ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Doroshenko** Jméno: **Yan** Osobní číslo: **406938**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Editor sémantických datových proudů

Název bakalářské práce anglicky:

Semantic pipeline editor

Pokyny pro vypracování:

SPipes is RDF-based scripting language based on SPARQL motion. It defines data pipelines in form of acyclic oriented graph of modules. Concrete modules are constructed in Java or defined declaratively within RDF. The goal of this work is to create web-based editor and debugger for SPipes scripts. The editor should support creation of scripts and declarative RDF-based modules. Moreover, it should be possible to use the editor together with a generic RDF editor.

1) review existing libraries for visualization of dataflows appropriate for SPipes scripts
2) analyze possible patterns to modularize SPipes scripts and define use cases for editing and debugging the scripts
3) design and implement the editor based on previous findings
4) compare implemented editor with existing SPARQL motion editor implemented in Topbraid Composer
5) test the implementation on specified use cases

Seznam doporučené literatury:

[1] Blaško, Miroslav and Petr Křemen, "SPipes" (online at https://kbss.felk.cvut.cz/web/kbss/s-pipes)
[2] Křemen, Petr, and Zdeněk Kouba. "Ontology-driven information system design." IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42.3 (2012): 334-344.
[3] Lanthaler, Markus, and Christian Gütl. "On using JSON-LD to create evolvable RESTful services." Proceedings of the Third International Workshop on RESTful Design. ACM, 2012.
[4]TopQuadrant, Inc. "SPARQL motion" (online at http://sparqlmotion.org)

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Mgr. Miroslav Blaško, Ph.D., Skupina znalostních softwarových systémů FEL

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

Datum zadání bakalářské práce: **19.02.2018** Termín odevzdání bakalářské práce:

Platnost zadání bakalářské práce: **30.09.2019**

Mgr. Miroslav Blaško, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Semantic Pipeline Editor

Yan Doroshenko
Mentor: Mgr. Miroslav Blaško, Ph.D.

Software Engineering and Technologies, Faculty of Electrical Engineering,
Czech Technical University in Prague, 2018

Acknowledgement

I would like to thank my mentor, Mgr. Miroslav Blaško, Ph.D., for his unprecedented dedication and endless enthusiasm and Ing. Martin Ledvinka, for his immense assistance and benevolence.

Author's Affirmation

I hereby declare that the submitted thesis is exclusively my own work and that I have listed all used information sources in accordance with the Methodological Guideline on Ethical Principles for College Final Work Preparation.

Prague May 24, 2018 Yan Doroshenko

Abstract

SPipes is a framework for processing web content using Semantic Web technologies. A pipeline is defined as an oriented graph with modules as nodes executed in order defined by the graph edges. This document contains the analysis of SPipes scripts based on the existing project, as well as design and implementation description for the SPipes editor, capable of editing and executing scripts and configuring modules using the SForms library
Keywords: Semantic Web, RDF, Ontology, Jena, JOPA, SPARQL, SPipes, Graph, Scala, Java, Spring, React, SForms

Abstrakt

SPipes je framework pro zpracování dat z webu pomocí technologií Sémantického webu. Datový proud je definován ve formě orientovaného grafu, kde uzly reprezentují moduly, spouštějící v pořadí, definovaném hranami. Tato práce popisuje analýzu skriptu SPipes na základě existujícího projektu a také návrh a implementaci SPipes editorů, umožňujícího editovat a spouštět skripty a konfigurovat moduly pomocí knihovny SForms.
Klíčová slova: Semantic Web, RDF, Ontology, Jena, JOPA, SPARQL, SPipes, Graph, Scala, Java, Spring, React, SForms

Contents

1 Introduction **2**
1.1 Project Goals 2

2 Background **4**
2.1 Domain Diagram 4
2.2 Semantic Web 4
2.3 RDF 5
2.3.1 RDFS 6
2.3.2 OWL 6
2.3.3 RDF Serialization Formats 7
2.3.4 SPARQLMotion 10
2.4 SPipes 11

3 Design **13**
3.1 Analysis of the existing SPipes scripts 13
3.1.1 Form model 16
3.1.2 Script Modularization Patterns 17
3.2 Integration Diagram 22
3.2.1 SPipes Engine 23
3.2.2 SPipes Editor Backend 23
3.2.3 SPipes Editor Frontend 23
3.3 Use Case Diagram 24
3.4 Requirements 25
3.4.1 Functional Requirements 25
3.4.2 Non-functional Requirements 26
3.5 Existing Solutions 26
3.5.1 Evaluation Criteria 27
3.5.2 Editors 28
3.5.3 Graph Visualization Libraries 29
3.5.4 Feature Matrix 30
3.6 Debugging Proposal 32

4 Implementation **34**
4.1 Application Architecture 34
4.2 Technology Stack 35
4.2.1 Scala 35

4.2.2 Spring . . . 35
4.2.3 Jena . . . 36
4.2.4 JOPA . . . 36
4.2.5 React . . . 37
4.2.6 SForms . . . 37
4.3 JSON-LD REST API . . . 38
4.4 Data Transformation . . . 38
4.4.1 Model Generation . . . 38
4.4.2 Graph Visualization . . . 39
4.4.3 Configuration Form Generation . . . 40
4.5 File System Synchronization . . . 47
4.6 Implementation Issues . . . 48
4.7 Application User Interface . . . 49

5 Testing **50**
5.1 Test Model . . . 50
5.2 Scenarios . . . 50
5.3 Automated Code Testing . . . 55
5.3.1 Statistics . . . 55
5.3.2 Issues . . . 56

6 Comparison With Topbraid Composer **56**

7 Conclusion **59**
7.1 Project Goals Fulfillment . . . 59
7.2 Future Development . . . 60

8 Installation Guide **61**
8.1 Prerequisites . . . 61
8.2 SPipes Editor . . . 61
8.3 Local SPipes Engine . . . 61

9 References **62**

10 Appendix **64**
10.1 Abbreviations . . . 64
10.2 CD Contents . . . 65

1 Introduction

The World Wide Web caused deep changes in communication between people and the way business is conducted. It enabled instant data transfer and reduced information fetch latency from hours or even days to seconds. But with the increasing complexity of the Web and growing number of web applications arose the problem of exchanging information without the loss of meaning. The World Wide Web extension called Semantic Web was created to solve this problem. Semantic Web tools are numerous, but the one that stands out due to its importance is the Resource Description Framework or RDF, a set of specifications for describing information on the Web in a uniformed way. RDF spawned several data description languages like SPARQL-Motion – a scripting language for describing data processing pipelines with a graphical notation, that represents pipelines as *script*s consisting of *module*s (processing nodes) and dependencies between them showing the dataflow direction. Modules can be defined declaratively in RDF or as a Java class, implementing a specific interface. Knowledge-based Software Systems group of the Department of Cybernetics, CTU in Prague, is developing its own dialect of SPARQLMotion, the SPipes, which is for now just a restriction of SPARQLMotion and the only added value are the concrete modules and the SPipes Engine, capable of SPipes scripts execution. Due to the nature of the SPARQLMotion (and SPipes) scripts they can be represented in form of an acyclic directed graph and therefore edited as one, using a suitable editor. The goal of the project is to design and partially implement the SPipes Editor and test it on typical use cases.

This document gives an insight into the domain of the SPipes (Section 2), describes the analysis of existing solutions, functional requirements and use cases for the SPipes Editor, as well as the model used and design of the SPipes Editor (Section 3), its implementation (Section 4), testing (Section 5) and comparison with the SPARQLMotion editor Topbraid Composer (Section 6).

1.1 Project Goals

The goals of this thesis are the following:

- Review existing graph visualization libraries and editors, based on specific requirements.
- Analyze possible patterns to modularize SPipes scripts.

- Define use cases for editing and debugging the scripts.
- Implement the editor for SPipes scripts, based on the analysis results.
- Compare the implemented editor with the SPARQLMotion editor Topbraid Composer.
- Test the editor on specified use cases.

2 Background

This section briefly describes the backgound for the resulting application and provides definitions for some of the important concepts.

2.1 Domain Diagram

The diagram, giving an initial insight into the domain in question is shown in the Figure 1. Each entity in the diagram is relevant for the scope of this project. The most significant ones will be described in the corresponding sections of this document.

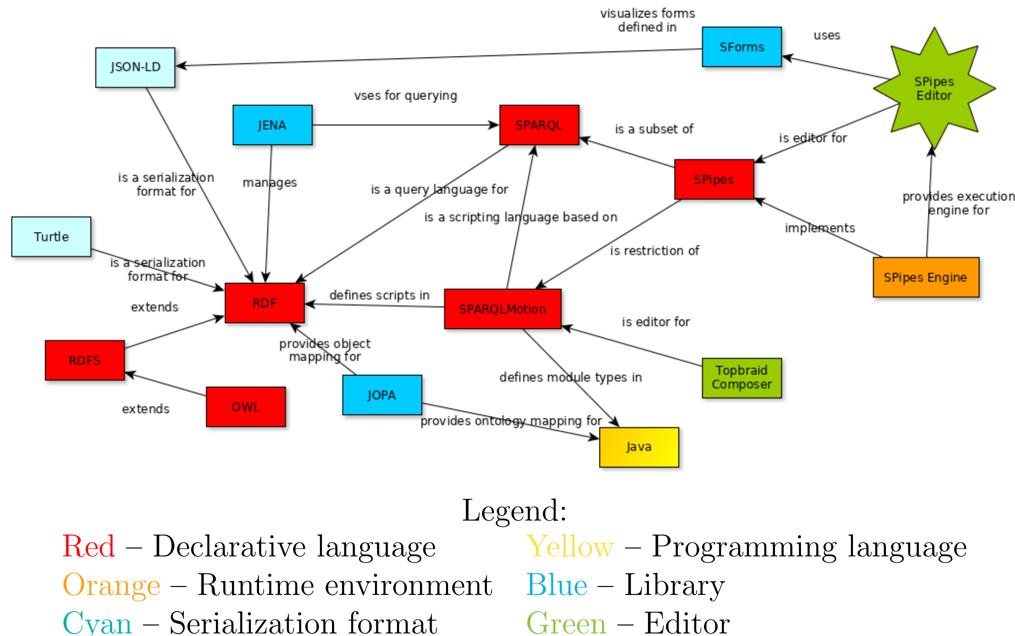

Figure 1: Domain diagram

2.2 Semantic Web

Semantic Web is a World Wide Web extension developed by the World Wide Web Consortium to solve the problem of data exchange between applications in the realm of rapidly growing Web. The way it achieves the goal

is representing Web content in a machine-processable form and using special techniques to take advantage of this representation. It wasn't necessarily meant to replace the World Wide Web, but rather to gradually evolve from it[1]. Semantic Web was created by its authors as a set of specifications and standards for data representation, rather than a particular technology stack. Therefore several technologies compliying with the standards are being developed, the main one being the Resource Description Framework also referred to as RDF.

2.3 RDF

The Resource Description Framework (RDF) is a framework for describing *resource*s. Anything can be viewed as a resource, including people, documents, physical objects and even abstract concepts. The main goal of RDF is to express information in a form that can be processed by different applications across the Web without the loss of meaning. As a framework, RDF lets application developers to use different available processing tools[2].

RDF features a mechanism for syntax-neutral resource description without defining a specific domain semantics, which means that any domain can be described using RDF model[13].

RDF data are represented as *triple*s `<subject> <predicate> <object>`. Each triple represents a relationship, where `subject` is a resource, connected to the *object* (that does not necessarily need to be a resource, can be a literal value like a string or a number) by the `predicate`, that represents the nature of the relationship and is called a *property*.

A collection of triples is a *graph* that represents an *ontology* or a part of it, ontology being a piece of information about a particular domain.

Informal example of RDF triples, showing the general structure, is shown in the Figure 2.

```
<Bob> <is a> <person>.
<Bob> <is a friend of> <Alice>.
<Bob> <was born on> <the 26th of April 1986>.
```

Figure 2: Triple examples

2.3.1 RDFS

RDF Schema is an extension of the RDF, which provides a data-modelling vocabulary, defining a backbone for the application designers to base their data models on, but not attempting to enumerate all the possible forms of data representation. Some of the important RDFS definitions are[15]:

- `rdfs:`

  `rdfs:` is the prefix for all the RDFS definitions and refers

  to `http://www.w3.org/2000/01/rdf-schema#`.

- `rdfs:Resource`

  All the things, described by RDF are resources – instances of the `rdfs:Resource` class.

- `rdfs:Literal`

  `rdfs:Literal` class represents literal values like strings and numbers.

- `rdfs:Property`

  `rdfs:Property` is a class of RDF properties.

2.3.2 OWL

The Web Ontology Language is developed as a vocabulary extension of RDF to serve as a semantic markup language for publishing and sharing ontologies. OWL defines many useful terms like `owl:imports` and `owl:Ontology`[16]. Imports play particularly important role in the SPipes scripts modularization.

An `owl:imports` statement references another ontology, whose definition is considered a part of the importing ontology. Importing ontology together with the imported triples is called an *import closure*. Importing is done through referencing the imported ontology URI and is transitive[16]. An example of importing is shown in the Figures 3 and 4 (prefixes are omitted for the sake of clarity), where the `fel:fruits` ontology imports `fel:apples` and thus contains the information about the `fel:apples/red-apple`.

```
fel:apples a owl:Ontology .

fel:apples/red-apple rdfs:label "This is a red apple" .
```

Figure 3: Apples ontology

```
fel:fruits a owl:Ontology ;
                owl:imports fel:imported .
```

Figure 4: Fruits ontology imports the apples ontology

2.3.3 RDF Serialization Formats

There are multiple serialization formats for the RDF used for textual representation of information. While keeping the general RDF structure (in that data is represented by triples) each language has different syntax, which makes it useful for a specific purpose.

The most important ones for the purpose of this project are:

- **Turtle**

  A Turtle document is an RDF serialization format that allows writing down an RDF graph in a compact textual form[3], suitable for human comprehension and processing. It allows for defining prefixes in the beginning of the document thus simplifying the document reading for a human. Also there is an anonymous nodes syntax which can be used to represent an entire collection of triples as a single object, that also positively affects the human perception of the document. An example is shown in the Figure 5.

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dc:  <http://purl.org/dc/elements/1.1/> .
@prefix ex:  <http://example.org/stuff/1.0/> .

<http://www.w3.org/TR/rdf-syntax-grammar>
   dc:title "RDF/XML Syntax Specification (Revised)" ;
   ex:editor [
      ex:fullname "Dave Beckett";
      ex:homePage <http://purl.org/net/dajobe/>
   ] .
```

Figure 5: Turtle document example[3]

- **JSON-LD**

  JSON-LD is an RDF serialization format that combines the simplicity and power of JSON with the concepts of Semantic Web. It combines three ways to process linked data: considering raw triples, using a graph processing API and building a tree structure from a portion of the graph[4]. In practice what JSON-LD provides us is the mechanism to access linked data from JavaScript as if it were a plain JSON. A JSON-LD example is shown on Figure 6.

```
{
    "@context": {
        "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
        "dc": "http://purl.org/dc/elements/1.1/",
        "ex": "http://example.org/stuff/1.0/"
    },
    {
        "@id": "http://www.w3.org/TR/rdf-syntax-grammar"
        "dc:title": "RDF/XML Syntax Specification (Revised)",
        "ex:editor": {
            "ex:fullname": "Dave Beckett",
            "ex:homePage": "http:purl.org/net/dajobe"
        }
    }
}
```

Figure 6: JSON-LD document example

- **SPARQL**

  SPARQL is a query language for RDF-based databases, allowing to retrieve and manipulate data stored in RDF[5]. The example SPARQL query is shown in the Figure 7. It selects distinct RDF nodes, based on the graph pattern `?module a ?type` (SPARQL variable `?module` is of type `?type`) and the filter WHERE clause that checks the absence of the subtype for `?type`, such that `?module a ?subtype` (`?module` is of type `?subtype`) and type and subtype are different.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?type where {
    ?module a ?type .
    FILTER NOT EXISTS {
        ?module  a ?subtype .
        ?subtype rdfs:subClassOf ?type .
        FILTER ( ?subtype != ?type )
    }
}
```

Figure 7: SPARQL query example

2.3.4 SPARQLMotion

SPARQLMotion is a scripting language based on SPARQL used to define data processing pipelines. It features a graphical notation for visualization[6]. The key concepts of SPARQLMotion are:

- Script

  Script is a textual representation of a data processing pipeline, written in some of the RDF languages. It can be represented in a form of a directed acyclic graph where the direction of the edges represents the flow of data. Scripts can be stored in files in any RDF serialization format (Turtle, JSON-LD, etc), having an extra `.sms` extension in case the script is meant to have its functions available through REST API. Each script is an ontology that can import triples from other ontologies.

- Variable

  SPARQLMotion variables can be bound to RDF nodes, compliant with SPARQL in a way that SPARQLMotion variables can be used within SPARQL queries and SPARQL expressions and vice versa. SPARQLMotion variables are shared throughout the entire script within a single scope.

- Module

  Module is a processing step of the pipeline, that can be represented as a node of the graph. It can be defined declaratively in RDF or programatically as a Java class, implementing a specific interface. Module can return RDF graph, bind a variable or do both. They can depend on each other in a way that one module consumes the data another one produces. These dependencies can be represented as an edge of the graph, the target node of the edge being the dependent module.

  Input modules are modules that depend on no other module within the pipeline. They can take not only RDF as input. Output modules are the modules no module depends on. They are typically associated with functions and don't have to return necessarily RDF.

- Function

  SPARQLMotion function is a function, backed by a SPARQLMotion script, which is executed on function call. When called, function binds its output parameters as variable bindings to specific modules. A function must point to the end of the pipeline and must not have any side effects.

- Module Type

  Module type is a definition of module's behavior (typically being driven by a SPARQL query). It defines parameters that individual modules can specify. Many module type definitions are a part of SPARQLMotion's core library, but the new ones can be defined as well.

2.4 SPipes

SPipes is an RDF scripting language, based on SPARQLMotion[8]. The main distinction is that SPipes does not allow cycles (repeated execution of the part of the pipeline, depending on the parameter). Moreover SPipes functions bind their output values globally in the shared pipeline scope instead of passing them to specific modules. At the moment the only RDF serialization format supported by SPipes is Turtle.

In addition, each module within SPipes can define constraints, in form of

SPARQL Ask queries[1], for RDF graph and bindings that it consumes/produces.

Spipes Engine, an implementation of the SPipes language, allows to serialize execution data and metadata into an ontology, compliant with Dataset Description Ontology[2] which can be saved into set of files or RDF4j server repository[14]. The latter allows to not only query execution data and metadata, but also to define pipelines based on previous executions (adaptive pipelines).

[1] https://www.w3.org/TR/rdf-sparql-query/#ask
[2] http://onto.fel.cvut.cz/ontologies/ddo/current/index-en.html

3 Design

This section describes the existing SPipes scripts and defines the possible script modularization patterns, lists the application requirements, based on the defined patterns, and describes the application design phase and the analysis it is based upon.

3.1 Analysis of the existing SPipes scripts

SPipes Editor was tested on existing scripts from the `16gacr-model` from the project Efficient Exploration of Linked Data Cloud No. GA 16-09713S of the Grant Agency of the Czech Republic[17]. This is a model, used in production by the KBSS group[3], featuring several pipelines of various complexity built on top of the `s-pipes-modules` module type definitions. `s-pipes-modules` is an extension to the SPARQLMotion Core library, featuring several module types, tailored for the KBSS group needs. Most of them are implemented using Java extension mechanism.

One of the scripts from `16gacr-model`, that will be used throughout the thesis, is shown in the Figure 8 as an example. It computes schema for the RDF data from a repository, specified by `snapshotServiceUrl`, using common RDF graph summarization techniques[14].

[3] kbss.felk.cvut.cz

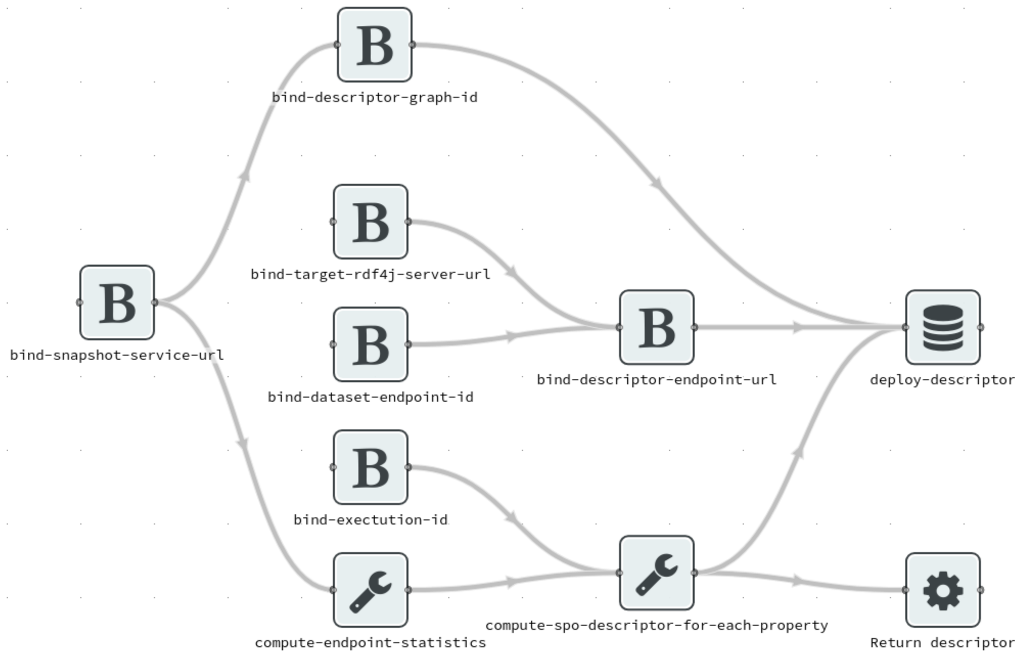

Figure 8: Example script from the `16gacr-model`

Modules with a B icon are of type `Bind with constant`, they takw a variable name and a value and set the value to the variable. The wrench modules are of type `Apply construct`, they take SPARQL CONSTRUCT query as input parameter and return an output graph that is result of applying the query on its input graph. The `Return descriptor` and `deploy-descriptor` modules are the output modules, having the functions bound to them. The former returns an input RDF graph in the specified serialization format, while the latter deploys the input to the RDF4J repository. The data flow in this script is the following:

1. Variable `snapshotServiceUrl`, representing the URL of some SPARQL endpoint providing input data[4] is bound by the corresponding module.

2. • `descriptorGraphId` is bound, using the value of `snapshotServiceUrl`.

[4]In case of the example shown in the Figure 9 it's `https://linked.opendata.cz/resource/dataset/seznam.gov.cz/rejstriky/objednavky`

- `targetRdf4jServerUrl, datasetEndpointId` and `executionId` are bound by the corresponding modules.
- Number of triples for each property is computed for the graph at `snapshotServiceUrl` by the `compute-endpoint-statistics` module.

3. - Value of `descriptorEndpointUrl` is assembled from the `targetRdf4jServerUrl` and `datasetEndpointId`.
   - Additional metadata is collected for each property.

     by the `compute-spo-descriptor-for-each-property`.

4. The resulting metadata is returned in the specified serialization format from the `Return descriptor` module, or deployed to the RDF4J repository and graph from `descriptorEndpointUrl` and `descriptorGraphId` respectively.

Result of the function `compute-spo-summary-descriptor-metadata`, applied on the Orders[5] dataset from the Czech Linked Data Cloud[6], is shown in the Figure 9.

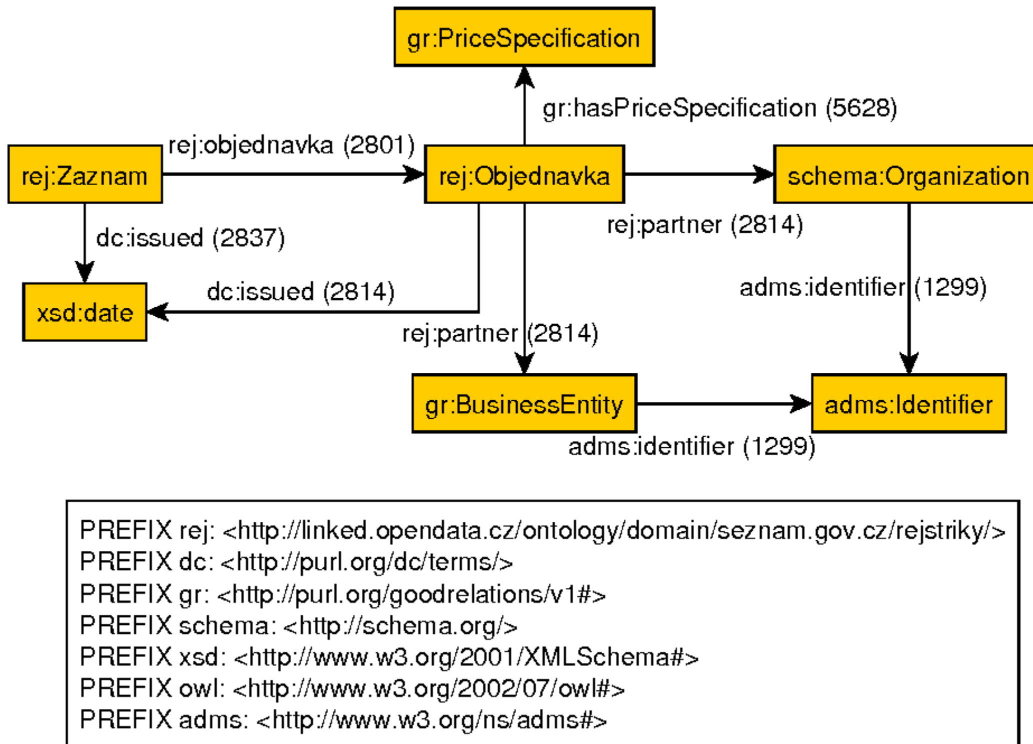

Figure 9: Descriptor funtion output[14]

3.1.1 Form model

Each module and function call is configured using a configuration form, which is represented as a question-answer tree structure, where the question can have zero or more subquestions, zero or more answers and some other metadata (e.g. label shown to the user and origin that allows to reference the triple the question was created from). The question-answer model diagram is shown in the Figure 16. The entire form is the root question which is,

[5] https://linked.opendata.cz/resource/dataset/seznam.gov.cz/rejstriky/objednavky
[6] https://linked.opendata.cz/

naturally, the root of the question tree structure. It then has wizard step subquestions (shown as tabs in the rendered form) that, due to the relatively simple structure of the configuration dialogue, are represented by a single entry. The actual configuration value questions are the leaves of the tree structure, in this case being the direct

3.1.2 Script Modularization Patterns

During the analysis of SPARQLMotion and SPipes scripts, several possible script modularization patterns were discovered (the screenshots are taken from the Topbraid Composer):

- **Function reuse**

  In case of function reuse, a SPARQL function is defined once and then imported and used in multiple scripts. In the Figure 10 the `form-ecc-lib:create-q` function is defined, which is then used in a SPARQL CONSTRUCT query, shown in the Figure 11.

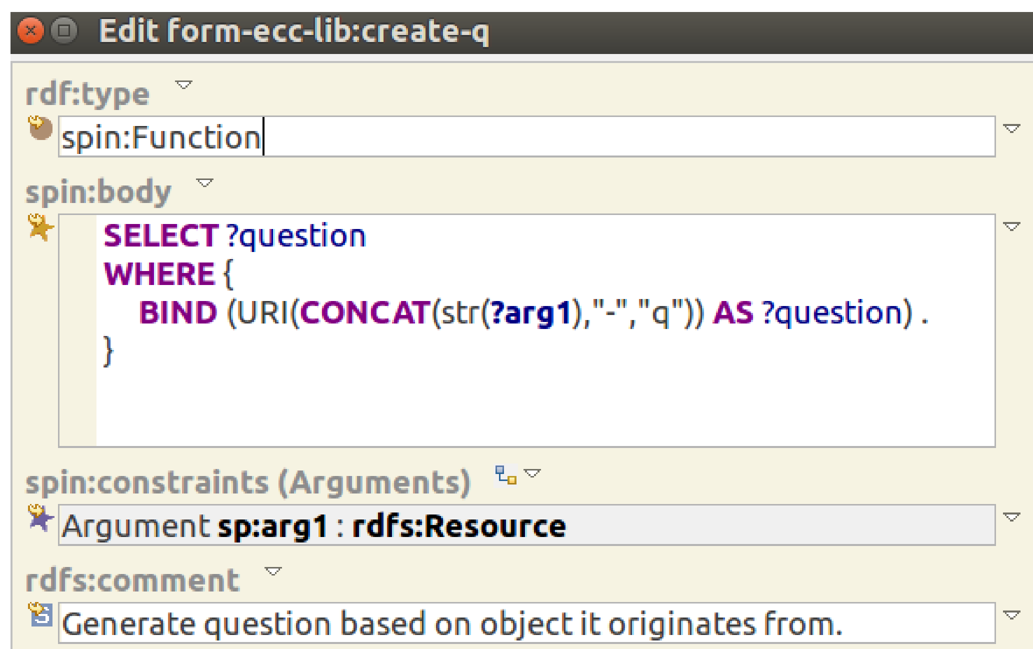

Figure 10: `form-ecc-lib:create-q` function definition

```
CONSTRUCT {
    ?s_q doc:has_related_question ?o_q .
}
WHERE {
    ?s a e-m:instance .
    ?s e-m:has-child-instance ?o .
}
    BIND(form-ecc-lib:create-q(?s) as ?s_q)
    BIND(form-ecc-lib:create-q(?o) as ?o_q)
}
```

Figure 11: `form-ecc-lib:create-q` function reuse

- **Module reuse**

  The module reuse scenario describes the case when certain module is defined once and then reused in multiple scripts for example by extending it. In the Figure 12 a module of type `sml:ImportFileFromURL` is described, which has two input parameters – `documentId` and `sheetId`. In the figure 13 a module, extending `sml:ImportFileFromURL` and creating a closure on the input parameters, is defined.

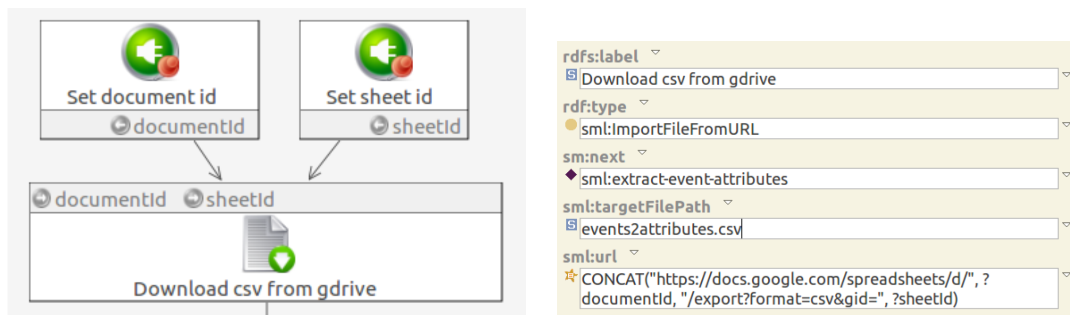

Figure 12: Module definition

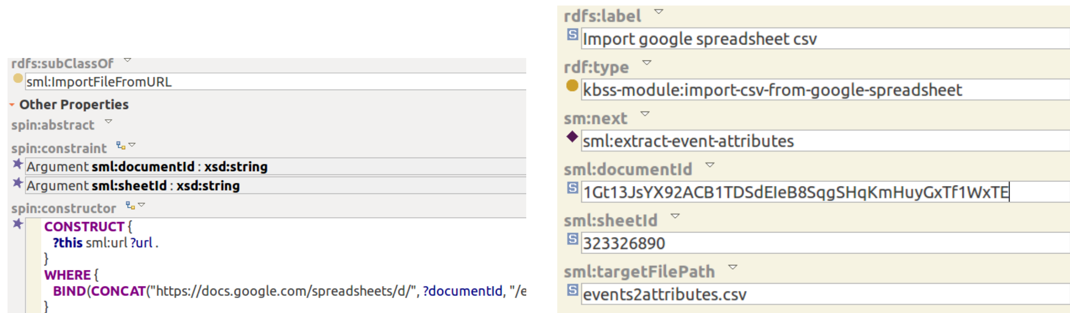

Figure 13: Module extension

- **Pipeline reuse**

  Pipeline reuse is using an entire pipeline definition or a part of it and including as a part of an another pipeline, possibly extending it with additional processing steps. The Figure 14 shows the pipeline later included as a part of an another pipeline, shown in the Figure 15.

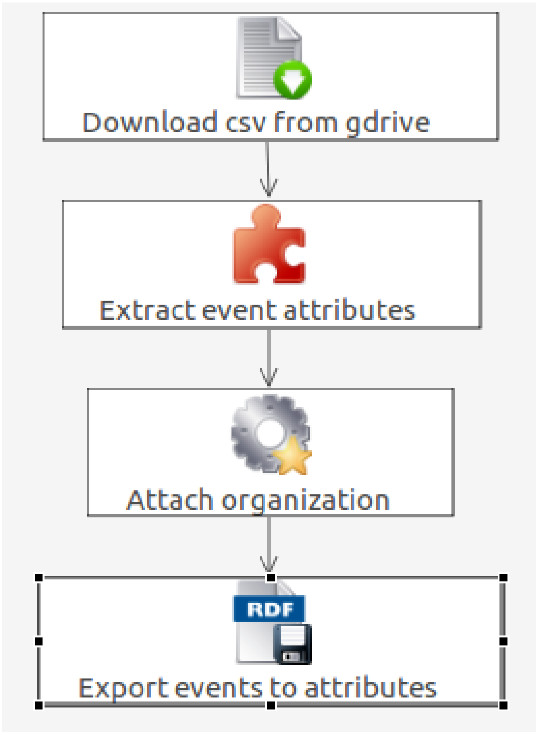

Figure 14: Pipeline definition

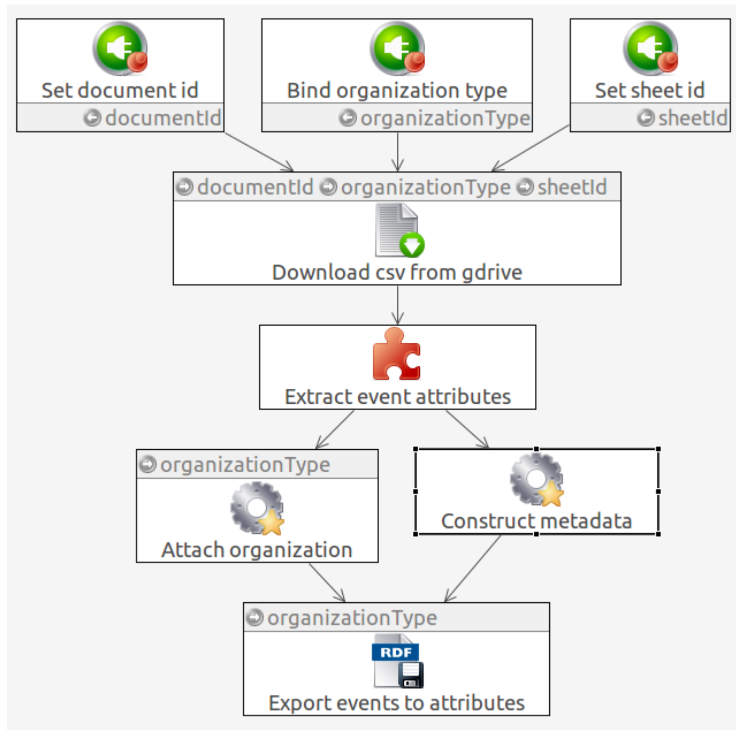

Figure 15: Pipeline reuse

Analysis of the use of those patterns in the `gacr16-model` revealed that frequence of the pipeline reuse pattern occurence was much higher, then the frequency of the other patterns. So the main pattern taken into accout during the editor requirements collection and design was the pipeline reuse.

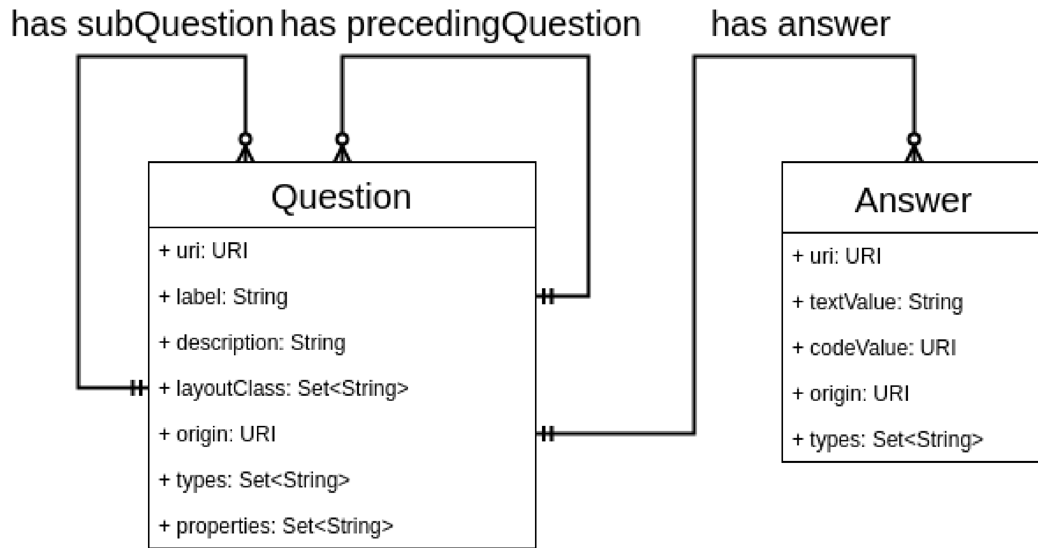

Figure 16: Question-answer model

3.2 Integration Diagram

The integration diagram, focused on high-level view on the application parts, is shown on Figure 17 with a description of individual parts following.

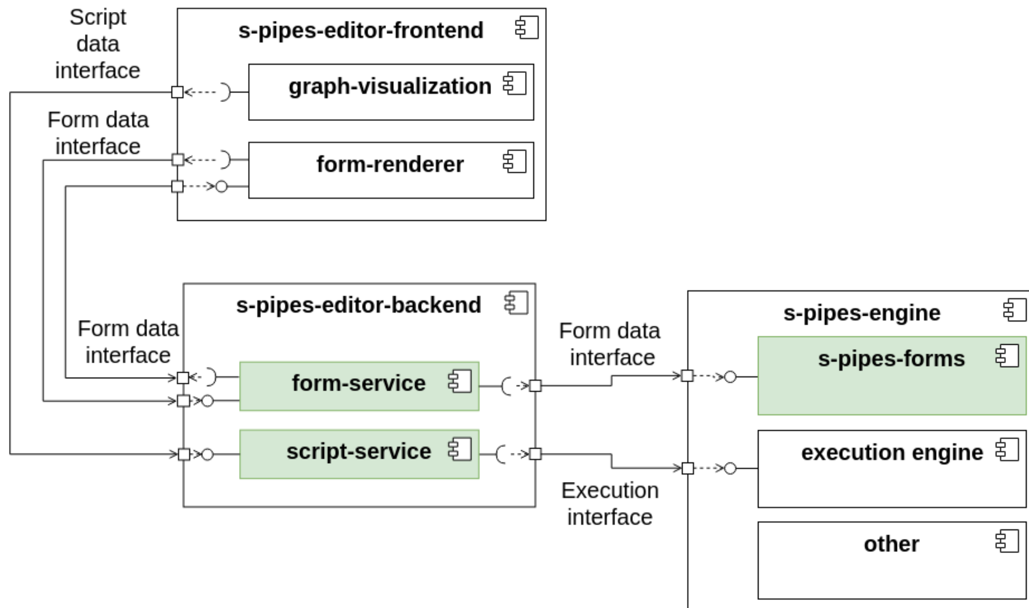

Figure 17: Integration diagram

3.2.1 SPipes Engine

SPipes Engine is a reference implementation of the SPipes language. In the scope of this project, it has two functions: generating form data for the configuration forms which is discussed in the Section 4.4.3 and processing function execution requests from the editor. Pipeline execution can be started in a way that all the data and metadata are logged into an RDF4J server repository, that can be queried for the information on the execution.

3.2.2 SPipes Editor Backend

SPipes Editor backend handles all the work on data reading, writing and transformation (except the form generation, mentioned above) and supplying data for the frontend to be presented to the user.

3.2.3 SPipes Editor Frontend

SPipes Editor frontend does all the UI work, presenting the user the graph representation of the script, module configuration and function call forms as

well as notifying user about the changes in the script currently being edited. All the data the fronted operates on is provided by frontend through either REST or Websocket API.

3.3 Use Case Diagram

Figure 18 shows the use case diagram for the SPipes Editor. Currently unsupported use cases are shown in *italics*.

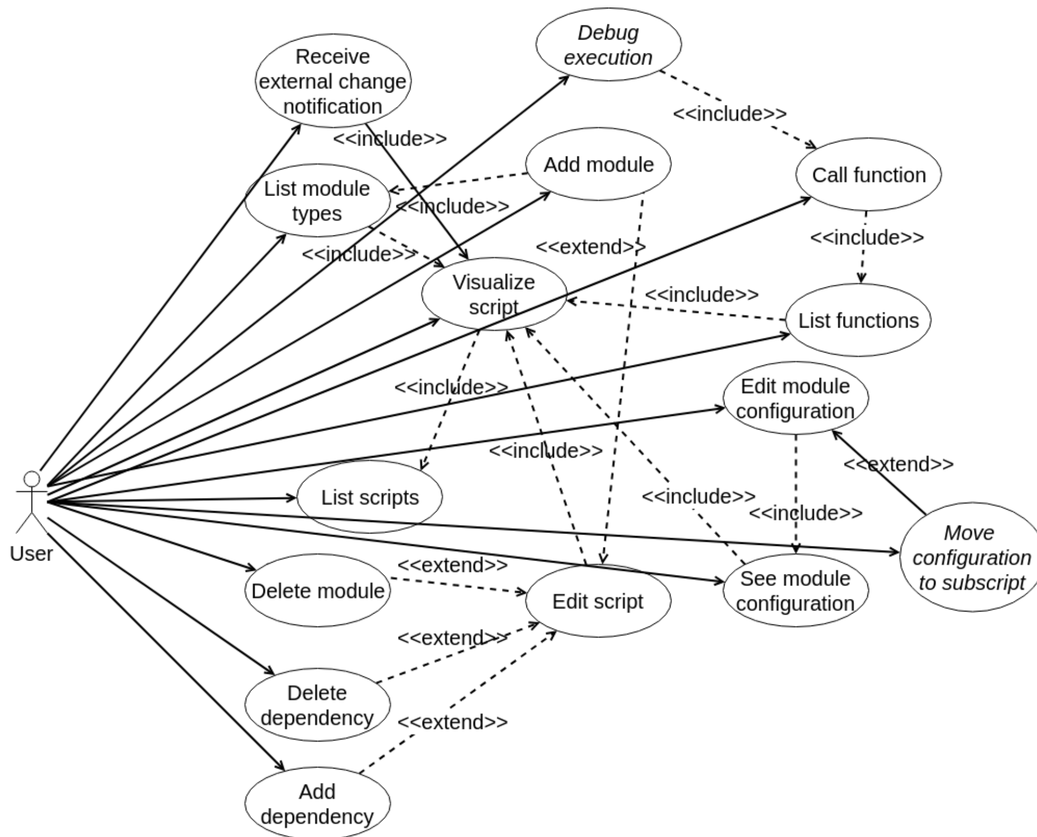

Figure 18: Use case diagram

3.4 Requirements

3.4.1 Functional Requirements

Functional requirements are separated by the relevant subsystem and graded using the MoSCoW method[12]:

M *Must have* requirements represent features critical for the project success.

S *Should have* requirements are important, but not as critical.

C *Could have* requirements are desired, but not necessary. Usually they represent features that improve user experience and can be implemented if time and resources permit.

W *Won't have* requirements represent least important features that can not be implemented at the given time due to the lack of time or resources.

- Editing

  FR1 **M** Visualize a script in form of a directed graph.

  FR2 **W** Create a new script.

  FR3 **C** Layout graph using multiple algorithms.

  FR4 **M** Alter graphs using a graphical interface.

  FR5 **S** Notify user of the changes in the script currently being edited.

  FR6 **M** Add/remove modules to a script.

  FR7 **M** Add/remove dependencies between modules to a script.

  FR8 **M** Configure modules using forms.

  FR9 **W** Move the entire module configuration or part of it to a specific subscript.

  FR10 **W** Visualize input/output module parameters.

  FR11 **C** Collapse parts of the graph respecting layout (for example, based on imports).

  FR12 **W** Add/remove script imports.

FR13 **S** Allow for the visual distinction between module types.

FR14 **S** Show the entire graph overview.

- Execution

FR15 **S** Execute scripts.

FR16 **C** Debug scripts.

FR17 **W** Visualize script execution process.

FR18 **W** Show the execution data and/or metadata.

FR19 **W** Autocomplete function call parameters, based on previous executions.

3.4.2 Non-functional Requirements

NR1 Allow for parallel graph edition in separate sessions.

NR2 Present the graph automatically layed out.

NR3 Notify user of the errors during the script visualization.

NR4 Be tested using automated testing.

NR5 Be suitable for the typical use cases (see Section 5.2).

3.5 Existing Solutions

There are several tools relevant to the subject, divided into two groups – editors and general-purpose graph visualization libraries.

While the graph visualization libraries review is necessary for the obvious reason of selecting the good basis for the SPipes Editor's UI, some existing graph editors are evaluated to better understand the graph editing process and define the features necessary for it to be efficient.

This section focuses on the ones relevant to the problem and defines the criteria for their evaluation.

3.5.1 Evaluation Criteria

There were several criteria considered during the research of existing tools and libraries. Main ones are:

- **Critical**

  1. **Automatic layout**
     Automatic layout is one of the main concerns regarding the functionality due to average scripts having tens and possibly hundreds of modules and therefore being very difficult to layout manually. The critical feature is a flow layout. An ability to change the layout direction (left to right, top to bottom, etc.) and support of multiple layout algorithms are desired features.
  2. **Collapsing parts of the graph respecting layout**
     Some use cases require parts of the graph to be collapsible without affecting layout.
  3. **Module parameter visualization**
     As modules can have input and output parameters, an ability to vizualize them is crucial.
  4. **Overall view of a graph**
     An ability to see the overall preview of a graph is important due to the necessity to orientate in very large graphs.
  5. **Module type icons**
     Modules have to be easily identifiable based on their types.

- **Nice-to-have**

  1. **License**
     Tool has to be licensed under the free license due to the likely need to adapt the tool for the specific applications.
  2. **Active development**
     The project should be in active development due to the need to receive bug fixes and possibly new features.
  3. **Visualization of node/edge state**
     As the SPipes Editor is meant to be capable of SPipes scripts execution, being able to visualize execution process is a useful feature.

4. **Documentation and code reusability**
   Codebase should be well structured and documented in order to be easily integratable and reusable.
5. **Technology**
   Technologies used have to be modern and actively maintained.

3.5.2 Editors

Under this section the editors in some way relevant to the topic are described. Each one is given a short annotation with a more detailed feature comparison shown in the feature matrix (Figure 19).

- Ontodia[7]

  Ontodia is a well documented and actively developed web-based tool for editing ontologies in form of a directed graph, partially licensed under the LGPL 2.1 (with more advanced features available in the proprietary version). The lack of pipeline support and absence of the way to add it due to a proprietary license limit its usefulness severely for the purpose of the SPipes Editor.

- LinkedPipes ETL[8]

  LinkedPipes ETL is a pipeline editor that features graph visualization. While providing an intuitive and user-friendly graphical interface and convenient means of form-based pipeline editing, it lacks many graph editor features like layout or module type icons. The other disadvantage is obvious lack of scalability of visualization.

- yEd Graph Editor[9]

  yEd Graph Editor is an advanced general-purpose graph editor, not suited for editing pipelines and having no SPARQLMotion support. The proprietary license further limits its use in the given context, as the necessary features can't be implemented without the access to the source code. Exists in the desktop and web-based versions.

[7]http://www.ontodia.org/
[8]https://etl.linkedpipes.com/
[9]https://www.yworks.com/

3.5.3 Graph Visualization Libraries

This section describes generic graph visualization libraries also providing a brief annotation, while the feature comparison is shown in the Figure 19.

- JointJS[10]

  JointJS is a popular JavaScript diagramming library, partially licensed under Mozilla Public License 2.0 (with more advanced features available in a paid proprietary version), that might be used as a graph editor library if needed.

- dagre-react[11]

  dagre-react is a JavaScript library that provides React components for graph visualization, using Dagre as a layout backend. It's very small and simple library that lacks any advanced features and is unlicensed.

- dagre-d3[12]

  dagre-d3 just like dagre-react uses Dagre as a backend utilizing D3 for rendering. It's officially unmantained and lacks advanced features.

- KlayJS-D3[13]

  KlayJS-D3 uses D3 for rendering and KlayJS[14] for layout. Simple visualization library that does not allow graph editing.

- vis.js[15]

  vis.js is a user-friendly visualization library with a predefined set of components. Lacks oriented graph support and any of the advanced features.

- ArborJS[16]

  Arbor is a graph visualization library providing a force-directed layout. Lacks any necessary features.

[10] https://jointjs.com
[11] https://github.com/osnr/dagre-react
[12] https://github.com/cpettitt/dagre-d3
[13] https://github.com/OpenKieler/klayjs-d3
[14] https://github.com/OpenKieler/klayjs
[15] https://visjs.org
[16] http://arborjs.org/

- Drakula Graph Library[17]

  Drakula Graph Library is a simple library for primitive graph visualization.

- Sigma js[18]

  Sigma is a JavaScript library dedicated to graph drawing, that is lacking advanced features.

- The Graph Editor[19]

  The Graph Editor is a JavaScript library built with React, that supports advanced features like thumbnail and parameter visualization. The library is well documented, actively developed and licensed under the MIT license.

3.5.4 Feature Matrix

Comparison between features supported by different libraries is shown on Figure 19.

[17]https://www.graphdracula.net/
[18]http://sigmajs.org
[19]https://github.com/flowhub/the-graph

	Ontodia	LinkedPipes ETL	yEd Graph Editor	JointJS	Dagre-React	Dagre-D3	KlayJS-D3	vis.js	ArborJS	Drakula Graph Library	Sigma js	The Graph Editor
Autolayout	✓		✓	✓	✓	✓	✓	✓				✓
Collapsing	**?**		✓									
Parameters			✓	✓								✓
Thumbnail	✓		✓									✓
Module type icons			✓									✓
Free license	∼	✓		∼		✓	✓	✓	✓	✓	✓	✓
Active	✓			✓				✓			✓	✓
State visualization		✓										✓
Documentation	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
Technology					✓							✓

✓ – library is compliant with the requirement for the given feature
? – supported in a proprietary paid version
∼ – library is partially compliant with the requirements for the given feature

Figure 19: Feature matrix

As seen from the feature matrix (Figure 19) and functional requirements (Section 3.4.1), the tools most suitable as a basis for the SPipes Editor are the yEd Graph Editor, which lacks extendability and flexibility due to its proprietary codebase, so none of the desired features can be added to it, and the The Graph Editor, which might have slightly reduced functionality, but has the source available for the audit and enhancement, with a strong active community standing behind it.

So the existing solution analysis showed that The Graph Editor from the Flowhub project would be the best library to base SPipes Editor on.

3.6 Debugging Proposal

The proposal for debugging a script was developed around the concept of execution ID. Once the user requests a function execution, a request with parameters is sent to the backend. A unique execution ID is then generated on the backend and attached to an execution request sent to the SPipes Engine. The execution ID is then returned to the fronted in a response. The client then subscribes for the execution notifications, sending the execution ID through the Websocket. Once there are execution changes, SPipes Engine sends a notification request to the editor backend. The updates on execution are then collected from the external repository (saving and queryng execution data and metadata is shown in the Figure 20) by the backend and propageted to the frontend through the Websocket. The sequence diagram of the process is shown in the Figure 21.

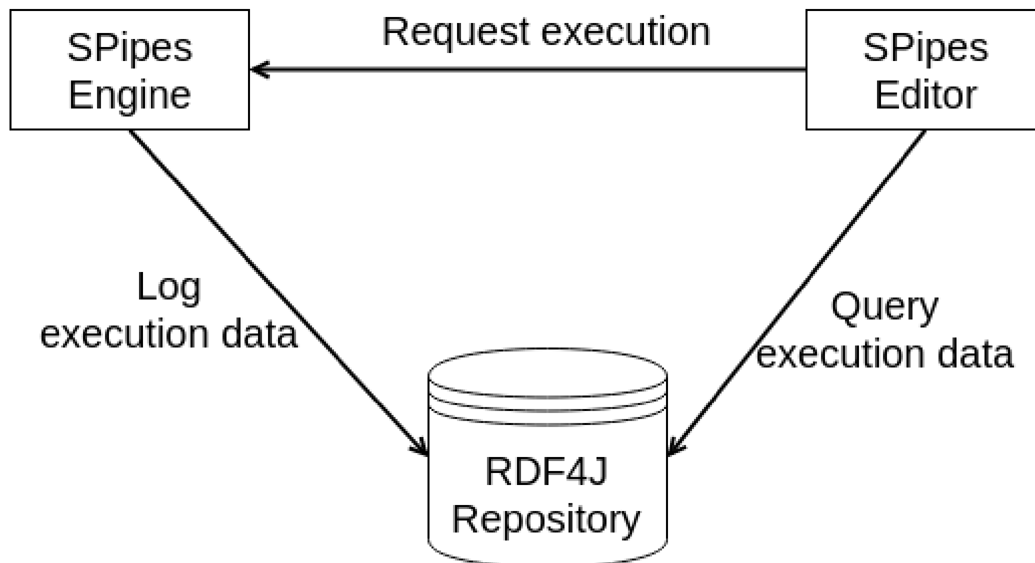

Figure 20: Execution data and metadata logging

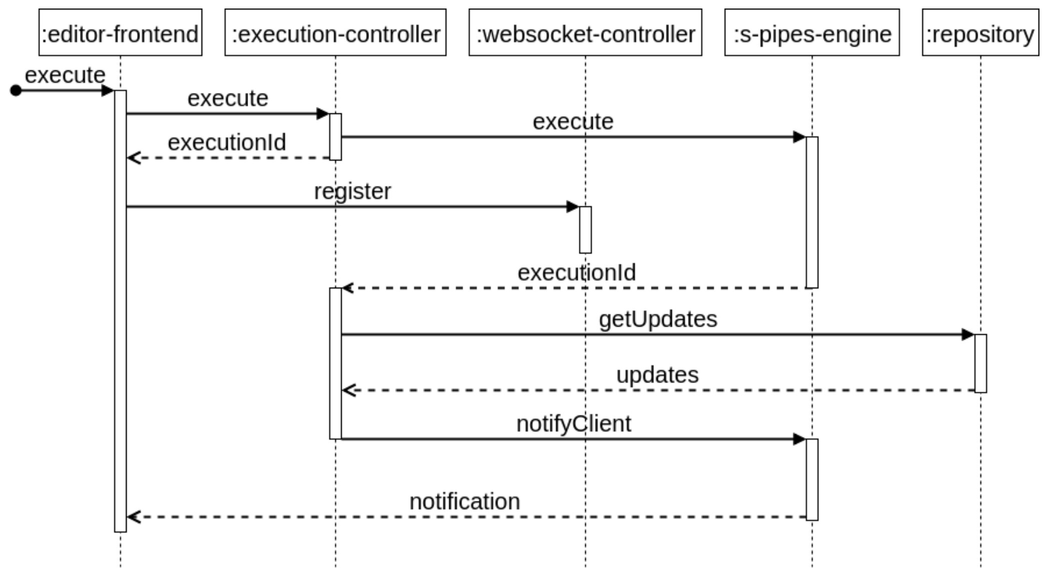

Figure 21: Execution sequence diagram

4 Implementation

This section describes the implementation, focusing on the topics, specific and unique for the given application.

4.1 Application Architecture

SPipes Editor is built as a typical layered client-server application, where each layer has its own role:

- Model layer contains the model and DTO classes, implemented in Java and annotated with JOPA annotations.
- Persistence layer is responsible for fetching data from the storage, files in this case.
- Service layer envelops all the data transformation and processing logic.
- REST controllers receive the client's requests and delegate them to the services.
- Websocket controller is used to actively send messages to the client.
- Client layer contains the client code, run in the user's browser and responsible for the entire UI.

The diagram, describing the architecture is shown in the Figure 22.

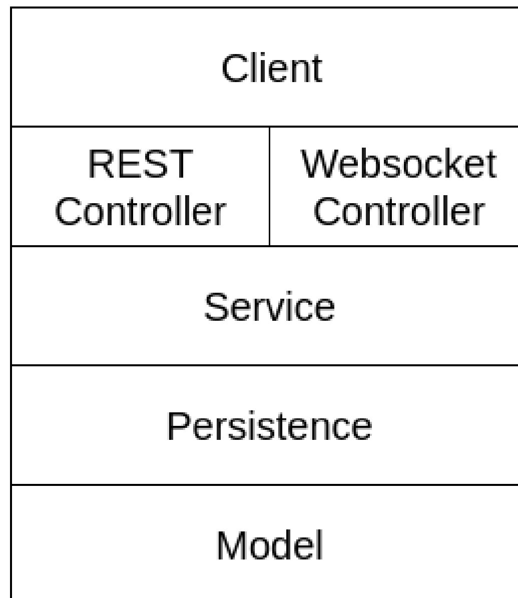

Figure 22: Application's layered architecture

4.2 Technology Stack

4.2.1 Scala

As SPipes Engine is implemented in Java and runs on JVM, the editor should be implemented in a programming language, also running on JVM. Scala was chosen because of the concise syntax, reach API, active development and possibility to use advanced concepts (like currying and implicits) while still being relatively easy to comprehend for an inexperienced programmer. Another reason is almost complete Java interoperability allowing to use Java libraries and frameworks with minimal boilerplate.

4.2.2 Spring

Spring framework is used for Dependency Injection and as a Web framework to create REST and Websocket APIs and also for automated testing. Spring is an industry standard and supports all the libraries necessary for semantic APIs.

4.2.3 Jena

The Java library used for ontology management used is Apache Jena, that provides a rich toolkit for ontology querying, serialization/deserialization and transformation, as well as adding and removing triples and managing imports (ontologies loaded from external files). There are two important concepts, relevant for the SPipes Editor:

- Model

  Model is a set of triples with some additional metadata like contexts. There are different implementation of the `Model` interface in Jena, the most relevant being the `OntModel`, that stores information about imported triples' origin and `InfModel`, capable of triple inference – creation of the new relationships, based on the information available[7]. Figure 23 shows the informal example of inference.

$$\left.\begin{array}{l} <\texttt{A}><\texttt{subClassOf}><\texttt{B}> \\ <\texttt{B}><\texttt{subClassOf}><\texttt{C}> \end{array}\right\} \xRightarrow{\text{inference}} <\texttt{A}><\texttt{subClassOf}><\texttt{C}>$$

Figure 23: Informal inference example

- Statement

  Statement is the Jena's look on triple that stores a reference to a model it originates from and provides some syntactic sugar over triple processing like fetching a typed value of the triple's object.

4.2.4 JOPA

Java Ontology Persistence API is the tool that enables loading ontological data as Java objects. For achieving this, the class definition as well as every class field is annotated with a corresponding URI value that binds data in semantic format to the entity class in a fashion, similar to JPA's `@Table` and `@Column` annotations. The URIs used to annotate entity classes are taken from the vocabulary, that can be automatically generated from the ontology

using a part of JOPA's toolkit – `OWL2Java`. An example of the JOPA entity is shown in the Figure 24.

```
@OWLClass(iri = Vocabulary.s_c_Modules)
public class Module extends AbstractEntity {

    @OWLDataProperty(iri = Vocabulary.s_p_label)
    private String label;

    @OWLObjectProperty(iri = Vocabulary.s_p_next)
    private Set<Module> next;

    @OWLObjectProperty(iri = Vocabulary.s_p_specific_type)
    private ModuleType specificType;

    @Types
    private Set<String> types;

// Constructors, getters, setters, equals, hashCode, ...
}
```

Figure 24: Example of a JOPA entity

4.2.5 React

React acts as a nexus of the application's frontend, providing vizualization components and tying different parts together while also mitigating some of the plain JavaScript disadvantages and being a mainstream well supported technology that is relatively easy to grasp with no prior experience.

4.2.6 SForms

JavaScript library SForms provides React components for vizualizing semantic question-answer structures as reactive forms, providing the user with a familiar interface for editing ontologies. In the SPipes Editor it is used to render module configuration and function call forms.

4.3 JSON-LD REST API

As the application is meant to integrate with the Semantic Web domain, a logical step to take was to make the public APIs compliant with the Semantic Web standards. The most reasonable way to achieving this being to reimplement REST API to make it accept and produce JSON-LD instead of the plain JSON[10] as it required minimal changes in the frontend of the application and had all the necessary mechanisms available from the JOPA toolkit.

All the changes necessary were:

- Register custom JSON-LD deserializer included in the JOPA toolkit.
- Add all the data about the entities being translated to/from JSON-LD to the ontology that vocabulary is generated from.
- Annotate the entity classes and their fields with the specific annotations provided by JOPA for them to be correctly serializable and deserializable.
- Rewrite the way objects' inner data is being accessed in JavaScript (mostly replace indices).
- Add a correct `@type` element to every request object send from JavaScript as it is required for the deserializer to know which entity class should be used for the deserialization.

4.4 Data Transformation

Due to the different formats and schemas of the source data, there is a necessity to transform it to be suitable for consumption. Two major aspects of the data transformation in the SPipes Editor are transformations from script data to graph and from module data to configuration form.

4.4.1 Model Generation

The common problem for both the script to graph transformation and configuration form generation is assembling the model. As mentioned before, scripts can import parts of other scripts (either locally from files or from the Web) so in order to get the complete information, all the imports

should be resolved and correctly appended to the model. Luckily, Jena provides its own solution for this problem, the `OntDocumentManager`, so the user only needs to take all the script files, map them to the URIs of ontologies they contain and pass the resulting map to the manager, that keeps all the complicated work under the hood. After that it can be queried for the model of the desired ontology by the ontology URI. To create an import closure, the `loadImports()` method must be called. Another useful feature of the ontology document manager is that it retains the ontology to file mapping that can be queried later for the purpose of finding the file a specific statement comes from. After the model is assembled, the transformation itself is done.

4.4.2 Graph Visualization

To translate the script, contained in the model, to the directed graph form, there is a sequence of steps required:

1. The model is queried directly for the module entries through JOPA using the SPARQL query.

2. The query shown in the Figure 7 is run to determine the most specific type of the module for the module to provide the most information to the user after being rendered as a graph node.

3. An instance of the `Node` class is created for each module, containing all the information relevant: module URI, label, types and the most specific type separately.

4. List of modules is traversed again to collect the dependencies. An instance of the `Edge` class is created for every dependency, which holds a reference to the source and target nodes.

5. An instance of the `View` class – the representation of the whole graph – is assembled from nodes and edges as well as the path of the file, containing the script.

6. The view is saved to an ontology storage and returned as a transformation result.

4.4.3 Configuration Form Generation

Each module should be configurable through a form, which is generated, based on the module data. As generation of the configuration form from the script data is a generally useful concept in SPipes, it was implemented as a part of SPipes Engine project. The sequence diagram of the configuration form creation process is shown in the Figure 25.

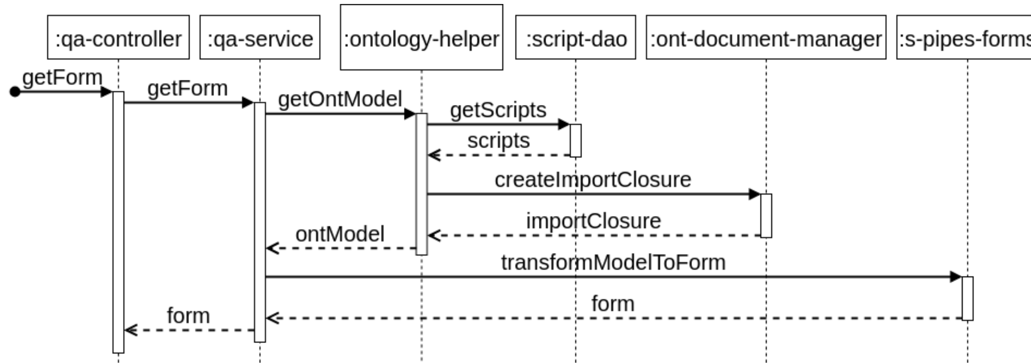

Figure 25: Form generation sequence diagram

The resulting form is represented by a question-answer model, described in the Section 3.1.1.

A module configuration form generation process (shown in the Figure 26) is the following:

1. Find the triples, describing the given module, in the script.
2. Create the root question, set its origin to the module's URI (to be able to locate the original module in case user changes the URI) and a layout class (necessary for the SForms to correctly render the question).
3. Create a wizard step question, set its label to show module type and origin to the module's URI (for the sake of completeness, it does not participate in lookup process) and add `section` and `wizard-step` layout classes for the sake of rendering.
4. Create a URI question with a constant origin and set the value of its answer to the actual module's URI.

5. Iterate through the triples, describing the module and create a leaf question from its predicate and an answer from the object, add the answer's original type to the question's metadata and set question and answer origins to values, deterministically generated from the triple.

6. If there are predicates in the module type definition absent from the module's description, create a leaf question for each of them the same way the module's questions are created, but leaving the answer blank.

7. If there was no question about label generated during the triples travelsal, create one.

8. Order the questions so that the URI question is on top, below it is the label question and all the rest are below that.

9. Append all the leaf questions to the wizard step question.

10. Append the wizard step question to the root question.

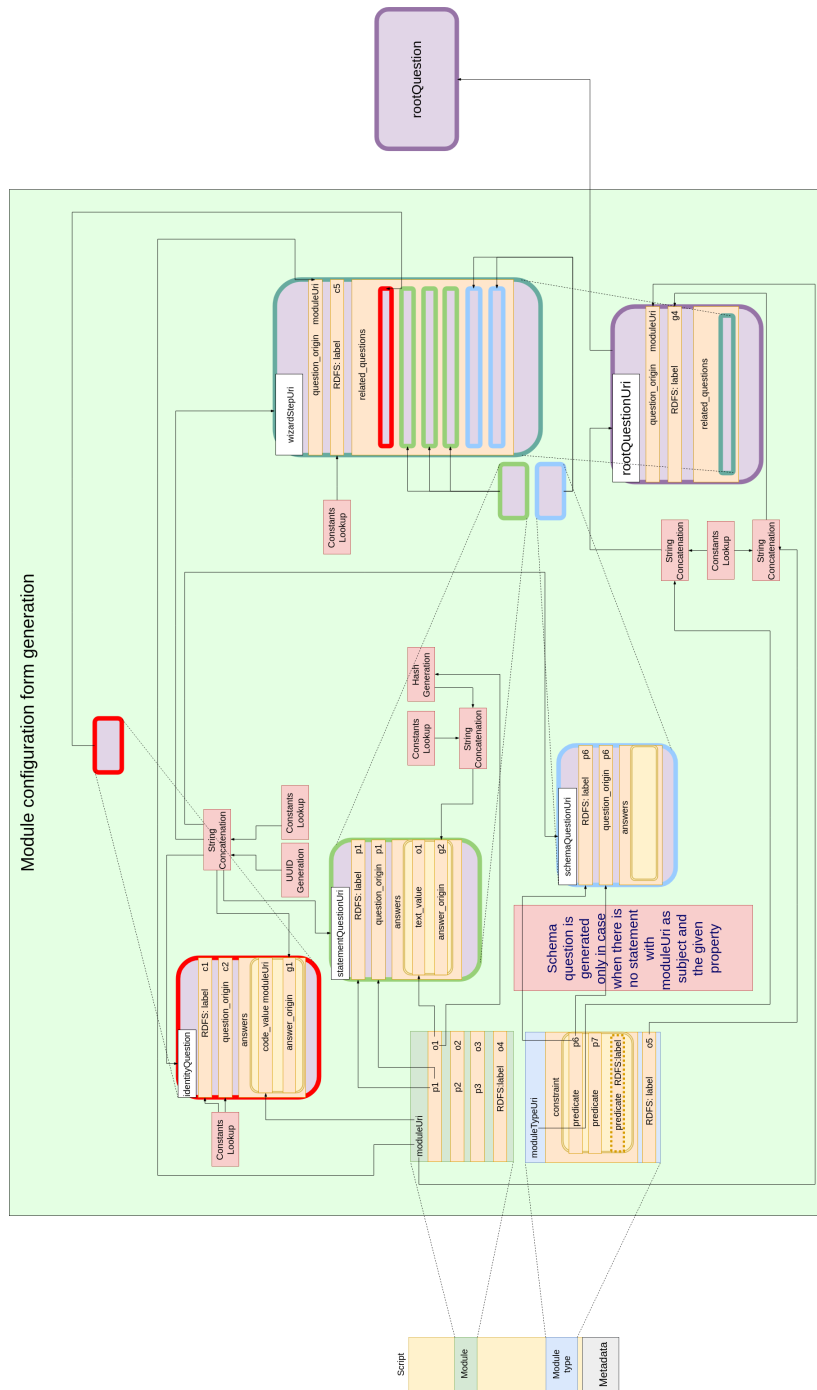

Figure 26: Module configuration form generation

Some aspects of the diagram, shown in the Figure 26, are:

- Identity question (red frame)

  Identity question is a question about the module's URI.

- Statement question (green frame)

  Statement questions are questions, generated from the concrete module definition.

- Schema question (blue frame)

  Schema questions are generated only for the statements, absent from the concrete module.

- Wizard step question (dark green frame)

  Wizard step question is an envelope to contain all the statement and schema questions.

- Root question

  Root question represents the entire form and is a wrapper around a wizard step question.

The example module type definition in the Turtle format, the part of the transformed form in JSON-LD and the configuration form for creating the module of that type are shown in the Figures 27, 28 and 29 respectively (ellipsis marks the omitted part of the document).

```
kbss-module:deploy
    rdf:type sm:Module ;
    spin:constraint [
        rdf:type spl:Argument ;
        spl:optional "false"^^xsd:boolean ;
        spl:predicate km-rdf4j:p-rdf4j-context-iri ;
        rdfs:comment "Context IRI" ;
    ] ;
    spin:constraint [
        rdf:type spl:Argument ;
        spl:optional "false"^^xsd:boolean ;
        spl:predicate km-rdf4j:p-rdf4j-repository-name ;
        rdfs:comment "Repository name" ;
    ] ;
    spin:constraint [
        rdf:type spl:Argument ;
        spl:optional "false"^^xsd:boolean ;
        spl:predicate km-rdf4j:p-rdf4j-server-url ;
        rdfs:comment "Server URL" ;
    ] ;
    spin:constraint [
        rdf:type spl:Argument ;
        spl:predicate km-rdf4j:p-is-replace ;
        spl:valueType xsd:boolean ;
        rdfs:comment "Replace context flag" ;
    ] ;
    sm:icon "database" ;
    rdfs:label "Deploy" ;
.
```

Figure 27: Module type definition in the Turtle format

```
{
  "@context": {
    "form": "http://onto.fel.cvut.cz/ontologies/form/",
    . . .
  },
  "doc:has_related_question": {
    "@id": "doc:question-681f0e57-d9af-49e2-9ab4-c10aa26563b9",
    "@type": "doc:question",
    "doc:has_related_question": {
      "@id": "doc:question-f2c799de-c79e-4de2-a807-0ad03ed1a15d",
      "@type": "doc:question",
      "doc:has_answer": {
        "@type": "doc:answer"
      },
      "doc:has_related_question": [],
      "form:has-preceding-question": {
        "@id": "doc:question-d7fdc37e-a243-49a7-ba23-b41fb9e20b3f"
      },
      "form:has-question-origin":
        "http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/
            p-rdf4j-repository-name",
      "dce:description": "Rdf4j repository name",
      "rdfs:label":
        "http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/
            p-rdf4j-repository-name"
    },
    "form-layout:has-layout-class": [
      "wizard-step",
      "section"
    ],
    . . .
  },
  "form-layout:has-layout-class": "form",
. . .
}
```

Figure 28: Part of the transformed module creation form

Configuration

Module of type Deploy (http://onto.fel.cvut.cz/ontologies/lib/module/deploy)

Module of type Deploy (http://onto.fel.cvut.cz/ontologies/lib/module/deploy)

URI

http://onto.fel.cvut.cz/s-pipes-editor/23a8ad

http://www.w3.org/2000/01/rdf-schema#label

http://www.w3.org/2000/01/rdf-schema#label

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-is-replace

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-is-replace

Is replace

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-rdf4j-context-iri

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-rdf4j-context-iri

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-rdf4j-repository-name

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-rdf4j-repository-name

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-rdf4j-server-url

http://onto.fel.cvut.cz/ontologies/lib/module/rdf4j/p-rdf4j-server-url

Save Cancel

Figure 29: Module creation form

The process of the function call form generation is similar, but simplified in that:

- There are only function definition leaf questions and therefore no answers.

- No origins are created as there is no need to track answers to the original triples.

- There's no label question as function call does not require a label.

The reverse transformation happens in the similar fashion, looking up the triples based on origins and replacing them with the ones, specified by the question and the answer. After all the triples are replaced, the entire module's URI is changed to the value of the URI question's answer.

4.5 File System Synchronization

One of the main requirements for the SPipes Editor is the ability for parallel edition of the same file which brings a synchronization problem between different sessions as well as between an editor user and someone editing the file externally. After considering several possibilities like only allowing one session to open the file for editing and restricting others to read-only mode or automatically reloading the view every time the file is changed, the final solution of the problem was to notify the user about the fact that changes occured in the file currently being edited and let user decide whether to update the view now or later.

To achieve that the combination of the `WatchService` and Websocket is used the following way:

1. During the application start the `WatchService` object is created. All the directories configured as script locations are recursively traversed to add each subdirectory to the watch service. After that the watch service starts waiting for the file system events.

2. Once user opens a view, a message, containing the name of the file being edited, is sent through Websocket to the backend where it's saved to a map from files being edited to a list of sessions associated with them.

3. When a change occurs on a file, a file system event is emitted. If the file in question is located in a directory watched by the watch service, the event is processed by the application. The target file path is extracted from the event and the mapping is checked if there are any sessions associated with the file. If there are, each one is sent a message through Websocket.

4. When frontend receives a Websocket message, the prompt is shown to the user where to reload the view or to ignore it. If user chooses to reload, the view is reloaded and the data consistency is restored. If the user ignores the prompt, he takes a risk of working on the outdated data.

The sequence diagram of the process is shown in the Figure 30.

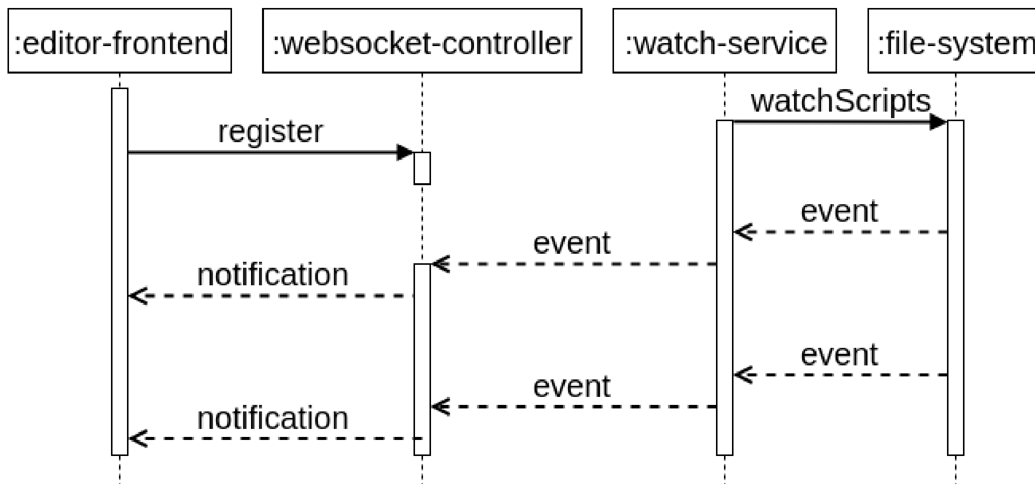

Figure 30: File system synchronization

So the application provides user with all the information he needs to make a right decision for the given situation while also not enforcing any specific behavior or making the choice for him.

There is an issue though in that user, who has caused the change, receives a notification too. This can be fixed by attaching some identifier (session ID for example) to every change request and ignoring the session, associated with that identifier, during notification. It has not been implemented yet though.

4.6 Implementation Issues

There were several issues during the implementation caused either by the bugs in the libraries or by the Scala-Java interoperability:

- JOPA vocabulary

  Vocabulary is generated as a Java class, containing constants, representing URIs[9]. This lead to an issue with Scala and Java interoperability in that Scala does not recognize Java's `static final` variables as constants and does not allow to use them as annotation parameter values leading to the necessity to implement all the entity classes in Java, effectively adding to the overall line count and amount of boilerplate.

- Jena OntDocumentManager cache

  There is an issue with Jena's `OntDocumentManager` in that it caches the imported models in a weird way not allowing to clear the cache in an intuitive way and therefore providing the outdated data from imported ontologies until the entire application restart. The issues has not been fixed to date.

- JOPA and Scala collection wrappers

  Another issue between JOPA and Scala was the way JOPA handled collections. Scala has its own collection API, that contains a mechanism for Java interoperability that wraps Scala collections in the wrappers Java can understand[11]. As Java does not support variation, there was a type checking issue in that JOPA's `CollectionInstanceBuilder` did not recognize the Scala collection wrappers being instances of the `Collection` so the workaround with the additional type checking and wrapping had to be applied.

4.7 Application User Interface

Main window of the resulting editor is show in the Figure 31. The main part of the window is occupied by the graph view. Bottom right features a thumbnail. On the top right there is a layout panel with a button allowing to duplicate the view in a new tab. Top left contains module creation and function call controls.

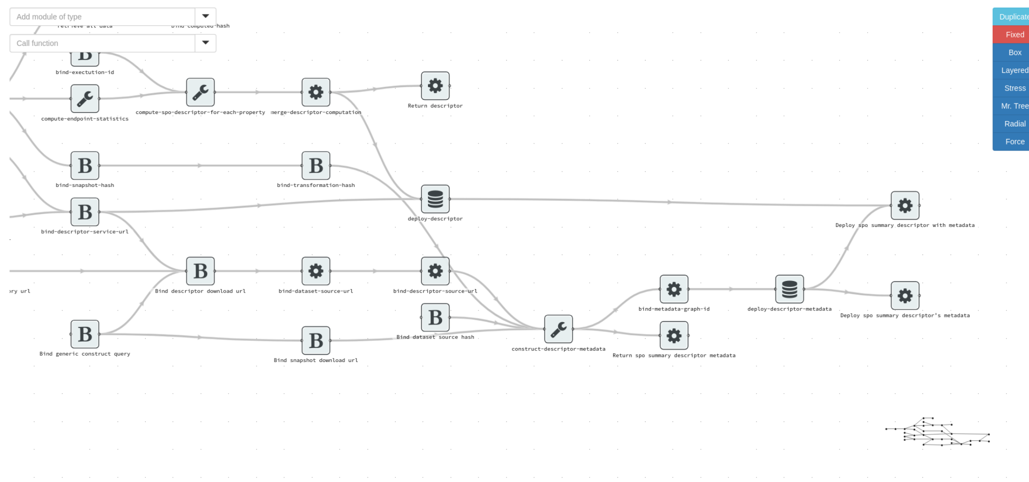

Figure 31: Main editor window

5 Testing

5.1 Test Model

As stated before 3.1 the implementation and testing was done using the `16gacr-model` (will be attached to the application source code) with the module type definitions from `s-pipes-modules`.

5.2 Scenarios

This section describes the typical scenarios SPipes Editor is meant to handle and evaluation of users' ability to use it for the specific tasks.

The user focus group consisted of three people of different levels of familiarity with RDF, SPARQL and SPipes. The users were given a brief intro about the SPipes and the Editor's capabilities and asked to perform the following tasks:

- Find the value of `outputVariable`, bound by module `bind-snapshot-service-url` in file `descriptor/spo-summary/transformation.ttl`.

  Script lookup was a bit confusing to all the users due to the suboptimal script selection screen layout, shown in the Figure 32. Occasionally all

the users discovered that the search feature, integrated in the browser, works and effectively decreased the time needed they needed to find a specific script in the list.

16gacr-model/ckan/script.sms.ttl 16gacr-model/dataset-descriptor-model.ttl 16gacr-model/ddo-header.ttl 16gacr-model/ddo-ufo.ttl 16gacr-model/descriptor/config.ttl
16gacr-model/descriptor/config/defs/config.ttl 16gacr-model/descriptor/data-properties/descriptor.ttl 16gacr-model/descriptor/data-properties/script.sms.ttl
16gacr-model/descriptor/descriptor-info.sms.ttl 16gacr-model/descriptor/metadata.ttl 16gacr-model/descriptor/pipeline-modules.sms.ttl 16gacr-model/descriptor/script.ttl
16gacr-model/descriptor/spo-summary-with-marginals/definition.ttl 16gacr-model/descriptor/spo-summary-with-marginals/model.ttl
16gacr-model/descriptor/spo-summary-with-marginals/transformation.sms.ttl 16gacr-model/descriptor/spo-summary-with-marginals/transformation.ttl 16gacr-model/descriptor/spo-summary/definition.ttl
16gacr-model/descriptor/spo-summary/description.sms.ttl 16gacr-model/descriptor/spo-summary/lib.sm.ttl 16gacr-model/descriptor/spo-summary/lib.spin.ttl
16gacr-model/descriptor/spo-summary/model.ttl 16gacr-model/descriptor/spo-summary/parameters.ttl 16gacr-model/descriptor/spo-summary/transformation.sms.ttl
16gacr-model/descriptor/spo-summary/transformation.ttl 16gacr-model/descriptor/temporal-v1/definition.ttl 16gacr-model/descriptor/temporal-v1/model.ttl
16gacr-model/descriptor/temporal-v1/transformation.sms.ttl 16gacr-model/descriptor/temporal-v1/transformation.ttl 16gacr-model/descriptor/temporal/old/descriptor.ttl
16gacr-model/descriptor/temporal/old/script.sms.ttl 16gacr-model/doc/dataset-descriptor-model/ontology.ttl 16gacr-model/doc/dataset-descriptor-model/provenance/provenance-en.ttl
16gacr-model/experiments/18gacr-journal-experiment-config.ttl 16gacr-model/experiments/2018-04-26-experiment-compute-spo-summary-statistics-old-datasets/config.ttl
16gacr-model/experiments/2018-04-26-experiment-compute-spo-summary-statistics/config.ttl
16gacr-model/experiments/2018-04-26-experiment-compute-spo-summary-with-marginals-statistics/config-loading-defs-statistics.ttl
16gacr-model/experiments/2018-04-26-experiment-compute-spo-summary-with-marginals-statistics/config.ttl
16gacr-model/experiments/2018-05-02-experiment-deploy-spo-with-metadata-lod-statistics/config.ttl s-pipes-modules/module.sms.ttl s-pipes-modules/sm-module-adapter.sms.ttl
s-pipes-modules/sm/functions-afn.ttl s-pipes-modules/sm/functions-fn.ttl s-pipes-modules/sm/functions-smf.ttl s-pipes-modules/sm/sparqlmotionlib-core.ttl s-pipes-modules/spin-function.spin.ttl

© Yan Doroshenko & KBSS FEL ČVUT v Praze, 2018 Version: 0.1 Production

Figure 32: Script selection screen

The mechanism to open the module configuration form (shown in the Figure 33) where the desired value could be found was also not intuitive for the users.

- Alter the label of module `bind-descriptor-type` in file `descriptor/spo-summary/transformation.ttl`.

Figure 33: Module configuration form

The module configuration scenario worked out as intended.

- Create a new module of type `Bind with constant`.

  The module creation being practically identical to the module configuraion also worked out too.

- Call function `compute-spo-summary-descriptor-metadata` with any parameters from file `descriptor/spo-summary/description.sms.ttl`.

The function call scenario was disturbed by the module type list's placeholder that distracted the user (Figure 34).

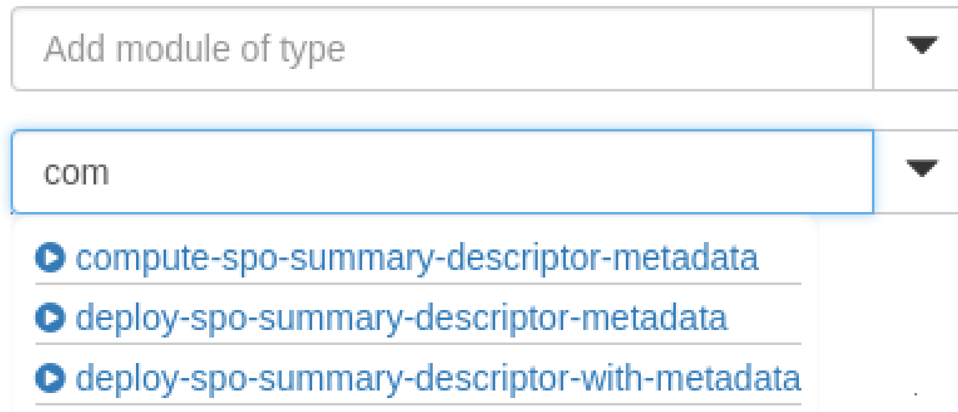

Figure 34: Function call selection

- Create dependency from `bind-execution-id` to `bind-target-rdf4j-server-url` in file `descriptor/spo-summary/transformation.ttl`.

  Due to the lack of familiarity with the editor, the dependency creation took some time to figure out.

- Delete module `bind-execution-id` from file `descriptor/spo-summary/transformation.ttl`.

  Module deletion was trivial as the button is located in the same menu that spawns the module configuration form (Figure 35).

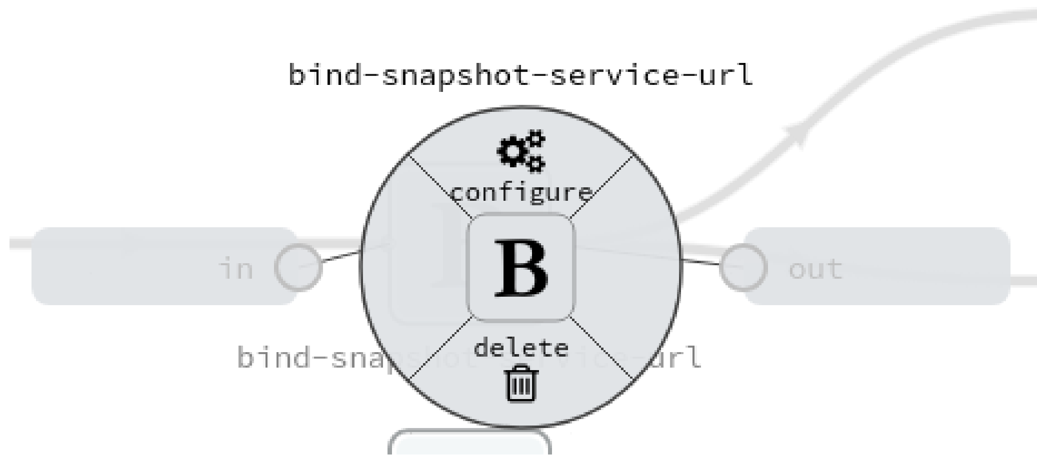

Figure 35: Module menu

- Delete dependency from `bind-execution-id` to `bind-target-rdf4j-server-url` in file `descriptor/spo-summary/transformation.ttl`.

  Dependency deletion is identical to the module deletion and thus was easy for the users to execute.

- Layout graph in file `descriptor/spo-summary/transformation.ttl` using Mr. Tree algorithm, duplicate the view and compare the duplicate with the original one.

  The layout scenario was succesfully done by the users, the only complaint being the absence of the current layout indication.

- Check if there are any modules in file `descriptor/config.ttl` and if there are any modules in `module.sms.ttl`.

  The user had to only find and open the correct script to be presented with a message about the absence of module types or modules so the scenario was succesful.

The user testing of the application showed that the main issue of the user interface is the user's lack of experience which can be considered a success.

There also were several other notes on usability from the users:

- There should be a possibility to see a current module configuration (for example, show read only configuration form).

- Module configuration form should close on the Escape key press.
- Going back to the script selection screen by using the browser's Back feature is unintuitive for the single-page application.
- Module icons as well as script entries can be extended to show much more useful information.
- There should be an indication of the script currently edited.

Not all of those features can be easily implemented as some of them require rewriting libraries, so each feature will be evaluated during the future development and the decision will be made if it is worth implementing.

5.3 Automated Code Testing

Due to the issues with the automated code testing, described in detail in the Section 5.3.2, the amount of work needed for MVP implementation and the need to meet the deadline, automated code testing was given less of a priority, being mostly replaced by manual testing, which was necessary in any case. Main focus of the automated testing was done on the service methods, while integration testing was mostly done manually.

5.3.1 Statistics

Some of the important statistics are shown in the Figure 36. During the class coverage statistics collection, classes, where automated testing is not applicable (enumerations, configuraion classes) were ignored.

	SPipes Editor	**SPipes Forms**
Tests written	10	16
Tests run	10	3
Tests ignored	0	13
Code coverage, classes	65%(19/29)	33%(1/3)
Code coverage, methods	24%(75/309)	15%(5/32)
Code coverage, lines	16%(147/871)	5%(14/265)

Figure 36: Some test statistics

5.3.2 Issues

Due to the use of Java libraries and APIs as well as Scala's focus on functional programming paradigm, application implementation became a mix of Object Oriented and functional code, which resulted in issues in automated testing. Main ones are the following:

- Mutable state

  One of the main advantages of pure functional programming is the absence of mutable state, which leads to deterministic results of all the actions (except the final IO stage). Mixing in the Object Oriented paradigm brings mutable state to the table therefore taking away the detereministic nature of the function calls and adding the need to consider the internal state of the objects.

- Complex configuration

  Some of the libraries used (e.g. Spring, JOPA) require complex configuration to function which takes time, adds boilerplate and another possibilty to make a mistake.

- Rigid testing tools

  Existing testing tools are aimed on testing either pure functions with no side effects and therefore incompatible with Object Oriented-style code, or classes, objects and methods, lacking support for more advanced functional programming concepts like currying.

- Dependency on the file system

  Many features of the editor are dependent on the file system (e.g. reading/writing files, traversing directory structure), sometimes leading to the need to either provide a part of directory structure or create quite complex testing data for the ability to test the given feature.

6 Comparison With Topbraid Composer

Topbraid Composer is a complex general purpose RDF editing suite, featuring a reach toolkit of diverse instruments suited for different tasks. However, being a general purpose editor, it's not optimized for SPARQLMotion and therefore has limited use for SPipes as well. SPipes Editor is not meant

to replace it in all the possible use cases, but rather to provide functionality, similar to the graph editor, integrated in the TBC[20], to complement the additional ontology or text editor. Therefore only comparison of the features relevant to the SPipes Editor will be mentioned in this Section.

- Functionality

  The aspects in which the SPipes Editor is superior to the TBC's graph editor are:

  - Multiuser support

    As TBC is a monolithic offline desktop application, it allows only one session to be run and therefore does not support sharing data or between users or parallel editing.

  - Module configuration copying

    Copying configuration values from one module to another is a typical scenario editor has to handle. However, TBC does not allow the used to simultaniously open multiple module configuration forms. In SPipes Editor it is possible to achieve that by the view duplication feature.

  However, there are some important advantages TBC has over SPipes Editor – for instance, module parameters visualization and collapsing parts of a graph respecting layout.

- Usability

  Regarding usability, SPipes Editor is better suited for the editing of large graphs due to the more efficient use of the screen space, achieved by minimal borders and toolbars. More intuitive controls behavior (drag-and-drop of module type to create a module instead of clicking on it and clicking on the main canvas, right click to show the menu instead of double click) also has positive effect on the overall user experience. SPipes Editor also supports mobile devices and touch input.

- Design

  SPipes Editor features a more modern responsive design achieved by using scalable components allowing to zoom the view in or out keeping

[20] Here and later in the text TBC refers to the Topbraid Composer Maestro edition

the most important information while also preventing the view from being cluttered.

More structured feature comparison is shown in the Figure 37.

	TBC	**SPipes Editor**
Script creation	✓	
Script visualization	✓	✓
Collapsing parts respecting layout	✓	
Multiple layout algorithms	$\sim$	✓
Script editing	✓	✓
External change notification	✓	✓
Module configuration	✓	✓
Module configuration copying		✓
Script execution	$\sim$	✓
Execution process visualization		?
Script debugging		?
Move configuration to a subscript		?
Show multiple configurations		✓
Visualize module parameters	✓	?
Show the graph overview	✓	✓
Show execution history		?
Multiple user editing		✓
Parallel editing editing		✓
Mobile devices support		✓

✓ – Fully supported $\sim$ – Limited support
? – Designed, but not yet implemented

Figure 37: Comparison between TBC and SPipes Editor

7 Conclusion

7.1 Project Goals Fulfillment

- Review existing graph visualization libraries and graph editors, based on specific requirements, described in Section 3.5.1

  The main result of the review is the feature matrix, shown in the Figure 19, that allowed to select the most suitable library to serve as the base of the editor.

- Analyze possible patterns to modularize SPipes scripts

  There were several possible script modularization patterns discovered, which are described in the Section 3.1.2.

- Define use cases for editing and debugging the scripts

  The typical use case analysis spawned several scenarios SPipes Editor should be able to handle, which are described as user test cases in the Section 5.2.

- Implement the editor for SPipes scripts, based on the analysis results

  The application implemented meets all the critical requirements, set during the analysis phase of the project, as well as some of the less crucial requirements.

- Compare the implemented editor with the Topbraid Composer SPARQLMotion editor

  As Topbraid Composer is a complex ontology editing suite, only the features, relevant to the SPipes Editor were taken into consideration. Those features are compared in the Section 6.

- Test the editor on specified use cases

  The specified use cases, as well as the feedback received from users during the testing, are described in the Section 5.2.

As all the project goals are fully or at least for the most part fulfilled, the project can be considered a success.

7.2 Future Development

The three issues future development has to address are the following:

- Fixing bugs

  As application is still in the phase of active development, there are implementation problems and bugs that have to be taken care of before the production release.

- Features

  There are still several important features, according to the requirements (Section 3.4.1) that have not been implemented due to the lack of time, that can be added in the future. For instance, collapsing parts of the graph respecting layout and module parameter visualization.

- Testing

  As stated in the Section 5 there is a plenty of work to be done on the automated code testing, that would benefit the long-term health of the application.

8 Installation Guide

8.1 Prerequisites

For installing the application from source, Apache Maven[21] and Node Package Manager[22] are necessary, as well as an application server (e.g. Apache Tomcat[23]).

8.2 SPipes Editor

To install the application, a configuration needs to be done. In the file `src/main/resources/config.properties` set the `scriptsLocation` property value to point to the root directory, containing scripts (for example, the included `16gacr-model` directory); for function calls to work set the `executionEndpoint` property value to point to the SPipes Engine execution endpoint (web service address). After configuration is done, run `mvn clean package -P production` from the root source directory and deploy the `target/s-pipes-editor.war` to your application server of choice.

8.3 Local SPipes Engine

To install a local instance of SPipes Engine, go to the `semantic-pipes-web` directory, included with the application sources and run `mvn clean package`. Deploy the resulting web archive from `target/semantic-pipes-web-0.1.0.war` to an application server.

[21] https://maven.apache.org/
[22] https://npmjs.com
[23] https://tomcat.apache.org

9 References

[1] Grigoris Antoniou, Frank van Harmelen, *A Semantic Web Primer*, 2nd edition, The MIT Press, Massachusetts, 2008.

[2] Guus Schreiber, Yves Raimond, *RDF 1.1 Primer*, `https://www.w3.org/TR/rdf11-primer/`, 2014

[3] Eric Prud'hommeaux, Gavin Carothers, *RDF 1.1 Turtle*, `https://www.w3.org/TR/turtle/`, 2014

[4] David I. Lehn, *JSON-LD Primer*, `https://json-ld.org/primer/latest/`, 2017

[5] Steve Harris, Andy Seaborne, *SPARQL 1.1 Query Language*, `https://www.w3.org/TR/sparql11-query/`, 2013

[6] Holger Knublauch, *SPARQLMotion*, `http://sparqlmotion.org/` 2010

[7] Graham Klyne, Jeremy J. Carroll, *Resource Description Framework (RDF): Concepts and Abstract Syntax*, `https://www.w3.org/TR/rdf-concepts/` 2014

[8] Miroslav Blaško, Petr Křemen, *SPipes*, `https://kbss.felk.cvut.cz/web/kbss/s-pipes`

[9] Petr Křemen, Zdeněk Kouba, *Ontology-Driven Information System Design*, IEEE Transactions On Systems, Man, And Cybernetics—Part C: Applications And Reviews, Vol. 42, 2012

[10] Markus Lanthaler, Christian Gütl, *On using JSON-LD to create evolvable RESTful services*, Proceedings of the Third International Workshop on RESTful Design, ACM, 2012

[11] Cay Horstmann, *Scala for the Impatient*, Pearson Education, 2012

[12] Kevin Brennan, *A Guide to the Business Analysis Body of Knowledger*, 2nd edition, IIBA, 2009

[13] Dan Brickley, Ramanathan Guha, *Resource Description Framework Specification*, 2000

[14] Miroslav Blaško, Bogdan Kostov, Petr Křemen, *Ontology-based Dataset Exploration – A Temporal Ontology Use-Case*, Intelligent Exploration of Semantic Data, 2016

[15] Dan Brickley, Ramanathan Guha, *RDF Schema 1.1*, https://www.w3.org/TR/rdf-schema/, 2014

[16] W3C OWL Working Group, *OWL 2 Web Ontology Language Document Overview (Second Edition)*, https://www.w3.org/TR/owl2-overview/, 2012

[17] Jakub Klímek, Martin Nečaský, Bogdan Kostov, Miroslav Blaško, Petr Křemen, *Efficient Exploration of Linked Data Cloud, in Proceedings of 4th International Conference on Data Management Technologies and Applications*, SciTePress, 2015

10 Appendix

10.1 Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
DAO	Data Access Object pattern
DTO	Data Transfer Object pattern
ETL	Extract Transform Load
ID	Identifier
IO	Input/Output
IIBA	International Institute of Business Analysis
IRI	Internationalized Resource Identifier
JOPA	Java Ontology Persistence API
JS	JavaScript
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
JVM	Java Virtual Machine
KBSS	Knowledge Based Software Systems
LGPL	Lesser GNU Public License
MIT	Massachusets Institute of Technology
OWL	Ontology Web Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
SPARQL	SPARQL Protocol and RDF Query Language
TBC	Topbraid Composer
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

10.2 CD Contents

```
/ ................................................ Root directory
├── thesis.pdf ........................... The thesis in PDF format
├── thesis ..................................... Thesis sources
│   ├── thesis.tex ....................... The thesis LaTeX source
│   └── images ..................................... Illustrations
└── app ..................................... Application files
    ├── s-pipes-editor ............ SPipes Editor source directory
    ├── semantic-pipes-web ............. SPipes Engine web service
    └── 16gacr-model ....................... 16gacr-model files
```