receive 55 results. There was no way to discover that only 55% of people have provided an email using the simple type explorer, unless first executing this query.

Having access to an easy-to-use overview of the contents of the endpoints would make the data more approachable and usable. This would make them a more attractive for a wider audience, as well as make the regular users' querying simpler.

### 1.1.1 Schema extraction from RDF data

Schema extraction is a rather broad topic, which can be interpreted in multiple ways.

- The purely RDF-focused interpretation could be generating RDFS and OWL ontologies based on the already present data to provide an RDF description of the dataset.
- A programmatic approach would attempt to generate classes or objects in a specific language or create parsers and constructors for said objects to automatically generate them from RDF data.

- An exploratory approach would focus mainly on presenting the data to the user graphically or interactively to allow for easier exploration of the RDF dataset.

The exploratory approach is desirable to satisfy the ideas proposed in 1.1. Giving users the ability to inspect and visually reason about the otherwise hidden schema of RDF data will be a goal of this thesis. Several works offering an endpoint exploration exist and are discussed in chapter 2. However, a user inexperienced with using SPARQL will still find difficulty in extracting this data from the endpoint even if provided with a schema. For that reason, creating or enabling alternate ways of accessing the endpoints is also required. Some of the solutions described in chapter 2 provide GraphQL access to the endpoint but lack in visualisation or user experience.

To maximise the appeal of data in this form, it would be beneficial to provide access to it through means with which developers are already familiar. Developers not familiar with SPARQL that are interested in accessing the data may be put off by the query language and may look elsewhere. LDKit (*LDKit Linked Data query toolkit for TypeScript developers* 2023), a JavaScript library solution to access SPARQL endpoints exists and could fulfil this role. However, it requires prior knowledge of the endpoint’s schema – something that is assumed unknown in our case. Therefore, generating this schema would further ease the use of the data sources.

## 1.2 Contribution

This thesis provides a solution to the above-described problems by introducing TypeSPARQ, an open source<sup>1</sup> SPARQL endpoint schema exploration and visualisation tool. TypeSPARQ is a web application that provides a user-friendly view of the endpoint’s schema. It handles the extraction of existing classes, their attributes and relationships using SPARQL queries and displays this information to the user using a well-readable visualisation. Subsequently, a subset of the endpoint may be selected to be exported as a schema definition in a selected format. This schema definition may further be used by developers to query the endpoint programmatically.

The rest of this thesis is structured as follows. Chapter 2 summarises and compares current approaches to the problem of schema extraction from RDF and their shortcomings. A solution to those problems and its requirements are outlined in chapter 3. Chapter 7 introduces TypeSPARQ, a web-based tool that resolves the outlined problems. Chapter 5 includes a technical overview of the solution and its components. For hosting the application, refer to chapter 6.

---

<sup>1</sup><https://github.com/Jkuzz/sparql-explorer>

## 2. Related Work

Several works exist in the domain of RDF data visualisation and exploration and are described below. The general aim of the schema extraction process is to reveal the structure of the data for the user. These works are similar in nature, but each addresses a different part of the problem using varied approaches. These solutions tend to provide an adequate overview of the endpoints contents, but not all are functional at the time of writing of this thesis (Spring 2023). Only displaying the endpoint's structure, as some solutions do, may not be sufficient for less proficient users, who might need further assistance.

Revealing the structure of the endpoint may take several forms. An endpoint can be explored through a graph visualisation or, as is the goal of some works, via the automatic creation of GraphQL schemata. Extraction of GraphQL schemata is a related problem, as GraphQL uses a graph data model and is close in nature to RDF. Therefore, given GraphQL's popularity and support (*Code using GraphQL* 2023) it naturally lends itself as an API into the RDF world. Some listed works do not provide a fully automatic extraction, but a semi-automatic one based on a predefined context. The aim of these works is to allow easy access to semantic data for developers who are not familiar with its concepts but are familiar with GraphQL. Providing access to RDF data through GraphQL is an adjacent problem and is worth investigating to learn from its methodology. A good overview of current approaches for GraphQL querying was presented by Taelman et al. at the 2019 W3C Workshop on Web Standardisation for Graph Data (Taelman, Sande, and Verborgh 2019). This chapter extends on their prior work.

The approaches listed in this chapter were primarily discovered using Google Scholar. The queries used were similar to the keywords of this thesis. That includes the queries for "RDF schema extraction", "SPARQL endpoint visualisation" and "SPARQL schema extraction". The first few pages of results were searched for fitting works. For every discovered work, its references and citations were also checked for fitting candidates. The works presented here are therefore those that closely matched this thesis' problem domain. Furthermore, these approaches were popular enough to appear as relevant results in Google Scholar, influenced or influenced by such papers.

Follows an overview of the discovered solutions. For each one, its approach and solution to the problem are described, alongside notes on how it will resolve the goal of this thesis. In the end, a summary of the described works is presented.

### 2.1 Enabling ad-hoc reuse of private data repositories through schema extraction

This work introduces an automated approach for extracting schemata from RDF data sources. The key focus is on schema extraction preserving privacy for sensitive information data stores, such as healthcare data. This method relies on RDFS and OWL to derive the schema from ontologies provided within the datasets.

The query shown in Listing 2.1 is introduced. It uses RDFS to discover classes

and properties of the data. It assumes the full inclusion of all used vocabularies in the data store. Reliance on the ontology means any deviation from it will not be detected by the query. It also relies on SPARQL entailment to materialise implicit triples, which is not common among SPARQL endpoints. This means classes and properties may not be properly annotated using `rdfs:Property` and `rdfs:Class` and therefore not discovered. (L. C. Gleim et al. 2020)

```

1   CONSTRUCT {?s ?p ?o}
2   WHERE {
3     {[] ?s []}
4     UNION {[] a ?s} .
5     ?s ?p ?o .
6   }

```

Listing 2.1: SPARQL Query for ontology schema discovery(L. C. Gleim et al. 2020)

Therefore, the following schema extraction query is introduced. This query discovers RDFS classes and properties directly instantiated in the queried dataset without the presence of an ontology based on the data structure:

```

1   CONSTRUCT {
2     ?predicate ?a ?b ;
3     a rdf:Property ;
4     rdfs:domain ?pDomain ;
5     rdfs:range ?pRange .
6     ?concept ?c ?d ;
7     a rdfs:Class .
8   } WHERE {
9     ?s ?predicate ?o .
10    OPTIONAL {?s a ?pDomain}
11    OPTIONAL {?o a ?pRange}
12    OPTIONAL {?predicate ?a ?b}
13    [] a ?concept
14    FILTER(!isBlank(?concept))
15    OPTIONAL {?concept ?c ?d}
16  }

```

Listing 2.2: SPARQL Query for instantiated schema discovery(L. C. Gleim et al. 2020)

This query discovers all classes and properties instantiated in the dataset, which fits the schema discovery requirement.

This approach, however, has two major shortcomings. Firstly, it pays no attention to the numerosity of each class or property and therefore does not reveal the schema completeness. Secondly, this schema extraction approach is not created with execution speed as a priority. The authors cite execution times up to minutes long. However, upon attempting to run this on the `data.gov.cz/sparql` endpoint, it returned the error shown in Listing 2.3.

```

1   Virtuoso 42000 Error The estimated execution time 15047075 (sec)
    exceeds the limit of 80000 (sec).

```

Listing 2.3: Execution error of query 2.2

This approach is likely not to be feasible for a significantly-sized dataset to provide quick feedback to users.

## 2.2 HyperGraphQL

HyperGraphQL provides a GraphQL interface for the defined RDF sources. It uses a configuration file to define RDF services to be wrapped. An annotated GraphQL Schema with a semantic context both define the endpoint schema as well as enables the transformation of the context-less GraphQL queries into SPARQL queries.(Ltd. 2022)

```
1 {
2   "name": "dbpedia-hgql",
3   "schema": "schema1.graphql",
4   "server": {
5     "port": 8081,
6     "graphql": "/graphql",
7     "graphiql": "/graphiql"
8   },
9   "services": [
10    {
11      "id": "dbpedia-sparql",
12      "type": "SPARQLEndpointService",
13      "url": "http://dbpedia.org/sparql/",
14      "graph": "http://dbpedia.org",
15      "user": "",
16      "password": ""
17    }
18  ]
19 }
```

Listing 2.4: HyperGraphQL configuration example(Ltd. 2022)

```
1 type __Context {
2   City:      _@href(iri: "http://dbpedia.org/ontology/City")
3   Country:  _@href(iri: "http://dbpedia.org/ontology/Country")
4   label:    _@href(iri: "http://www.w3.org/2000/01/rdf-schema#
5   label")
6   comment:  _@href(iri: "http://www.w3.org/2000/01/rdf-schema#
7   comment")
8   country:  _@href(iri: "http://dbpedia.org/ontology/country")
9   capital:  _@href(iri: "http://dbpedia.org/ontology/capital")
10 }
11
12 type City @service(id:"dbpedia-sparql") {
13   label:    [String] @service(id:"dbpedia-sparql")
14   country:  Country @service(id:"dbpedia-sparql")
15   comment:  [String] @service(id:"dbpedia-sparql")
16 }
17
18 type Country @service(id:"dbpedia-sparql") {
19   label:    [String] @service(id:"dbpedia-sparql")
20   capital:  City @service(id:"dbpedia-sparql")
21   comment:  [String] @service(id:"dbpedia-sparql")
22 }
```

Listing 2.5: Annotated schema example (Ltd. 2022)

This allows users to send plain GraphQL queries to the instance, which is convenient for developers. The response is formatted in JSON-LD, which preserves the semantic context information while staying comprehensible to those familiar with JSON.

While this appears to be a functional bridge between the linked data world and GraphQL, this solution depends on a manually configured schema with annotation. The creation of that schema relies on a specialist with domain knowledge to both create the schema and maintain it, should the wrapped endpoint be modified.

## 2.3 TopBraid

TopBraid is a commercial Enterprise Data Governance (EDG) solution, one of whose features is querying RDF data sources using GraphQL. It uses SHACL (W3C 2022f) data shape constraints to generate GraphQL schemata. The query results return JSON, discarding the semantic context.

The TopBraid approach is as described: (TopQuadrant 2022)

- Generates GraphQL Schemata from SHACL ontologies.
- Watches for any changes in ontologies and updates generated schemata on the fly.
- Lets users query not only data that is captured by EDG, but also the change the ontology models themselves.

```
1 enterprise:Country
2   a owl:Class ;
3   a sh:NodeShape ;
4   rdfs:label "Country" ;
5   sh:property enterprise:Country-hasBorderWith ;
6   sh:property enterprise:Country-historicNote ;
7   sh:property enterprise:Country-label ;
8   sh:property enterprise:Country-status ;
```

Listing 2.6: Class SHACL constraint (TopQuadrant 2022)

This approach does not require an annotated GraphQL Schema, unlike HyperGraphQL. Thanks to the higher expressiveness of SHACL compared to the annotated GraphQL Schema, the generated schemata are more expressive (Taelman, Sande, and Verborgh 2019). The schema generation also allows for schema introspection. However, the approach relies on the SHACL data constraints, which must be supplied by the data provider. This is a solution for users of the TopBraid EDG software, but it is not suitable for general use on the web.

## 2.4 Stardog

Stardog is another commercial enterprise semantic data solution with support for querying RDF data using GraphQL. The schema for the data is optional and if it is not provided, it is generated automatically from the RDFS/OWL schema or SHACL constraints defined in the database (Union 2022).

The Stardog approach relies on RDFS/OWL classes that are mapped to GraphQL types. Each unique class is mapped to a GraphQL type. Any property whose `rdfs:domain` is set to that class will be added as an additional field to the type. The GraphQL query is then translated into a SPARQL query and executed. The following example illustrates the transformation.

```
1 {
2
3
4
5
6
7
8 }
```

```
Human(id: 1000) {
  name
  knows: friends {
    name
  }
}
```

Listing 2.7: Stardog example GraphQL query (Union 2022)

```
1 SELECT *
2 FROM <tag:stardog:api:context:all>
3 {
4
5
6
7
8
9 }
```

```
?0 rdf:type :Human .
?0 :id "1000"^^xsd:integer .
?0 :name ?1 .
?0 :friends ?2 .
?2 :name ?3 .
```

Listing 2.8: Stardog example generated SPARQL query (Union 2022)

Further, Stardog offers Stardog Explorer, a graph and node visualisation of the data store. It appears to be a force-based node visualisation with searching capabilities. An example from the Stardog website is shown in figure 2.1.



Figure 2.1: Stardog Explorer interface (Union 2022)

## 2.5 GraphQL-LD

GraphQL-LD is an academic approach to querying linked data using GraphQL, developed by Ruben Taelman et al (Taelman, Sande, and Verborgh n.d.). This method uses JSON-LD context alongside the GraphQL queries to transform them into SPARQL queries. This reduces the expressiveness of the available queries

from the SPARQL baseline. Unlike HyperGraphQL, this approach does not require an intermediate service to be set up. The SPARQL results are then transformed from the SPARQL graph structure into a hierarchical tree structure and returned as an ordinary JSON response.

The need for a domain expert to provide the JSON-LD context in order for developers to create GraphQL queries means that it is not suitable for a developer looking to approach an unknown SPARQL endpoint, nor does it provide any insights into the schema.

HyperGraphQL and GraphQL-LD are extensively compared in a construction context in (Werbrouck et al. 2019), where some shortcomings of both approaches are addressed. A criticism of HyperGraphQL is the inability to query for a subtype. The provided example describes a type `Space` with a subtype `Kitchen`. A relation of `hasSpace` would fail to retrieve instances of `Kitchen` from the object. (Werbrouck et al. 2019)

In contrast, with GraphQL-LD’s direct translation into SPARQL, more of SPARQL’s behaviour is preserved at the cost of the absence of the extracted schema and introspection. The work commends GraphQL-LD’s serverless flexibility without the need to set up an intermediary, as opposed to HyperGraphQL’s more rigid nature.

## 2.6 UltraGraphQL

UltraGraphQL is an academic approach presented in (L. Gleim et al. 2020) and based on the authors’ previous work in (L. C. Gleim et al. 2020). It focuses on providing a GraphQL interface to RDF data without the need for any Semantic Web-related knowledge. Extends HyperGraphQL and aims to resolve its shortcomings by providing automatic schema extraction and mutation support.

First, an RDF schema is extracted by mapping the RDF types and properties to UltraGraphQL object types. The schema extraction process is described by authors in (L. C. Gleim et al. 2020) and summarised in this work in 2.1. The extracted schema is converted to a GraphQL Schema by mapping classes to objects and RDF properties to fields of those types. This process is to allow for GraphQL introspection. The mapping process relies on GraphQL Interfaces to represent RDF subclassing and other relationships feasible in RDF that are not trivially translatable to GraphQL Schema. Nevertheless, the mapping will not be perfect due to RDF’s increased expressiveness.

Based on the generated schema an adapter instance is created. This adapter handles the outgoing GraphQL queries. It transforms the queries it receives into SPARQL queries, as described in (L. Gleim et al. 2022). Finally, the results from SPARQL are merged into one, enriched with context and returned by the adapter.

## 2.7 ViziQuer

ViziQuer (Zviedris and Barzdins 2011) SPARQL endpoint exploration and visualisation tool. It differs from others as it uses a UML visualisation. Figure 2.2



contains the authors’ example of the visualisation. ViziQuer offers an option to export selected classes as a SPARQL query that selects the desired classes.

The tool appears to have limitations in its scalability. The authors suggest against using it to visualise complex endpoints, such as DBpedia. This may be due to technical limitations of the time (2011) or an inappropriate choice of queries.

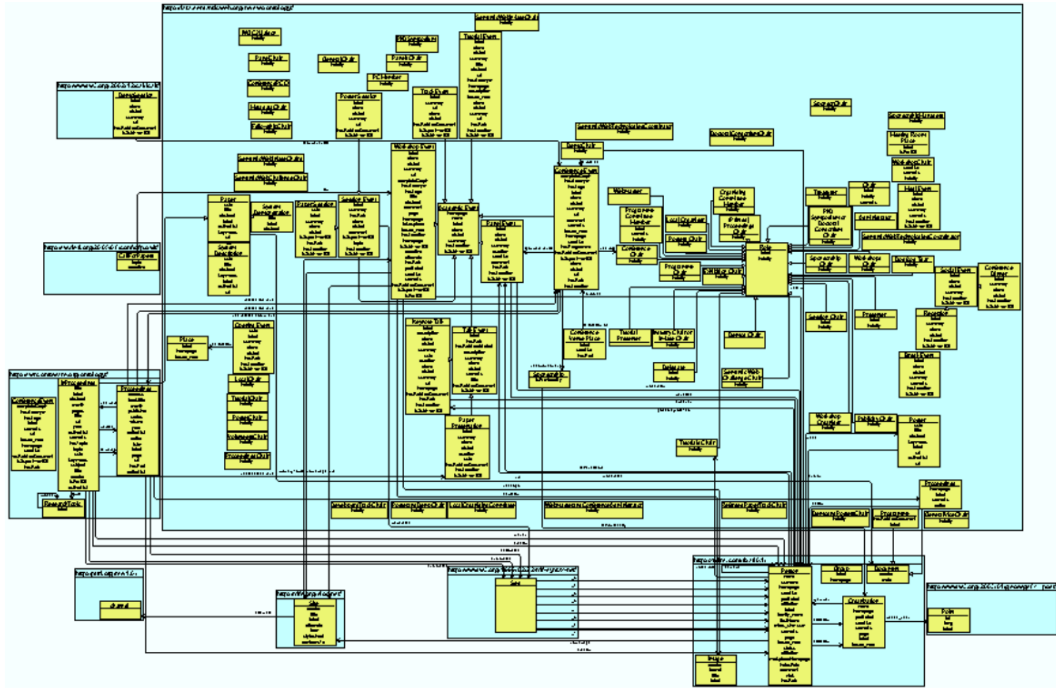


Figure 2.2: ViziQuer endpoint visualisation (Zviedris and Barzdins 2011)

## 2.8 LODSight

This work (Dudáš, Svátek, and Mynarz 2016) introduces a Java application for summarising and visualising the relationships and structure in an RDF dataset. Summarisation is based on [type1 - property - type2], as well as [type - data property - datatype] paths that are extracted from object instances in the dataset. The results are stored in an SQL database and visualised using D3.js. The query used to extract the schema is shown in Listing 2.9.

```

1 SELECT ?a ?p ?b (COUNT(*) AS ?count)
2 WHERE { [ a ?a ] ?p [ a ?b ] . }
3 GROUP BY ?a ?p ?b

```

Listing 2.9: Path summarisation SPARQL query (Dudáš, Svátek, and Mynarz 2016)

As of the writing of this thesis, the links leading to the application appear to no longer function. To use it users are required to run their own instance. This obstacle unfortunately makes it less accessible and less likely to be used by users.

Regarding performance, the authors write that ”processing several hundred triples takes less than 20s. For large datasets, implementing incremental exploration (...) might be an option.”, suggesting that an incremental solution would

be beneficial. The authors also note the severe unreliability of their solution, citing a 53% success rate in preliminary evaluation using several endpoints. This is likely due to the complexity of the pathfinding query and the SPARQL endpoints not managing to compute it or timing out. In exploratory testing, running the query occasionally succeeds but endpoints do not answer reliably enough to use this solution.

It is therefore apparent that to more reliably perform schema extraction from foreign endpoints, a series of exploratory queries is necessary, rather than a single monolithic query. This is corroborated by other approaches, such as in 2.10.

## 2.9 A Visual Aide for Understanding Endpoint Data

This approach (Florenzano et al. 2016) aims to improve the previously presented LODSight (Dudáš, Svátek, and Mynarz 2016) by providing a more dynamic system with improved information about the graph as a whole, as well as more detailed information about each class and relation. Visually, it brings a more modern that appears to be based on a force-directed layout. This visualisation provides the ability to divide nodes of super-types into sub-type nodes, such as dividing a `Person` into `actor`, `director`, `editor` and others.

Performance-wise, this system also sustains the problems that LODSight experienced. The authors themselves note that "computing the necessary files for our visualisation may take several minutes or even hours, so it is not possible to offer on-demand visualisations."(Florenzano et al. 2016). Unfortunately, as of writing this thesis, the application doesn't seem to be available.

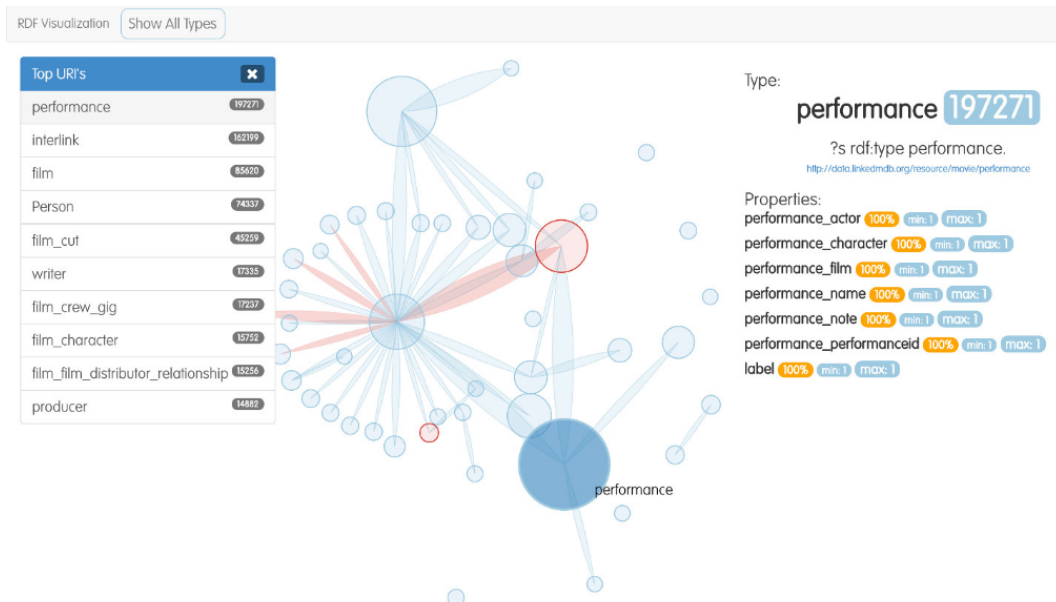


Figure 2.3: A Visual Aide for Understanding Endpoint Data(Florenzano et al. 2016)

```

1 SELECT DISTINCT ?class (COUNT( ?instance ) AS ?instanceCount )
2 WHERE {
3     ?instance a ?class .
4 }
5 GROUP BY ?class
6 ORDER BY DESC( ?instanceCount )
7 LIMIT 10 OFFSET 0

```

Listing 2.10: SPARQL query retrieving the 10 classes having the most instance (Weise, Lohmann, and Haag 2016)

```

1 SELECT (COUNT( ?val ) AS ?valCount ) ? valType
2 WHERE {
3     ?instance a <http://dbpedia.org/ontology/Agent> .
4     ?instance ?prop ?val .
5     BIND(DATATYPE( ?val ) AS ?valType ) .
6 }
7 GROUP BY ?valType
8 ORDER BY DESC( ?valCount )
9 LIMIT 10

```

Listing 2.11: SPARQL query retrieving the 10 datatypes most often linked to the DBpedia class Agent (Weise, Lohmann, and Haag 2016)

## 2.10 LD-VOWL

LD-VOWL is an online<sup>1</sup> academic tool that extracts and visualises schema information from Linked Data endpoints, based on a number of SPARQL queries (Weise, Lohmann, and Haag 2016). It employs a class-centric approach by querying for classes in the dataset first. To save execution time and complexity, the schema extraction is split into multiple (up to hundreds of) queries and the visualisation is built iteratively, as results are retrieved from the endpoint. This improves latency issues experienced by other approaches, which focus on more complex and complete SPARQL queries. This is a user-friendly approach, as it provides information as soon as it is available, unlike some solutions which perform a complete extraction before showing anything. This approach is fitting for our outlined goals.

LD-VOWL is a more dynamic and easier accessible tool than the other visualisation tools. It, however, lacks some features that are satisfied in (Florenzano et al. 2016), such as attribute numerosity. The graph visualisation can be overwhelming when an endpoint is visualised, as is shown in Figure 2.4. The number of objects on the screen quickly makes the visualisation over-saturated and confusing. This is something 2.8 attempted to resolve. The visualisation employs a force-directed layout. As the queries resolve, the choice of layout causes the visualisation to move around rapidly as new edges are created. Perhaps a more static version would be less jarring.

<sup>1</sup><http://vowl.visualdataweb.org/ldvowl/>



The initial observation stage uses several `Observer` classes that each collect `Observations` about the endpoint. The collection of these observations is a lengthy process, up to tens of minutes according to the documentation. Observations are recorded in a Turtle format file and are constructed by the `Observers` using SPARQL `CONSTRUCT` queries, like the example in Listing 2.12. The listed query creates an instance of `AttributeObservation` for every literal connected to the class instances.

Although this would be a correct way of extracting the complete schema from an endpoint, it may take multiple hours on a large endpoint. Furthermore, for an endpoint like `dbpedia` where instance counts are in the millions, this will generate tens of gigabytes of data. Given the focus on speed and ease of use, this method of schema extraction is unsuitable for our described use case.

The parsing process uses the collected observations to create JavaScript objects – `Descriptors`, which describe the classes. These objects are then transformed into a GraphQL Schema in the final phase. The schema creation creates GraphQL objects with fields based on the original literal attribute types. Interestingly, only numbers and language strings are considered, everything else is mapped as a plain string.

```
1 CONSTRUCT {
2   []
3     a se:AttributeObservation ;
4     se:describedAttribute <${propertyIri}> ;
5     se:attributeSourceClass <${classIri}> ;
6     se:targetLiteral ?targetLiteral .
7 } WHERE {
8   {
9     SELECT ?targetLiteral
10    WHERE {
11      GRAPH ?g {
12        ?instance
13          a <${classIri}> ;
14          <${propertyIri}> ?targetLiteral .
15        FILTER isLiteral(?targetLiteral)
16      }
17    }
18  }
19 }
```

Listing 2.12: SPARQLLess attribute observation query (*SPARQLLess* 2023)

## 2.13 Simplod

`Simplod` (*Simplod - Visualization tool for simplifying access to Linked Data.* 2023) is a React application providing simple access to SPARQL data. It works by first visualising the endpoint’s schema. The schema extraction does not seem to be considered. Instead, the service relies on downloading a schema file from a provided URL and creating a visualisation based on that. This is unfortunately not suitable for unknown endpoints, as it does not resolve the issue of schema extraction.

The area where `Simplod` shines is its ability to query the visualised data schema by generating SPARQL queries based on the user’s selection. The example

in Figure 2.6 shows the interface of Simplod. It provides users with a good search and filtration system for data and object properties of each class. Once selected, the selection is transformed into a SPARQL query, which can be run on the endpoint. The advantage of this is that the output is platform-agnostic. Any developer with access to a SPARQL library can make use of this service once the query is generated. As opposed to other solutions, this does not require an intermediate service to be run, simplifying its use. A slight complaint could be that the SPARQL query does not come with typing support in statically typed languages, but it makes up for it in its universality.

Simplod would be a good solution if it provided a schema extraction. If a schema extraction tool could be used to generate the schema Simplod requires, both would complement each other well.

## 2.14 Summary

The presented methods all aim to resolve a related problem, each by its own implementation. Yet each of them focuses on different aspects of the problem, leaving room for improvement in achieving the goal that was laid out for this thesis. The presented approaches are summarised in Table 2.1. The following paragraphs explain the comparison points.

**Availability** tracks whether or not the tool can be accessed immediately online (as of the writing of this thesis). A check mark (✓) denotes that the tool is available and usable immediately on the web. One cross (✗) means that the source code or the application can be downloaded, but must be executed (and potentially compiled) locally by the user. A double cross (✗✗) means that neither the application nor the source code could be found. That is caused by either the links leading to non-functioning websites, or missing altogether.

**Incremental Extraction** shows the rate at which the schema is extracted. Note that this is not the speed of the entire operation, but rather the time it takes for the user to receive feedback. A check mark (✓) shows that the service works by gradually exploring the endpoint and providing some feedback within seconds. A cross (✗) says that the method runs a background query that could take multiple minutes, up to hours. The long query provides no feedback until the end when everything is extracted.

**Output** describes the output format of the schema extraction tools. Some offer only a visual representation of the endpoint. In such cases, the type of visualisation used is noted (Force/Flow/UML). Other tools output a schema definition that is programmatically usable (TypeScript, GraphQL).

**No setup** analyses whether or not the service can be used without any setup. This means that the service should be executable without any prior knowledge about the endpoint, the problem domain or even the concept of linked data. A need for source code compilation is not considered here. Tools marked with a

cross (**X**) require the user to provide a semantic context, an ontology, a mapping or another form of additional information before providing the schema.

Name	Available online	Increm. extraction	Output	No setup
ViziQuer	<b>X</b>	<b>X</b>	UML, SPARQL	✓
LD-VOWL	✓	✓	Force	✓
VizLOD	<b>XX</b>	✓	Force	✓
Visual Aide	<b>XX</b>	✓	Force	✓
LODSight	<b>X</b>	<b>X</b>	Force	✓
HyperGQL	✓ <sup>1</sup>	<b>X</b>	GraphQL	<b>X</b>
TopBraid	✓ <sup>1</sup>	<b>X</b>	GraphQL	<b>X</b>
Stardog	✓	<b>X</b>	GraphQL, Force	<b>X</b>
UltraGQL	<b>X</b>	<b>X</b>	GraphQL	<b>X</b>
GraphQL-LD	<b>X</b>	<b>X</b>	GraphQL	<b>X</b>
SPARQLess	<b>X</b>	<b>X</b>	GraphQL	<b>X</b>
Simplod	✓	<b>X</b>	Flow, SPARQL	<b>X</b>
TypeSPARQ	✓	✓	Flow, TypeScript	✓

Table 2.1: Overview of existing schema extraction solutions

There are two main groups of approaches that were explored. One group of tools prioritises schema extraction. They are capable of extracting the schema and presenting it to the user, typically via a visualisation. They tend to not be concerned with how that schema is used further. Furthermore, the visualisation can end up being crowded and chaotic, especially if they rely heavily on force layouts.

Another group focuses mostly on making endpoints accessible through an API. They tend to require prior setup in the form of providing a schema, context or shape constraints. Subsequently, an access point is created for easy access to the data through means other than SPARQL. The need for prior setup or domain knowledge makes these inaccessible to casual users. However, the access they provide is very valuable for developers in accessing the data. It would be preferable if this approach could be adapted for use in unknown endpoints. This group also tends to perform a complete schema extraction first, taking minutes or up to hours before it is accessible.

TypeSPARQ – the solution presented in this thesis aims to combine the advantages of these two groups into one easy-to-use tool. It enables the exploration and visualisation of an unknown endpoint in a way that eliminates excessive force-related movement. Attention is paid to ensuring that the extraction is fast and gradual, minimising individual response times.

TypeSPARQ’s exploration is combined with API schema export functionality. It provides the users with the option to export the schema and use it to directly access the data through an API. TypeSPARQ is designed so that this functionality is extensible, should a different API be desired. With this approach, TypeSPARQ takes a middle ground, satisfying both needs.

<sup>1</sup>Paid commercial solutions

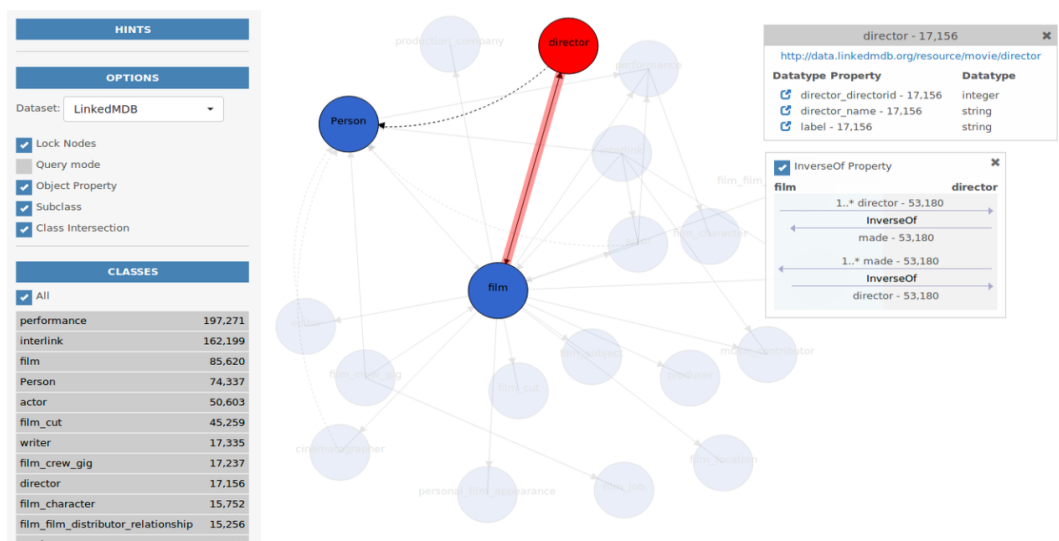


Figure 2.5: VizLod visualisation interface (Anutariya and Dangol 2018)

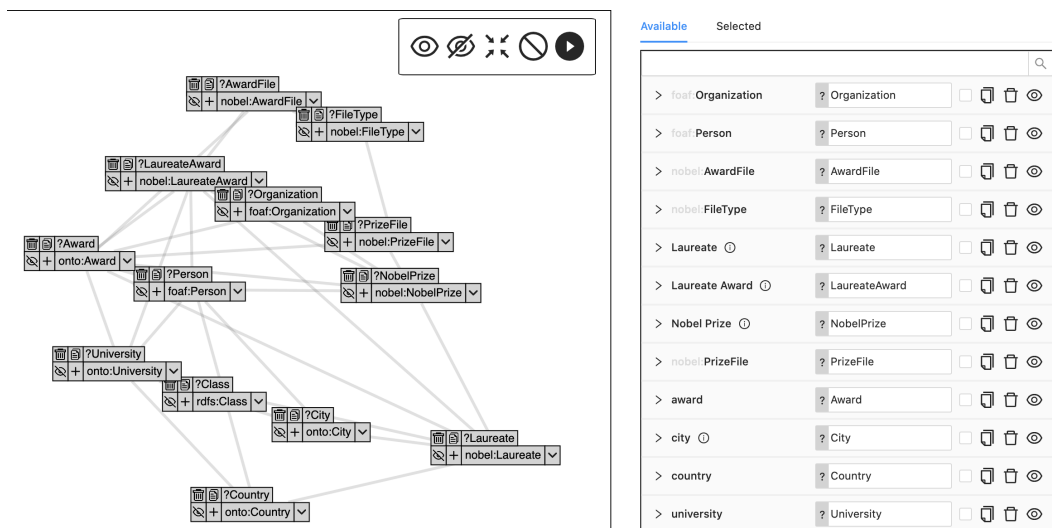


Figure 2.6: Simlod visualisation interface (*Simlod - Visualization tool for simplifying access to Linked Data.* 2023)



# 3. Requirements

This chapter defines the requirements for TypeSPARQ to fulfil. These will ensure that the solution solves the given problem, as well as avoids the shortcomings of other solutions described in chapter 2. The requirement definitions presented here will govern further development choices. Concrete implementation decisions based on these requirements are discussed in section 3.3.

## 3.1 Non-Functional Requirements

As was outlined in section 1.1, the aim is to provide easy access to SPARQL endpoints for people with or without prior SPARQL knowledge. Therefore the following requirement<sup>1</sup> arises.

R1: The application SHOULD be easily accessible to a majority of users, even those without SPARQL knowledge.

Some solutions required the endpoint provider to run the schema extraction or provide metadata or ontologies alongside their service. Reliance on this external support is unfeasible. It makes sense to only consider approaches that perform exploratory queries on unknown endpoints.

Prior setup or technical knowledge requirements should be avoided at all costs. Having to download source code from a code repository to access the service may be more reliable, but adds a level of complexity to the process. That could deter its use with more casual users.

R2: The application SHOULD require minimal setup.

Source code compilation is a hurdle that should be preferably avoided. Issues such as having an incompatible compiler version would diminish *R2* compliance. For this reason, it would be preferable to provide the application as either an executable out of the box or a web application or service.

## 3.2 Functional Requirements

Considering the prior analysis of currently and previously available solutions, functional requirements will be defined. These will serve as a guide of what functional features will be implemented. Primarily, the service should provide a graphical view of the endpoint.

R3: The schema SHALL be revealed by displaying RDF classes and their relationships to the user.

To cater towards users without SPARQL knowledge, providing access to the endpoint in another format will improve the tool's utility, as well as support R1. A fitting export medium would be a programmatic schema or API of the

---

<sup>1</sup>Requirements are presented in accordance with RFC2119

endpoint. Providing developers without LD knowledge access to SPARQL data stores makes the data more accessible. Creating new consumers of this data would certainly increase the appeal for other publishers. To facilitate this, access in a modern programming language is appropriate.

R4: The extracted schema SHALL be exportable in a non-visual programmatic format.

Given the analysis of various schema extraction methods and the end-user orientation of this solution, a long latency should be avoided. Explored solutions such as queries presented in (Dudáš, Svátek, and Mynarz 2016) (L. C. Gleim et al. 2020) or (*SPARQLess* 2023) presented a major computational cost in extracting using a single query. Having a singular extraction step makes the application appear frozen and unresponsive while the query is being executed. Furthermore, as noted in (Dudáš, Svátek, and Mynarz 2016), the schema extraction is not reliable due to endpoint throttling and timeouts. Therefore, the following requirement emerges:

R5: Schema extraction SHOULD provide feedback as quickly as possible. Extraction SHALL follow an iterative process and display information as soon as it is available.

To resolve an oversight of (Weise, Lohmann, and Haag 2016), wherein only the top 9 classes are displayed, it would be beneficial to include an option to query more than the initial number of classes. Iterative querying will lead to a subset of classes from the dataset being displayed at first, followed by more on request to capture as much of the data as possible.

R6: The application SHALL enable the user to request additional classes beyond those initially queried.

One might wonder why a limitation to the number of classes is used at all. The reason is R5 compliance. Should all classes of the dataset be queried at once, the number of possible edges grows exponentially. In waiting for those queries, other queries, such as attributes could be delayed. It can be expected that most users will be interested in the most common information about the endpoint. However, if a user consciously wants the entire endpoint, they can simply request these additional classes.

### 3.3 Application design

Resolution of some requirements necessitates the making of more informed decisions. The solutions for compliance with some requirements and their implementation are outlined in this section.

#### 3.3.1 Non-functional requirements (*R1*, *R2*)

It is important to decide whether the application will be developed as a native application or a web-based app. To avoid environment and interpreter compatibility issues, having to run an executable locally should be avoided. Further,

maintaining software versions for multiple operating systems (or mobile devices) adds an additional level of complexity to the development process.

Therefore, the most appropriate approach would be a web-based application. Designing the application to perform all functionality client-side would mean that no server is required to run. Hosting a static site reduces hosting expenses, ensuring the service stays available longer. Should the hosting of the website fail in the future, source code will also be provided and can be run locally.

### 3.3.2 Schema visualisation (*R3*)

Schema visualisation is the main feature of some of the existing solutions. However, a common approach is using a force-directed layout. Whilst this can seem natural to account for an iteratively changing visualisation, the result is that nodes and edges move around erratically during query execution. Another option, implemented by 2.7 used UML. The familiarity of that style could be helpful, but displaying all existing properties at once could clutter the visualisation. Hundreds of attributes would be better displayed in a more advanced interface, perhaps including searching and filtering.

To avoid force layouts' problems and to experiment with a different approach, the Flow library will be used. It is described further in section 5.2.3. A more static visualisation will allow users to arrange the endpoint to their liking. A layout is nevertheless important, so some initial placement support should be included. As can be seen in Figure 2.4, some further steps should be taken to reduce visual clutter. Aggregating edges and attributes behind dialogues may be required to avoid a similar situation.

### 3.3.3 Schema extraction (*R4*)

A fitting realisation of export will be employing the LDKit library. LDKit is a TypeScript library that handles querying a SPARQL endpoint and is further described in chapter 4. However, it requires a schema definition, describing the endpoint. Enhancing this existing solution with a schema generation tool would adequately provide access to an unknown endpoint. Thanks to TypeSPARQ's schema extraction, the access is not gated by a need to manually explore the endpoint, nor prior SPARQL or RDF experience.

The goal of TypeSPARQ is not to present itself as a service that facilitates these queries, as is the case with many of the GraphQL adapter solutions. The goal is rather to generate an output that can be used independently of TypeSPARQ. Any TypeScript developer can use TypeSPARQ to access SPARQL endpoints programmatically using only LDKit, once the schema is generated. This should make developing applications over SPARQL endpoints easier for developers.

### 3.3.4 Querying rate (*R5*)

Multiple queries will be issued to display information to the user as soon as possible and to avoid long query evaluation times. To realise this, queries should be rather short and simple. This will have the positive side effect of helping

avoid endpoint timeouts due to long query times. The overall extraction will likely be slightly slower due to request overhead, but the application will provide information and be usable much faster.

## 4. External integration

TypeSPARQ is integrated with external tools to improve both their and its own utility. The reason for this is that they provide the functionality required for TypeSPARQ to accomplish its goals without requiring the re-implementation of existing solutions. The application is designed in a way that makes it easy to integrate even further tools by extending the extension points outlined in 5.12. This section outlines the options for external integration.

### 4.1 LDKit

TypeSPARQ relies on LDKit, an external open-source (*LDKit Linked Data query toolkit for TypeScript developers 2023*) library, to provide TypeScript access to the extracted schema. LDKit is a tool created at the Faculty of Software Engineering at MFF UK. It uses a data schema definition, such as the example in Listing 4.1 to create a data lens into a SPARQL endpoint. The endpoint can then be queried through the LDKit API to fetch entities from the schema. The fetched objects are typed using TypeScript according to the schema.

```
1 import { dbo, rdfs, xsd } from "ldkit/namespaces";
2
3 const PersonSchema = {
4   "@type": dbo.Person,
5   name: rdfs.label,
6   abstract: dbo.abstract,
7   birthDate: {
8     "@id": dbo.birthDate,
9     "@type": xsd.date,
10  },
11 } as const;
```

Listing 4.1: Example LDKit schema definition (*LDKit Linked Data query toolkit for TypeScript developers 2023*)

### 4.2 Schema Import

To allow for interoperability of TypeSPARQ with other schema extraction approaches, it is necessary to allow a schema to be imported. A schema import will replace TypeSPARQ's built-in endpoint querying with user input. This will allow users to perform more detailed extractions that would take too long to be run in-browser.

To import data into TypeSPARQ, an Import modal dialogue is created, accessed via the "Import" button. The technical details of the process are discussed in 5.5.4. To import data, supply a JSON document compliant with the JSON Type Definition (JTD) shown in Listing 4.2. This input will be parsed and, if valid, seeded into the application.

The schema is structured as follows. The `nodes` array contains the schema's classes and the `edges` contains the predicates. The `id` `id` property is the IRI of the class or property. For edges, `source` or `target` are subject and object IRIs. A

```

1 {
2   properties: {
3     endpoint: { type: 'string' },
4     nodes: {
5       elements: {
6         properties: {
7           id: { type: 'string' },
8           instanceCount: { type: 'uint32' },
9         },
10        optionalProperties: {
11          attributes: {
12            elements: {
13              properties: {
14                id: { type: 'string' },
15                type: { type: 'string' },
16                instanceCount: { type: 'uint32' },
17              },
18            },
19          },
20        },
21      },
22    },
23    edges: {
24      elements: {
25        properties: {
26          source: { type: 'string' },
27          target: { type: 'string' },
28          id: { type: 'string' },
29          instanceCount: { type: 'uint32' },
30        },
31      },
32    },
33  },
34 }

```

Listing 4.2: TypeSPARQ import data JSON Type Definition

class can contain attributes, which are literal values related to instances of the class. Attributes must contain a type definition in the form of an IRI.

When creating the import data, be aware of the correctness of the input. Whilst the schema is validated and errors are reported, semantically incorrect data may be handled silently. Issues such as providing nodes with non-unique ids or edges connecting non-existent classes may not report errors.

### 4.3 Schema Export

If desired, TypeSPARQ can be used as a schema extraction tool for you to build other tools and applications on top of. TypeSPARQ offers an extensible set of export options that should allow you to make use of its schema extraction capabilities for whichever purpose. Either make use of one of the provided export options to import schemata into your applications, or extend TypeSPARQ with the format you need (see Section 5.12 for extending TypeSPARQ).

# 5. Developer documentation

This chapter serves as the technical documentation for the source code, which is available at <https://github.com/Jkuzz/sparql-explorer>.

## 5.1 Technologies and Frameworks

This section explains the frameworks, technologies and tools that will be used in the implementation. Smaller libraries and other dependencies will be described in 5.2. A summary of each technology and the reason for its inclusion will be outlined. For a more complex explanation, please refer to the linked documentation.

### 5.1.1 Single Page Application

A Single Page Application (SPA) is a method of implementing a web page that does not use traditional HTTP redirects to move users between pages. Rather, the application is a single document, whose content is dynamically changed. This eliminates the need for a page reload on every route, speeding up the user experience. The experience can also be smoother, as more control is in the hands of the application, allowing for requests to run in the background. The user can then use other features of the application, not having to wait for a page to load.

This approach comes with some drawbacks. Primarily, Search Engine Optimisation (SEO) is negatively affected. This is due to the way that engine crawlers index the site's contents. Without proper support, some frameworks (such as Vue) initially serve an almost blank HTML page, which is then populated dynamically using JavaScript. Since web crawlers do not tend to run this code, the page appears empty. There are ways to mitigate this, such as using Server Side Rendering (SSR). However, since we wish to remain serverless, SSR is not used. Another option if loading times are a concern can be code splitting. Code splitting is used for this purpose in the project, mainly to separate the router imports.

### 5.1.2 Technologies used

1. **HTML** HyperText Markup Language is a web standard describing the document which serves as the structure and content of a web page. The tree structure of the document defines elements using HTML tags, each with its own purpose and functionality. The document tree is also referred to as the Document Object Model (DOM), a concept describing the hierarchical structure of the document. (*HTML Living Standard 2022*)
2. **CSS** Cascading Style Sheets is a language used to define the appearance of a DOM document, primarily used for HTML in web applications. CSS is applied to the HTML document by the browser during rendering. Whilst pure CSS is not used (see 5.2.1), its paradigms and implementation are heavily present in order to style the application.

3. **JavaScript** (JS) is the de-facto standard programming language for web apps, as it is used by over 98% of all websites (*Usage statistics of JavaScript as client-side programming language on websites* 2022). JS is an implementation of the ECMAScript standard by ECMA International. It is a language primarily created for use in browsers to run client-side code in browsers. Modern browsers include an ECMAScript-compliant compiler and engine that runs JS code downloaded alongside HTML and CSS from a website. (International 2022).
4. **Git** is an open source version control software for software development (*Git Version Control System* 2022). It is used to track files and their changes. The remote repository for this project is github.com and the code is available <https://github.com/Jkuzz/sparql-explorer>.
5. **JSON** JavaScript Object Notation is a lightweight data-interchange format that is language-independent and based on a subset of ECMAScript. (*Introducing JSON* 2022) It will be used to communicate with SPARQL servers. SPARQL schema extraction query requests will be answered by endpoints using JSON, which is then easily processed in JS.

Provided JavaScript is the primary development language, the vast suite of JS libraries and frameworks is available. The following will be used in the application. The next sections assume a basic understanding of the above-described technologies.

### 5.1.3 NPM

Node Package Manager (NPM)<sup>1</sup> is a dependency manager for Node. It tracks the declared dependencies, including the desired version and installs them from the npm registry. The npm registry is a public database containing freely available user-submitted software packages. The majority of dependencies and tools listed in this section are installed using npm and can be found in the project's `package.json` – the file where npm reads dependencies.

Npm supports two kinds of dependencies: regular and devDependencies. Regular dependencies are used at runtime and therefore are required to be present in the final bundle. This includes libraries and frameworks, although sometimes only parts that are actually used are included. The devDependencies are usually tools that are used during development but are not included in the final application. This includes tools such as formatters, linters, and bundlers. Also included are type definitions for typescript packages, as no typescript is exported (see 5.1.4).

Npm also allows the declaration of scripts, which can execute the installed packages or other system scripts. Scripts are used to execute lifecycle stages or other utilities. Listing 5.1 shows an extract of script definitions from the project.

```
1  "scripts": {  
2    "build": "run-p type-check build-only",  
3    "build-only": "vite build",  
4    "type-check": "vue-tsc --noEmit -p tsconfig.vitest.json --  
    composite false",
```

---

<sup>1</sup><https://www.npmjs.com/>



```
5   "format": "prettier . --write"  
6 },
```

Listing 5.1: package.json script definitions

### 5.1.4 TypeScript

Typescript is a programming language that evolved from JavaScript. The need for a better development experience and support inspired the creation of Typescript. It was developed by Microsoft and released as open source in 2012 (*Announcing TypeScript 1.0* 2023). The language provides static support for type definitions and type checking among other features.

Typescript code must be "transpiled" into JavaScript, which is performed by the Typescript compiler. This compilation step performs the before-mentioned type checking and other optimisations. The resulting code is served to the browser as a normal JS file. This step is often performed by a bundler, in our case, Vite.

TypeScript's static typing and improved tooling support have been shown to improve developer productivity over JavaScript (Fischer and Hanenberg 2015). The majority of libraries used within this project provide TypeScript type definitions of their APIs. This implementation will also utilise this type of safety for all its APIs in order to ease current and future development. Furthermore, one of the output methods will be the LDKit schema definition. LDKit itself is a type-safe access API for SPARQL endpoints, as was discussed previously. Providing type-safe access to endpoints should therefore provide similar benefits for developers.

### 5.1.5 Vite

Vite is a modern JS bundler. (*Vite* 2023) A bundler is a developer tool that takes the source files and modules and turns them into a native JS, HTML and CSS bundle. This can involve multiple steps, including but not limited to:

- Packing dependencies
- Typescript transpiling
- CSS preprocessing from SCSS, Sass, others
- Minifying JavaScript, compressing resources

and many more. Bundlers are typically configurable with plugins to support whatever steps are required.

The disadvantage of this approach is felt during development. As the application becomes larger, it can take much longer to pack. If the developer wants to see their changes, waiting up to a few minutes is detrimental to productivity. Vite resolves this by implementing Hot Module Replacement (HMR). Instead of invalidating the entire bundle on change and re-bundling, Vite is able to only invalidate the changed files and dynamically replace the changed modules in the application, which removes a lot of time overhead from the bundling. This approach also scales much better, as each rebuild tends to be of a constant size, rather than the entire application.

### 5.1.6 Vue.JS

Vue<sup>2</sup> is a JavaScript framework for building user interfaces. Vue's syntax extends HTML by defining new tags and attributes, which the framework interprets and implements. Vue uses a single-file component (SFC) structure. That means that every component is contained within its individual file. These components are reusable and mountable within other components. Each SFC defines its own HTML, JS and CSS, which are bundled and included by the framework wherever the component is used.

Vue's reactivity model simplifies the flow of data between the state and the DOM. The framework automatically invalidates and redraws any DOM elements that have been set up to model a variable. Listing 5.2 shows a simple button component that increments the `count` variable on click and shows its current value. Vue uses enhanced HTML to provide efficient declarations of common web and programming paradigms. Those include functions called on user actions, event emitting and receiving, conditional rendering and many more. Another example in 5.2 shows a simple data-driven SFC.

```
1 <button @click="count++">
2   Count is: {{ count }}
3 </button>
```

Listing 5.2: Vue reactivity example (*VueJS* 2023)

```
1 <template>
2   <div
3     v-for="(person, i) in people"
4     :key="i"
5   >
6     User: {{ person }}
7   </div>
8 </template>
9
10 <script setup lang="ts">
11 const people = ['Alpha', 'Bravo', 'Charlie', 'Delta']
12 </script>
```

Listing 5.3: Vue data-driven component

## 5.2 Dependencies

This section will describe the libraries and other dependencies used in the project. The dependencies are managed by NPM and can be seen in the `package.json` file.

### 5.2.1 Tailwind CSS

To improve application accessibility, responsive design is favoured in web development, as it provides improved accessibility and user experience (Almeida and Monteiro 2017). A framework providing a streamlined and easily implementable responsive design is Tailwind<sup>3</sup>. It will facilitate the layout of the application in a simple but sufficient manner.

---

<sup>2</sup><https://vuejs.org/>

<sup>3</sup><https://tailwindcss.com/>

Tailwind eliminates the need to write custom CSS by providing a set of utility classes, which can be directly used in the page's HTML. This pattern makes the HTML more readable by directly containing the style, so developers can find it easier to reason how each of the tags look and what their role is.

Tailwind also employs a bundling step, performed by a bundler plugin. In order to not ship all of its classes, Tailwind only adds the classes that are used to the final bundle. This is a major decrease in bundle size, as no duplicate or unused CSS is shipped.

### 5.2.2 Pinia

Pinia<sup>4</sup> is Vue's official state management solution. A state storage is essentially a managed global data store. It primarily handles data that is handled and modelled by multiple components, serving as a single source of truth for the application. A store is used to avoid passing data as properties up and down multiple levels in the dependency tree. Using these global stores with defined access points gives better control and untangles the application dependency tree.

The stores are modular and multiple can be used. The implementation uses several stores, described in detail in 5.8.

### 5.2.3 Vue Flow

Vue flow<sup>5</sup> is the Vue port of React Flow<sup>6</sup>. It is a component library providing the Flow component. The Flow component is used in the application to display the schema of the endpoint to the user. It allows the nodes to be rearranged by dragging, providing an important user experience factor.

Flow enables the developer to provide custom components for the visualisation, which is the case here. The nodes and their behaviour (besides dragging) were created within the `CustomNode.vue` component.

Flow enables data modelling, which is a regular feature of Vue components. The component is handed some data and when it is modified, Vue handles notifying the component, which then redraws itself to reflect the change. This feature is used alongside a Pinia store. The store's contents are filled as the schema extraction requests resolve. whenever the store's contents are changed, the component automatically changes to account for that. This is performed entirely using Vue's reactivity paradigms, requiring almost no further implementation.

### 5.2.4 Prism

"Prism is a lightweight, extensible syntax highlighter"<sup>7</sup> Prism is used to syntax highlight the generated LDKit code export. The exported code is valid JS and Prism makes it easier to read for the user. This is important for the user experience since developers are used to syntax highlighting, as it is provided by most modern IDEs.

---

<sup>4</sup><https://pinia.vuejs.org/>

<sup>5</sup><https://vueflow.dev/>

<sup>6</sup><https://reactflow.dev/>

<sup>7</sup><https://prismjs.com/>

### 5.2.5 Prettier

Prettier<sup>8</sup> is a code formatter, meaning it reformats code according to its set of rules without affecting the functionality. Usually, prettier is used only as a developer tool, to ensure consistency and readability in the source code. Since one of the outputs of this application is the LDKit schema definition, the formatter is run dynamically on the output text as well. This is for much the same reasons as the primary use, since the intention is that the generated code will be copied and used.

### 5.2.6 Zod

Zod<sup>9</sup> is a schema validation library. It provides parser definitions that parse and validate inputs against a defined schema. The incoming data are validated and inserted into a Zod-defined wrapper type, which provides complete TypeScript typing, validated by the parser.

Zod is used to parse the responses from SPARQL endpoints, ensuring their shape is as expected. Therefore, no other abstraction or preprocessing happens. All further computing can run on these well-known and well-typed objects. This ensures that validation and internal use of types are consistent and the data is correct, without needing to maintain a validator and a parser as two separate entities. Zod validators are defined in `src/stores/validators.ts`.

## 5.3 Project Structure

The application follows a typical Vue component application structure. The main HTML file is `index.html`, which provides very little content. However, it is the file which is served initially to the client. It contains `<div id="app">`, which is where Vue mounts the application. The Vue entry point is `src/main.ts`, wherein the app and its plugins are mounted. All other views and components are mounted by Vue automatically.

Components are located in `src/components` and are described below, as well as within in-code documentation. Components do not include CSS, as all styling is implemented using Tailwind. Therefore, elements include styling classes instead of within the HTML.

## 5.4 Vue Router

The project uses Vue Router<sup>10</sup>, the official SPA routing solution for Vue. Single-page routing simulates the traditional browser routing experience whilst avoiding page loads. The router's component acts as a placeholder, where the routed elements are swapped. The routed elements are regular Vue SFCs, but present in a different folder – `src/Views` for semantic purposes.

The router's configuration and route definitions can be in `src/router/index.ts`. An optimisation to reduce initial load time is employed here. The routes

---

<sup>8</sup><https://prettier.io/>

<sup>9</sup><https://zod.dev/>

<sup>10</sup><https://router.vuejs.org/>

```

1 <VueFlow
2   v-model:nodes="endpointStore.nodes"
3 >
4   <template #node-custom="props">
5     <CustomNode
6       :data="props"
7       @click="onClickNode(props)"
8     />
9   </template>
10 </VueFlow>

```

Listing 5.4: Passing a custom node component to VueFlow

beyond the initial one are imported dynamically, instead of statically, reducing the amount of files that must be sent to the client.

The router component is located in `src/App.vue`, alongside the header. Ultimately, the entire application besides the header is controlled by the router and each View therefore acts as its own route screen (hence the name).

The most important view `FlowView`. The visualisation and exports happen here. The `<VueFlow>` component is mounted here, modelling the data stores. Further, the bindings of the modal dialogues are defined here (more detail in 5.5).

Furthermore, the layout functionality is mounted here. This enables the user to select a layout for the existing nodes, before manipulating them further. The implementation can be found in `src/stores/layout.ts`.

## 5.5 Components

The application views are made up of components. Vue uses a single file component (SFC) structure, meaning each component is contained in its own file. Components are present in the `src/components` folder. Some components warrant additional description beyond what is included as in-code documentation.

### 5.5.1 CustomNode

The Flow library uses node components to model the provided data. It allows the definition of a custom node component to be used instead. `CustomNode` is passed to the `<VueFlow>` component in `FlowView`. Listing 5.4 shows how this is performed. Flow takes the modelled data and creates a custom node for each data object. Line 2 defines which data structure is modelled. Pay closer attention to line 5, where the modelled objects are passed to the node component as a property.

### 5.5.2 NodeModal

This modal dialogue is displayed to the user upon clicking a class node in the visualisation. It contains information about the node's attributes and incoming and outgoing edges. The modal contains three list sub-components, each of which displays one of the three options and can be toggled between. This dialogue is the only way that a user can add nodes or their attributes to the export selection. The

```

1  {
2    label: 'LDKit',
3    exporter: exportSchema satisfies Exporter,
4    prismClass: 'language-ts',
5    prettierConfig: {
6      semi: false,
7      parser: 'typescript',
8      plugins: [parserTypescript],
9    },
10 }

```

Listing 5.5: Defining an export format

selected entities will be included in the exported schema, performed in `ExportModal` (5.5.3).

### 5.5.3 ExportModal

The export modal employs the Prettier and PrismJS dependencies. The code is generated from the contents of the Pinia store in `src/stores/schema.js`. The output is passed to Prettier, which formats the code at runtime. This is an unusual use of Prettier, as it is usually used on the project's source code exclusively. It is important that the generated code is syntactically correct, or else the formatting will fail. The formatted code is passed to Prism, which performs syntax highlighting.

The default LDKit export is a transformation of Pinia JavaScript objects into text containing LDKit schema JavaScript objects. Each node creates an output class with its selected attributes and selected outgoing edges as fields. Attention must be paid to the proper definition of namespaces. LDKit comes with some most common namespace definitions, but others must be registered to LDKit. Every IRI that is being transformed into an output object must have its namespace defined.

The export functionality is set up to be modular and easily extensible. The `NodeModal` contains the `ExportOptions` component, wherein all export option definitions are located. Listing 5.5 shows the definition of the LDKit exporter. Once selected, the `exporter` function is provided with the selected data nodes. The output, alongside the Prism class and Prettier configuration, is passed back to the `NodeModal`. There, the class is injected into the Prism code block and Prettier is run using the defined configuration. When adding new export options, be mindful of the correctness of these definitions.

### 5.5.4 ImportModal

The `ImportModal` is used to manually import a schema into the application. It presents a text input, into which the input data is inserted. The modal contains a list of parsers, which can independently process the input text and populate the `endpointStore`. Each parser also provides a data schema, by which it validates the input data. The parser type definition is shown in Listing 5.6. The import options were designed with extensibility in mind and more details are provided in section 5.12.

```

1 type InputParser = (input: string) => SchemaType
2 type ImportDef = { parser: InputParser; schema: string }

```

Listing 5.6: Import format definition

```

1 SELECT DISTINCT ?class (COUNT(*) AS ?instanceCount)
2 WHERE {
3   ?s a ?class
4 }
5 GROUP BY ?class
6 ORDER BY DESC(?instanceCount)
7 LIMIT 10

```

Listing 5.7: Class discovery query

## 5.6 Visual Design

Some visual design methods were employed to ensure a consistent visual style. In order to appear orderly and avoid elements appearing out of place, care was given to ensure elements had consistent padding, rounding and colours.

Generic elements were turned into components to force their consistent use. Whenever a button is used, the `ButtonGeneric` fulfils that role. This means whenever the style of the button had to be modified, the change to the component is propagated correctly. Bypassing the use of the generic component would present obstacles, should there be changes to the visual style in the future. In line with this, component reuse is prominent with the header, although only a single instance is used and shared between the views.

TailwindCSS helps in keeping a coherent design system through the design of its utility classes. The consistency in its padding and margins ease the spacing decisions by granulating the available sizes to fewer possible steps. Tailwinds' colour definitions are also deliberate, providing a sizeable palette that was hand-picked and tested <sup>11</sup>.

## 5.7 Schema queries

A collection of TS modules handles the querying and response validation. The actual SPARQL queries are present in `src/stores/sparql.ts`. As explained in requirement R5, the queries are rather simple and are designed to extract sufficient information without overwhelming the endpoint. The query results should hold enough information to add to the visualisation as soon as they are complete without having to wait for other responses. This means that a pipeline of sorts is created. Once each request resolves, it is handled by a defined handler class in the `QueryQueue` 5.7.1. The initial query is 5.7, which discovers classes present in the endpoint.

Once the classes have been discovered, a series of queries can be run to find their literal attributes and edges between other existing classes. Query 5.8 is run for each pair of currently known classes.

<sup>11</sup><https://tailwindcss.com/docs/customizing-colors>

```

1 SELECT DISTINCT ?property (COUNT(*) AS ?instanceCount)
2 WHERE {
3   ?class1 a <${class1URI}> .
4   ?class2 a <${class2URI}> .
5   ?class1 ?property ?class2 .
6 }

```

Listing 5.8: Query finding edges between classes

```

1 SELECT DISTINCT ?attribute ?type (COUNT(1) AS ?instanceCount)
2 WHERE {
3   ?instance
4     a <${classURI}> ;
5     ?attribute ?targetLiteral
6   FILTER isLiteral(?targetLiteral)
7   BIND(datatype(?targetLiteral) AS ?type)
8 }
9 ORDER BY DESC(?instanceCount)

```

Listing 5.9: Query discovering a class' attributes

Simultaneously, a query to discover the new class' attributes is run. This finds a list of its literal attributes. The attributes are retrieved alongside their types and counts. The types are stored for later use in the export, where they are converted from type IRIs to target environment types.

All queries are executed using the fetch API<sup>12</sup>. The query is URI-encoded and inserted as a GET parameter. A JSON format GET parameter is also added to the fetched URL. Endpoints that do not provide JSON results or do not answer GET requests are currently not supported.

### 5.7.1 QueryQueue

This module handles the querying capabilities that are used to populate the data stores (5.8.1). It consists of a queue of queries that the application requested and executes them in order. The queue employs a lock mechanism to ensure it does not run queries simultaneously. This is used in order to not overwhelm the endpoint with hundreds of queries, which could result in the requests being denied or filtered.

The queued queries must provide the query string, a callback and a validator. Validators are defined using Zod (5.2.6)w and incoming responses are validated immediately. Once valid, they are passed to the callback function.

QueryQueue implements a clearing mechanism, which means that if query execution should stop, the queue is cleared and queries run before the clear are ignored should the return a response later. This is important for changing endpoints, as showing query results from the last point would lead to confusion.

<sup>12</sup>[https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)



## 5.8 Stores

State storage is managed by Pinia (5.2.2). Two main Pinia stores are present, accompanied by functional and utility modules.

### 5.8.1 EndpointStore

The `endpointStore` contains all information known about the endpoint. Primarily, that is the discovered nodes and edges of the endpoint. The data structures are defined to directly contain the query response objects, as parsed and validated by Zod. This data is used to display the edges and properties in node 5.5.2 and export 5.5.3 modal dialogues. The actual visualisation does not model these edges, as that would cause a render calculation for each present edge. Instead, a unique set of edges is maintained in `VisStateStore` 5.8.2.

This store also includes query callback functions. Once the `QueryQueue` returns a valid response, the callback functions place the response into the appropriate data structure within the store.

### 5.8.2 VisStateStore

The `visStateStore` holds information about the current state of the visualisation. It is closer to the user than `endpointStore`. As mentioned, the edges are used for visualisation, instead of all existing edges for performance reasons. When the user selects classes and edges they want to include in the export, those selections are tracked and handled in this store. The export takes this information from this store when required.

## 5.9 Data model

This section describes the shape of the data that is stored within the stores. The most important types are `StoreNode` and `StoreEdge`, which are defined in `src/stores/validators.ts` alongside the Zod validators for their constituent parts. The data that are received from the endpoint are passed to the validators to ensure their shape is correct. Once validated, Zod also provides a TypeScript type definition equivalent to the validation shape. This is used to conduct the parsing and validation steps at once. The parsed data is therefore stored in the same shape that it arrives in.

Observe Listing 5.10 to see the `StoreNode` type. The incoming data is stored in the `data` field of the node. `z.infer<...>` extracts the TypeScript type from the Zod parser definition. An alternative is that the data are not received as an endpoint response, but are imported from text (as described in section 4.2). In such cases, some of the fields expected by the Zod shapes are filled artificially from the import.

The `StoreNode` is being directly modelled by the Flow component. This is to avoid the need to manage a separate copy of the objects. That means that datatypes required by Flow must also be present. This creates the `position`, `id` and `type` fields. The IRI is used as an `id`, but for edges, an artificial `id` is created as a combination of source, `id` and target IRIs. Those three IRIs are also stored

```

1 type StoreNode = {
2   position: {
3     x: number
4     y: number
5     //...
6   }
7   id: string
8   type: string
9   data: {
10    node: z.infer<typeof NodeBinding>
11    labels: z.infer<typeof Labels>
12    attributes: z.infer<typeof AttributeBindingArr>
13  }
14 }

```

Listing 5.10: StoreNode type definition

separately on the Edge objects for easy access, as they are required often. Note that at runtime, Flow will create more fields on these objects.

## 5.10 Assets

The applications' assets, available in `src/assets` hold font files. Their use is managed by Tailwind in the font families definitions found in `tailwind.config.js`. Fonts were downloaded from open source font repositories <https://fontesk.com> and <https://fonts.google.com/>.

## 5.11 Tests

Unit tests are included with the application. Tests are present for parts of the core functionality but complete coverage was not pursued. Tests were created using Vitest<sup>13</sup>, which is the native testing framework for Vite. Tests are present in the `src/__tests__` folder, with each test file describing a module. Tests can be run using the `npm run test:unit` npm script.

## 5.12 Points of Extension

TypeSPARQ is designed in a way that expects future extension. The main points of extension that are considered are explained in this section.

### 5.12.1 Export formats

In order to implement a new export format, a new `ExportDef` must be registered in the `registeredExports` array in `src/components/ExportOptions.vue`. The definition (shown in Listing 5.11) should be implemented in order to provide all exporting

---

<sup>13</sup><https://vitest.dev/>

```

1 interface Exporter {
2   (nodes: StoreNode[], selectedAttributes: { [key: string]:
3     string[] }): string
4 }
5 type ExportDef = {
6   label: string
7   exporter: Exporter
8   prismClass: string
9   prettierConfig: Object
10 }

```

Listing 5.11: ExportDef type definition

functionality. Most importantly, the `Exporter` function will be handed the contents of `VisStateStore`, namely the user-selected nodes and attributes they wish to export.

Should you wish to make use of PrismJS syntax highlighting functionality, set the `PrismClass` field to a supported Prism language<sup>14</sup>. This is optional, an empty string may be used instead. In case a new language is used, the Prism style for the language will need to be imported as well. Do this in the `ExportModal` script.

To use Prettier formatting, provide a configuration object. Refer to the Prettier documentation<sup>15</sup> or use the TypeScript hints for more information. If you wish to skip the formatting step, use a falsy value.

The new implementation should preferably be provided as a standalone file. The exporter function should be imported into `ExportOptions` and used only in `registeredExports`.

### 5.12.2 Import format

To support a new import format, a parser implementation must be provided and subsequently registered in the `ImportModal` component.

The parser is a function that will be called and provided with the user input. The `endpointStore` provides the `handleParsedImport` function to populate itself with `StoreNode` and `StoreEdge` instances. This function accepts an object that satisfies the `SchemaType` type. Your parser implementation must create this `SchemaType` data object. `endpointStore.handleParsedImport(data)` will be called on your parsed input. Should your parser encounter an error, simply throw an `Error` with your message and it will be displayed to the user.

To register your parser, import it into the `ImportModal` and create an `ImportDef` object. Place it in the `importOptions` object alongside the schema string. The field you choose will be used as the button label.

### 5.12.3 Visualisations

Thanks to the design of the data model, adding a new visualisation does not require any complicated handling of the data. All fetching and handling is done by the stores and components contained within `VisSidebar`.

<sup>14</sup><https://prismjs.com/#supported-languages>

<sup>15</sup><https://prettier.io/docs/en/api.html#prettierformatsource-options>

```
1 endpointStore.nodes.forEach((n) => {
2   n.position.x = 0
3   n.position.y = 0
4 })
```

Listing 5.12: Layout example

First, create a new view and place it in `src/views`. Register it to the router in `src/router/index.ts` and add it to the `links` array in `src/components/PageHeader.vue`. When implementing the visualisation, make sure to take the following steps

**Include the `visSidebar` component** The `visSidebar` component provides the users with ways to control the state of the application. It also contains the importing and exporting functionality

**Model `endpointStore` data** The data store, located in `src/stores/endpoint.ts` contains all data extracted from the endpoints.

**Update `visStateStore`** This store holds the state of the visualisation. When users select and deselect nodes or attributes, they should be updated here. The exporter read the user selections from here.

#### 5.12.4 Flow layouts

Providing a new layout to the existing view requires defining the layouting function in `src/stores/layout.ts`. The function will be called when the user clicks the layout's button, which is created automatically from the `layoutTypes` dictionary. The layout definition must provide the function, a label and an optional tooltip. Your function is expected to mutate the `position` field of nodes, as the example in Listing 5.12.

# 6. Administrator documentation

This chapter describes how to set up the environment, how to build the application and how to host it. Everything was tested on a machine, running Windows 10 Home 64-bit using Node v18.12.1 and NPM 9.5.1. Since the application runs on the client, a server capable of hosting static content is sufficient. The source code of the application is available at <https://github.com/Jkuzz/sparql-explorer>.

## 6.1 Prerequisites

- Node.JS version 18+
- NPM version 9+

Node is available for download at its website<sup>1</sup> and should include NPM. Make sure they are both installed correctly by running `node --version`, `npm --version`.

## 6.2 Getting started

Once your environment is set up correctly, take the following steps to install the dependencies of the application.

1. Create an empty folder and navigate to it
2. Run `'git clone https://github.com/Jkuzz/sparql-explorer.git'` .
3. Run `'npm ci'` This will find the required dependencies as defined in `package.json` and download them to the `node_modules` folder.

## 6.3 Build

Once all prerequisites are installed, the application must be built. All build scripts are defined in `package.json` under `"scripts"`. They can be run using `npm` by executing `npm run [script]`. Primarily the following are present:

`npm run build` performs a TypeScript type check and then builds the application for production. The build output will be placed in the `./dist` folder.

`npm run dev` will host a local development server using Vite. This is used during development, as it utilises Vite's HMR to speed up developer feedback.

`npm run preview` will host a local server, which will contain the built application. This can be used to preview what the live, built application will look like.

---

<sup>1</sup><https://nodejs.org/en/download>

## 6.4 Hosting

In order to host the built application, provide the static content present in the `./dist` folder using your static hosting solution of choice.

At the time of writing, a demo of TypeSPARQ is available on GitHub using GitHub Pages<sup>2</sup>. The hosted version is rebuilt and redeployed automatically whenever the master branch of the repository is updated. This is performed via a GitHub Actions workflow, which is located in `./.github/workflows/deploy.yml`.

---

<sup>2</sup><https://pages.github.com/>

# 7. User documentation

This section serves as the user documentation for TypeSPARQ. It provides guidance on how to use the web tool and its exporting and importing features.

## 7.1 Using the web application

To begin using the schema extraction tool, navigate to the Schema tab. To begin exploring an endpoint, press the "Select endpoint" button, shown in Figure 7.1 as 1. Insert your endpoint's URL into the dialogue window and start the extraction by pressing "Extract". Make sure that the URL points to the SPARQL endpoint directly.

Once an endpoint is selected for extraction, schema extraction queries will be issued. You can track their progress in the sidebar, as shown in Figure 7.1 number 2. As the queries resolve, their results will be immediately added to the visualisation. The visualised nodes can be rearranged by dragging and zooming is performed using the mouse wheel. The nodes can be placed in one of the available layouts by expanding the layout slider on the top right and selecting a layout.

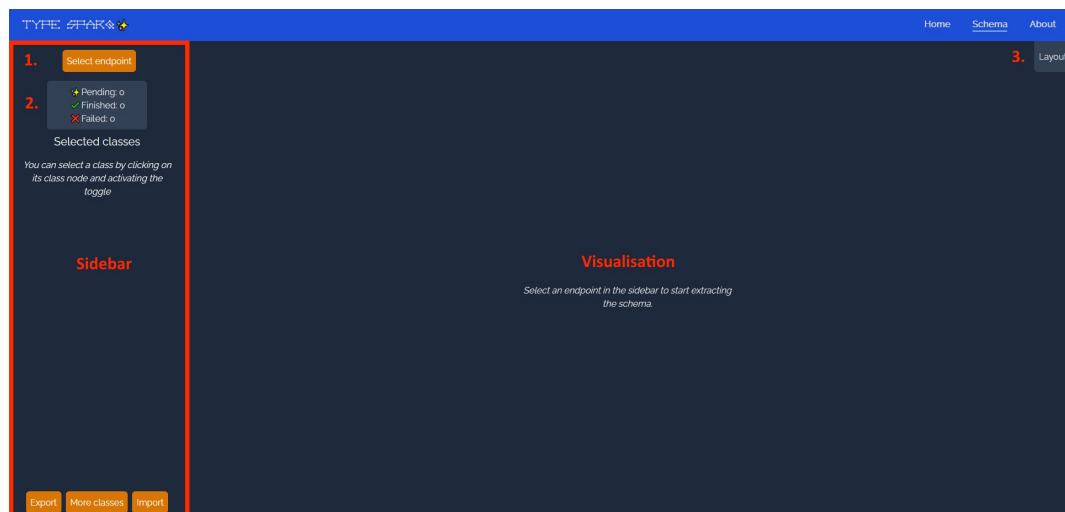


Figure 7.1: TypeSPARQ user interface

To see details about a class, click any of the class nodes to display a modal dialogue window. The dialogue window shown in Figure 7.2 shows information about the class. Here, the class' attributes and incoming and outgoing edges can be seen, alongside their types and occurrences. The type of relationship displayed can be changed using the tabs marked as 3. Edges can be filtered to only show the classes that are also selected using 4. The URIs can be also filtered via a substring search located in 2. "Occurrence" means how many instances of the attribute/edge are present on average on an instance of the class. For example, an occurrence of 120% says that an average instance of the class has 1.2 instances of the attribute.

The node modal allows you to select the class using the component marked as 1, at which point it will show up in the sidebar and be included in the schema

export. Similarly, attributes and edges can be selected as well.

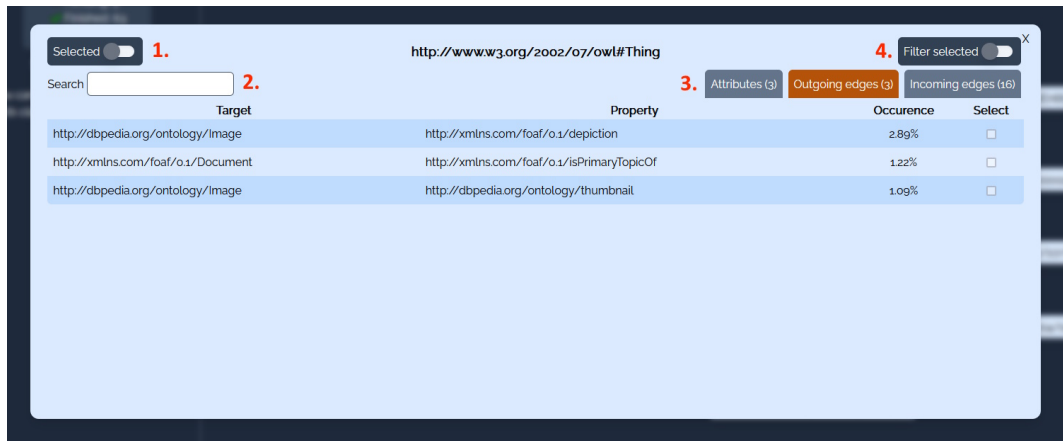


Figure 7.2: TypeSPARQ node detail modal window

More classes beyond the initial 10 can be queried. To do this, click the "More classes" button at the bottom of the sidebar and select the number of classes. Be mindful of visualising a large number of classes, as the number of queries for edges between classes grows exponentially.

To import a schema into the application, click the "Import" button at the bottom of the sidebar and select an import format in the displayed dialogue window. You can view the schema definition by clicking the "Schema" button on the bottom left. The input text is validated against this schema once the "Import" button is pressed. If valid, the schema will be immediately available in the visualisation. If the parser refuses the schema, an error will be displayed. Be mindful when importing a schema, since semantic errors will likely not be reported. Watch out for issues such as defining edges between non-existent node IRIs or duplicate class IRIs.

To export the selected schema parts, click the "Export" button at the bottom of the sidebar. An export dialogue will open, presenting a selection of possible export formats. Selecting one will generate the corresponding output and present it. Here, the schema can be copied or downloaded as a file.

## 7.2 Using the default exported schema

The default schema export provides an LDKit (*LDKit Linked Data query toolkit for TypeScript developers 2023*) schema definition. This section briefly illustrates how to start using it. For more in-depth information, see [ldkit.io](http://ldkit.io) and read the documentation.

To start using the LDKit export, you will need a JavaScript runtime, a bundler and a package manager. As an example, I suggest NodeJS, NPM and Vite (described further in section 5.2). Install the typescript transpiler and LDKit. In the example setup, this can be performed by running commands shown in Listing 7.1.

```
1 npm i typescript -D
2 npm i ldkit
```

Listing 7.1: Installing dependency packages



Once your environment is ready, add the file generated by TypeSPARQ into your project and begin using LDKit to access the SPARQL endpoint. See the Getting Started section of LDKit's documentation for simple working examples.

# 8. Evaluation

The viability of this approach was evaluated by running a series of scripts and testing responses of unknown endpoints. Their results are evaluated in this section. For information about unit tests, see Section 5.11.

## 8.1 Queries evaluation

The queries used to extract the schema information were tested on endpoints to verify their effectiveness. A list<sup>1</sup> of publicly available SPARQL endpoints was used to determine the endpoints that will be tested. It listed 90 unique and supposedly available endpoints.

A testing script was used that ran the class extraction query and recorded the response or any errors. The source code for it can be found at <https://github.com/Jkuzz/sparql-queries-testing>. The queries were given a timeout of 120 seconds. The results of the testing can be seen in Table 8.1. The endpoints were queried for their 10 most common classes and subsequently for the edges between the two most common classes. The queries used are identical to those described in Section 5.7. The measured states are described below.

**No response** is likely caused by the 120-second timeout. This means that the endpoint was either too slow or contained too many classes to answer the query in the given time. Since these were not tested further, it is possible that given more time, it would work, but no such guarantee can be made.

**Invalid response format** seems to be caused by endpoints that do not obey our queried URL parameters. The query URL contains the request format in the following way: `&format=application%2Fsparql-results%2Bjson`. These 10 endpoints responded with variations of HTML or XML responses. This is a possible point of improvement to improve the usability of the application. Unfortunately, it is not addressed at this moment due to time limitations.

**Error 40X** 3 Endpoints returned `400 Bad Request` and 1 returned `406 Not Acceptable`. This suggests a failure on the sent fetch request. However, upon manual inspection of all these cases, it turns out that the 406 is not accessible at all. Of the three 400s, two return errors are shown in Listing 8.1. The last one responds to the query correctly, suggesting it does not provide an API and its only interface is the query input. These are regarded as failures since the schema extraction could be performed given additional development. Given the number of these cases, this is not addressed.

**Error 50X** 3 Endpoints returning a `503 Service Unavailable` implies that reality has changed since recording the initial endpoint dataset and these are disregarded. The remaining errors are 2 `504 Gateway Timeout` and 5 `500 Internal Server Error`.

---

<sup>1</sup><https://skoda.projekty.ms.mff.cuni.cz/horizon/data-endpoint/sparql-2023-04-03.json>

```

1 Error: You have an error in your SQL syntax; check the manual
  that corresponds to your MySQL server version for the right
  syntax to use near '*) AS `instanceCount`

```

Listing 8.1: 400 responses error message

These cases are not regarded as a failure of the used queries, because presumably, any schema discovery query would have failed on these endpoints anyway.

**OK** These queries passed both query tests successfully. Interestingly, an outlier containing only one class was present, so the edge query could not be tested.

In summary, of the 86 applicable tested endpoints, 10 failed by their fault, 14 failed by our fault and the remaining 60 succeeded. Therefore, the success rate of queries used by TypeSPARQ can be evaluated at 81%.

Status	Number	Percentage
Total	90	100%
No response	3	3.3%
Invalid response format	10	11.1%
Error 40X	4	4.4%
503 Not Available	3	3.3%
Other 50X	7	7.7%
OK	63	70%

Table 8.1: Query evaluation results

### 8.1.1 Endpoint Evaluation

Since a volume of data was extracted from various endpoints, it would be amiss to not explore them. This section provides a brief exploration of the gathered data.

First, Figure 8.1 shows the frequencies and volumes of different classes. This only includes the top 10 most frequent classes in each endpoint. The explored endpoints appear very diverse in their top classes. Of the 63 queried endpoints a total of 372 different classes were present and 247 classes were only present in a single endpoint. Out of all discovered classes, only 4 classes were present in at least 10 endpoints and 12 classes were present in at least 5 endpoints. Table 8.2 shows the most ubiquitous classes.

Unsurprisingly, the most common classes are very generic. Interestingly, many of these common classes are equivalent, such as the Person and Agent classes. Some endpoints included all three Person classes with similar instance counts in their top 10, whilst others used only one. This suggests that sometimes, TypeSPARQ’s schema extraction process could produce misleading results if objects in the store are marked with multiple equivalent classes. Using ontologies during the schema extraction process would allow us to merge these classes using equivalence definitions.

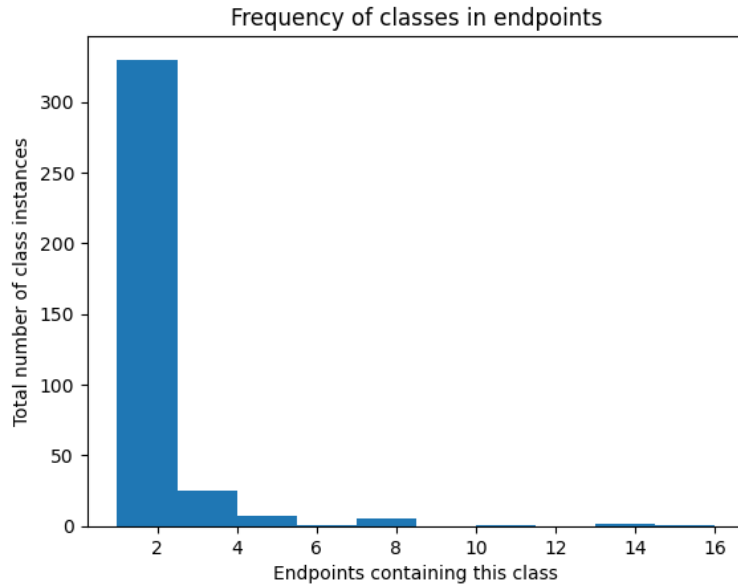


Figure 8.1: Number of endpoints containing different classes

Class IRI	#endpoints
<a href="http://www.w3.org/2004/02/skos/core#Concept">http://www.w3.org/2004/02/skos/core#Concept</a>	16
<a href="http://www.w3.org/2002/07/owl#Thing">http://www.w3.org/2002/07/owl#Thing</a>	14
<a href="http://xmlns.com/foaf/0.1/Document">http://xmlns.com/foaf/0.1/Document</a>	13
<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#Property">http://www.w3.org/1999/02/22-rdf-syntax-ns#Property</a>	10
<a href="http://dbpedia.org/ontology/Image">http://dbpedia.org/ontology/Image</a>	8
<a href="http://purl.org/linked-data/cube#Observation">http://purl.org/linked-data/cube#Observation</a>	7
<a href="http://schema.org/Person">http://schema.org/Person</a>	7
<a href="http://xmlns.com/foaf/0.1/Person">http://xmlns.com/foaf/0.1/Person</a>	7
<a href="http://dbpedia.org/ontology/Person">http://dbpedia.org/ontology/Person</a>	7
<a href="http://xmlns.com/foaf/0.1/Agent">http://xmlns.com/foaf/0.1/Agent</a>	6
<a href="http://dbpedia.org/ontology/Agent">http://dbpedia.org/ontology/Agent</a>	5
<a href="http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Agent">http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#Agent</a>	5

Table 8.2: Most commonly found classes in the top 10

Figure 8.2 shows the occurrence of classes and the total number of instances. It is noteworthy that there are classes present in very few endpoints, whose instance counts surpass those more common classes. Table 8.3 shows the classes with the most instances among the queried endpoints. Clearly, spatial databases and data cube observations generate the greatest volume of data points, but such endpoints appear to be rare. TypeSPARQ’s schema extraction process is capable of extracting even at these volumes thanks to relying on aggregation queries.

Figure 8.3 shows the instance counts of different namespaces. A total of 130 different namespaces were present among the top 10 classes of each endpoint. This is another measure of the massive diversity of endpoint contents. Every endpoint on average introduces at least two new namespaces to the potential data consumers. This combines with the previous view to support the following observation. Either SPARQL endpoints are extremely specialised, each requiring custom class definitions to support the stored data, or data stored in SPARQL

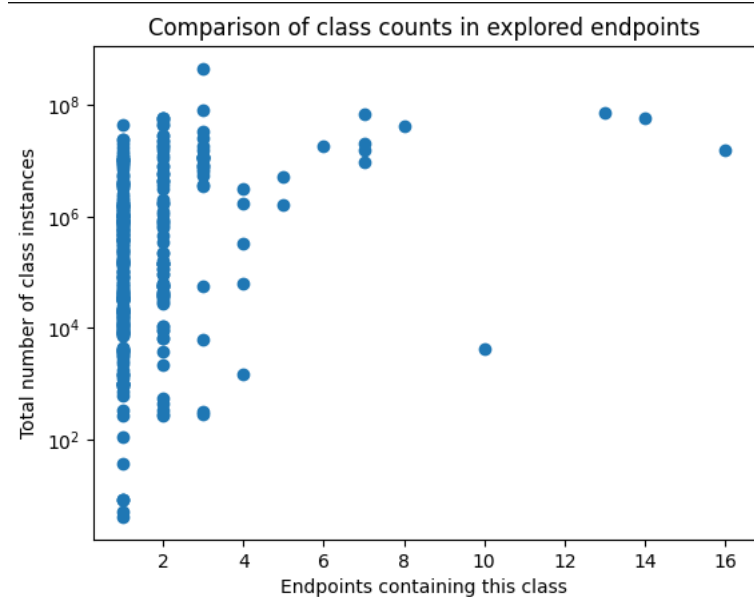


Figure 8.2: Comparison of frequency and volume of classes

Class IRI	Instances	Endpoints
<a href="http://geovocab.org/geometry#Geometr">http://geovocab.org/geometry#Geometr</a>	454492113	3
<a href="http://geovocab.org/spatial#Feature">http://geovocab.org/spatial#Feature</a>	82127836	3
<a href="http://xmlns.com/foaf/0.1/Document">http://xmlns.com/foaf/0.1/Document</a>	72519180	13
<a href="http://purl.org/linked-data/cube#Observation">http://purl.org/linked-data/cube#Observation</a>	69418092	7
<a href="http://schema.org/GeoCoordinates">http://schema.org/GeoCoordinates</a>	59023338	2
<a href="http://www.opengis.net/ont/gml#Point">http://www.opengis.net/ont/gml#Point</a>	59023338	2
<a href="http://www.opengis.net/ont/geosparql#Geometry">http://www.opengis.net/ont/geosparql#Geometry</a>	58869036	2
<a href="http://www.w3.org/2002/07/owl#Thing">http://www.w3.org/2002/07/owl#Thing</a>	57230656	14
<a href="http://linkedgeodata.org/meta/Node">http://linkedgeodata.org/meta/Node</a>	52703808	2
<a href="http://dati.camera.it/ocd/voto">http://dati.camera.it/ocd/voto</a>	45157643	1

Table 8.3: Classes with the most instances

endpoints fail to make use of foreign namespaces' classes and create their own instead.

These findings emphasise the heterogeneity of data stored within publicly available SPARQL endpoints. It suggests that endpoints tend to be very specialised and use a custom schema for each endpoint. This level of variety shows why it is very difficult to approach an endpoint and why tools like TypeSPARQ or similar are needed by developers.

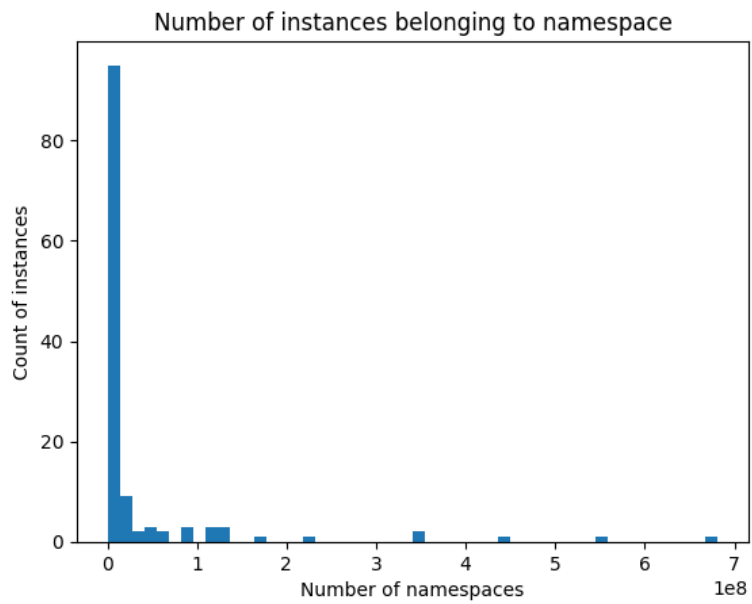


Figure 8.3: Counts of namespace member instances in endpoints

## 9. Conclusion

In this thesis, we presented TypeSPARQ, a web application for exploring and visualising a SPARQL endpoint’s schema. TypeSPARQ is an open-source project designed to be extensible and inter-operable with other services and libraries. TypeSPARQ allows users and developers unfamiliar with SPARQL to view the contents of the endpoint. Further, TypeSPARQ provides code generation integrated with LDKit that creates a TypeScript fetching API to access the selected subset of the endpoint.

This thesis explained the concepts of RDF and SPARQL. A multitude of existing approaches to schema extraction were explored and summarised. Iterating on the existing solutions, requirements for an application for extracting the schema of SPARQL endpoints were outlined. TypeSPARQ, the presented solution fulfils the outlined requirements. TypeSPARQ not only provides a standard visual presentation of the endpoint’s schema but specialises in exporting the schema in various easy-to-use formats. TypeSPARQ also accounts for the occurrences of properties of classes. By these metrics, all requirements for this thesis were fulfilled.

TypeSPARQ can be extended to work with any kind of schema extraction model by providing a schema import option. The schema export can also be extended to provide further functionality. These features ensure TypeSPARQ’s utility beyond the initial version. Developing more of these options is a point of future work. Considering not all RDF data are stored in SPARQL endpoints, extracting a visualising from an RDF dump.

Further future work includes integrating TypeSPARQ with other SPARQL libraries and generating starter code for other languages beyond TypeScript. This could involve platform-specific libraries similar to LDKit to provide static typing, or alternatively generating SPARQL queries to use with generic SPARQL libraries.

In exploring the endpoints using the schema extraction method, we found the method to work as designed in a majority of cases. Using simple aggregation queries seems to be a fitting solution for schema extraction from unknown endpoints with user experience in mind. Furthermore, we concluded that the variety of data contained within the publicly available endpoint is vast. This fabricates the need for software and approaches like the one presented in this thesis and the need for their further development.

# List of Abbreviations

Abbreviation	Definition
RDF	Resource Description Framework
RDFS	RDF Schema
LD	Linked data
W3C	World Wide Web Consortium
SPARQL	SPARQL Protocol and RDF Query Language
OWL	Web Ontology Language
JSON	JavaScript Object Notation
SQL	Structured Query Language
HTTP	Hypertext Transfer Protocol
API	Application programming interface
EDG	Enterprise Data Governance
SHACL	Shapes Constraint Language
UML	Unified modeling language
IRI	Internationalised Resource Identifier
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
TS	TypeScript
DOM	Document Object Model
SPA	Single Page Application
SEO	Search Engine Optimisation
SSR	Server-side Rendering
NPM	Node Package Manager
HMR	Hot Module Replacement
SFC	Single-File Components

Table 9.1: List of abbreviations



# References

- Zviedris, Martins and Guntis Barzdins (2011). “ViziQuer: A Tool to Explore and Query SPARQL Endpoints”. In: *The Semantic Web: Research and Applications*. Ed. by Grigoris Antoniou et al. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 441–445. ISBN: 978-3-642-21064-8.
- Fischer, Lars and Stefan Hanenberg (Oct. 2015). “An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability Using TypeScript and JavaScript in MS Visual Studio”. In: *SIGPLAN Not.* 51.2, pp. 154–167. ISSN: 0362-1340. DOI: 10.1145/2936313.2816720. URL: <https://doi.org/10.1145/2936313.2816720>.
- Dudáš, Marek, Vojtěch Svátek, and Jindřich Mynarz (2016). “Dataset Summary Visualization with LODSight”. In: URL: [https://link.springer.com/chapter/10.1007/978-3-319-25639-9\\_7](https://link.springer.com/chapter/10.1007/978-3-319-25639-9_7) (visited on 07/04/2022).
- Florenzano, Fernando et al. (2016). “A Visual Aide for Understanding Endpoint Data”. In: URL: <http://ceur-ws.org/Vol-1704/paper9.pdf> (visited on 07/06/2022).
- Weise, Marc, Steffen Lohmann, and Florian Haag (2016). “LD-VOWL: Extracting and Visualizing Schema Information for Linked Data Endpoints”. In: *Proceedings of the 2nd International Workshop on Visualization and Interaction for Ontologies and Linked Data (VOILA 2016)*. Vol. 1704. CEUR-WS. CEUR-WS.org, pp. 120–127. URL: <http://ceur-ws.org/Vol-1704/paper11.pdf>.
- Almeida, Fernando and José Monteiro (2017). “The Role of Responsive Design in Web Development”. In: 14 No. 2, pp. 48–65. URL: [https://www.researchgate.net/profile/Fernando-Almeida-10/publication/324131848\\_The\\_role\\_of\\_responsive\\_design\\_in\\_web\\_development/links/5aca790ea6fdcc8bfc84eea8/The-role-of-responsive-design-in-web-development.pdf](https://www.researchgate.net/profile/Fernando-Almeida-10/publication/324131848_The_role_of_responsive_design_in_web_development/links/5aca790ea6fdcc8bfc84eea8/The-role-of-responsive-design-in-web-development.pdf) (visited on 07/19/2022).
- Anutariya, Chutiporn and Reshma Dangol (2018). “VizLOD: Schema Extraction And Visualization Of Linked Open Data”. In: *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–6. DOI: 10.1109/JCSSE.2018.8457325.
- Taelman, Ruben, Miel Vander Sande, and Ruben Verborgh (2019). “Bridges between GraphQL and RDF”. In: URL: <https://www.w3.org/Data/events/data-ws-2019/assets/position/Ruben%20Taelman.pdf>.
- Werbrouck, Jeroen et al. (2019). “Semantic query languages for knowledge-based web services in a construction context”. In: URL: [https://www.researchgate.net/publication/334263376\\_Semantic\\_query\\_languages\\_for\\_knowledge-based\\_web\\_services\\_in\\_a\\_construction\\_context](https://www.researchgate.net/publication/334263376_Semantic_query_languages_for_knowledge-based_web_services_in_a_construction_context).
- Gleim, Lars et al. (2020). “Automatic Bootstrapping of GraphQL Endpoints for RDF Triple Stores”. In: URL: <http://ceur-ws.org/Vol-2722/quweda2020-paper-2.pdf> (visited on 06/27/2022).
- Gleim, Lars Christoph et al. (2020). “Enabling ad-hoc reuse of private data repositories through schema extraction”. In: *Journal of Biomedical Semantics*. URL: <https://jbiomedsem.biomedcentral.com/articles/10.1186/s13326-020-00223-z> (visited on 06/26/2022).

- Announcing TypeScript 1.0* (2023). URL: <https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/> (visited on 03/20/2023).
- Code using GraphQL* (2023). URL: <https://graphql.org/code/> (visited on 03/25/2023).
- Git Version Control System* (2022). URL: <https://git-scm.com/> (visited on 08/07/2022).
- Gleim, Lars et al. (2022). *UltraGraphQL Translation Phase*. URL: [https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/docs/translation\\_phase.md](https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/docs/translation_phase.md) (visited on 07/03/2022).
- Group, Ontology Engineering (2022). *Linked Open Vocabularies*. URL: <https://lov.linkeddata.es/dataset/lov/> (visited on 06/27/2022).
- HTML Living Standard* (2022). URL: <https://html.spec.whatwg.org/multipage> (visited on 08/07/2022).
- International, ECMA (2022). *ECMAScript® programming language standard*. URL: <https://www.ecma-international.org/technical-committees/tc39/> (visited on 08/07/2022).
- Introducing JSON* (2022). URL: <https://www.json.org/json-en.html> (visited on 08/07/2022).
- LDKit Linked Data query toolkit for TypeScript developers* (2023). URL: <https://ldkit.io/> (visited on 03/20/2023).
- Ltd., Semantic Integration (2022). *HyperGraphQL*. URL: <https://www.hypergraphql.org/tutorial/> (visited on 06/24/2022).
- Simplod - Visualization tool for simplifying access to Linked Data*. (2023). URL: <https://jaresan.github.io/simplod/> (visited on 04/16/2023).
- SPARQLess* (2023). URL: <https://mff-uk.github.io/sparqlless/> (visited on 04/10/2023).
- Taelman, Ruben, Miel Vander Sande, and Ruben Verborgh (n.d.). *GraphQL-LD: Linked Data Querying with GraphQL*. URL: <https://biblio.ugent.be/publication/8578324/file/8579408.pdf>.
- TopQuadrant, Inc. (2022). *Querying TopBraid EDG with GraphQL*. URL: <https://www.topquadrant.com/querying-topbraid-edg-with-graphql/> (visited on 06/24/2022).
- Union, Stardog (2022). *Stardog GraphQL Documentation*. URL: <https://docs.stardog.com/query-stardog/graphql> (visited on 06/24/2022).
- Usage statistics of JavaScript as client-side programming language on websites* (2022). URL: <https://w3techs.com/technologies/details/cp-javascript/> (visited on 08/07/2022).
- Vite* (2023). URL: <https://vitejs.dev/> (visited on 03/23/2023).
- VueJS* (2023). URL: <https://vuejs.org/> (visited on 03/23/2023).
- W3C (2022a). *A JSON-based Serialization for Linked Data*. URL: <https://www.w3.org/TR/json-ld/> (visited on 08/09/2022).
- (2022b). *A line-based syntax for an RDF graph*. URL: <https://www.w3.org/TR/n-triples/> (visited on 08/09/2022).
- (2022c). *OWL Web Ontology Language Overview*. URL: <https://www.w3.org/TR/owl-features/> (visited on 06/19/2022).
- (2022d). *RDF 1.1 Concepts and Abstract Syntax*. URL: <https://www.w3.org/TR/rdf11-concepts/> (visited on 06/18/2022).

- W3C (2022e). *RDF Schema 1.1*. URL: <https://www.w3.org/TR/rdf-schema/> (visited on 06/18/2022).
- (2022f). *Shapes Constraint Language (SHACL)*. URL: <https://www.w3.org/TR/shacl/> (visited on 06/19/2022).
- (2022g). *SPARQL 1.1 Overview*. URL: <https://www.w3.org/TR/sparql11-overview/> (visited on 06/18/2022).
- (2022h). *Terse RDF Triple Language*. URL: <https://www.w3.org/TR/turtle/> (visited on 08/09/2022).