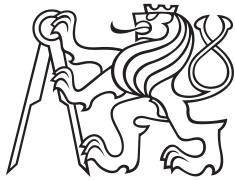


Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer science

Debugging scripts in SPipes editor

Bc. Petr Jordán

Field of study: Open informatics

Subfield: Software Engineering

Supervisor: Mgr. Miroslav Blaško, Ph.D.

August 2021

I. Personal and study details

Student's name: **Jordán Petr** Personal ID number: **423317**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Science**
Study program: **Open Informatics**
Specialisation: **Software Engineering**

II. Master's thesis details

Master's thesis title in English:

Debugging scripts in SPipes editor

Master's thesis title in Czech:

Ladění skriptů v SPipes editore

Guidelines:

SPipes (Semantic data pipelines) [1] is an RDF-based scripting language based on SPARQL motion [2]. It defines data pipelines in the form of an acyclic oriented graph of modules. Concrete modules are constructed in Java or defined declaratively within RDF. SPipes Editor [3] is a web-based editor that provides very basic support to manage SPipes scripts.

The goal of this work is to reimplement or extend the editor with main focus on advanced features to manage SPipes scripts. The new editor should be extended with validation and debugging of scripts. Validation will be used to check semantic constraints of the SPipes language and custom best-practice rules to write scripts. Debugging will allow defining test-cases to validate the pipeline, set up inputs and output of modules manually, reuse outputs of previous executions, or query execution history

Instructions:

- 1) become familiar with related Semantic Web technologies (OWL, RDF, JSON-LD, SPARQL, SHACL)
- 2) describe the current state of SPipes editor
- 3) review related data pipeline editors and libraries to support debugging, validation, and visualization of scripts
- 4) analyze requirements of the new editor
- 5) design and implement a prototype of new editor
- 6) test the implemented prototype, including user testing on at least 3 people
- 7) compare the prototype with its predecessor

Bibliography / sources:

- [1] Blaško, Miroslav and Petr Křemen, "SPipes" (online at <https://kbss.felk.cvut.cz/web/kbss/s-pipes>)
[2] TopQuadrant, Inc. "SPARQL motion" (online at <http://sparqlmotion.org>)
[3] 2018, Doroshenko Yan, Semantic pipeline editor (<https://dspace.cvut.cz/handle/10467/76534>)
[4] 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.
[5] 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.

Name and workplace of master's thesis supervisor:

Mgr. Miroslav Blaško, Ph.D., Knowledge-based Software Systems, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **21.02.2021** Deadline for master's thesis submission: **13.08.2021**

Assignment valid until: **19.02.2023**

Mgr. Miroslav Blaško, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Acknowledgements

I would first like to thank my supervisor for his guidance, domain expertise, and leadership while writing this thesis.

I would also like to acknowledge my friends Matěj and Hanka for their help, patience, and endless support during the thesis writing.

And lastly, to my girlfriend Kristýna and my friend Petr for mutual motivation and words of endurance till the end.

Declaration

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

In Prague, 12. August 2021

Abstract

The SPipes language is a technology that enables the processing of structured data in the form of the Semantic Web. This thesis attempts to improve the existing SPipes script editor. The thesis first introduces the principles of the Semantic Web and related technologies. Based on a thorough analysis of the existing editor and conducted survey, the application architecture was redesigned and functional and non-functional requirements for the editor were defined. Main contributions of this work are re-implementation of the backend part from Scala to Java, which eliminates the compatibility issues arising from the incompatibility between Scala and the Spring framework that is used. Special attention was paid to writing tests for most parts of the application, which simplifies the detection of potential bugs in the application. Major change in architecture was to split the originally monolithic application into several separate services with the use of Docker and docker-compose, leading to simpler configuration and easier deployment of the application. Last but not least, this thesis introduces new non-trivial features of the editor - the capability of validating and debugging of SPipes scripts and modules.

Keywords: SPipes, Semantic web, RDF, SPARQL, SHACL, Spring boot, JOPA

Supervisor: Mgr. Miroslav Blaško, Ph.D.

Abstrakt

Jazyk SPipes je technologie umožňující zpracování strukturovaných dat Sémantického webu. Tato diplomová práce se zabývá zlepšením stavu stávajícího editoru SPipes skriptů. V práci jsou nejprve představeny principy Sémantického webu a relevantní technologie. Na základě zevrubné analýzy již existujícího editoru a provedené rešerše byla navržena úprava architektury aplikace a definovány funkční a nefunkční požadavky na editor. Hlavní přínosy práce jsou převedení backendové části z jazyka Scala do Javy za účelem odstranění problémů vyplývajících z nekompatibility mezi jazykem Scala a Spring frameworkem, který je použit. Dále pak vytvoření testů, které zjednodušují odhalení potenciálních chyb v aplikaci, rozdělení původně monolytické aplikace na několik oddělených služeb využívajících Docker a docker-compose, čímž se výrazně sníží práce spojená se správnou konfigurací a spouštěním aplikace. V neposlední řadě přináší tato práce nové a netriviální funkce editoru - možnost validovat a ladit editované skripty a moduly.

Klíčová slova: SPipes, Sémantický web, RDF, SPARQL, SHACL, Spring boot, JOPA

Překlad názvu: Ladění skriptů v SPipes editoru

Contents

1 Introduction	1	4.1.1 Visualization tools	37
2 Background	3	4.1.2 Graph database	39
2.1 Ontology	3	4.1.3 Standalone solutions for data visualization	41
2.2 Semantic Web	5	4.1.4 Summary	41
2.2.1 Layers	5	4.2 Data pipeline editors	42
2.3 RDF	7	4.3 Visualization libraries	43
2.3.1 RDFS	9	4.3.1 Original evaluation criteria	44
2.3.2 OWL	10	4.3.2 Original analysis results	45
2.3.3 RDF Serialization Formats	12	4.3.3 Libraries analysis	45
2.3.4 SPARQL	14	4.3.4 Feature matrix	47
2.4 SPARQLMotion	15	4.3.5 Evaluation of the results	47
2.5 SPipes	17	4.4 Validation	48
3 The Original State of the SPipes Editor	21	4.4.1 SHACL	48
3.1 Functionality	21	4.4.2 SHACL execution engines	48
3.1.1 Script Editing	21	5 Requirement Analysis	51
3.1.2 Notifications	23	5.1 New SPipes Editor	51
3.1.3 Script Execution	23	5.2 Analysis of the SPipes editor requirements	52
3.2 Implementation of the SPipes Editor	23	5.2.1 Prioritization Technique	52
3.2.1 SPipes Editor Backend	24	5.2.2 Functional Requirements	53
3.2.2 SPipes Editor Frontend	25	5.2.3 Non-Functional Requirements	62
3.3 Tests	25	5.2.4 Use-cases	63
3.3.1 Backend Tests	25	6 Architecture Design and Technologies	65
3.3.2 Frontend Tests	27	6.1 Application structure	65
3.4 Design Related Issues and Bugs in the SPipes Editor	27	6.1.1 Server side	66
3.4.1 Design Issues	27	6.1.2 Client side	66
3.4.2 Bugs	29	6.2 Technology stack	66
3.5 Problems Related to the Scala Language	30	6.2.1 Server side	66
3.5.1 Java	30	6.2.2 Client side	68
3.5.2 Scala	30	6.3 Design of non-trivial requirements	69
3.5.3 Interoperability between Scala and Java	31	6.3.1 Execution	69
3.5.4 Scala with the Spring framework	32	6.3.2 Debugging	70
3.5.5 Issues in Implementation with Spring Framework and SPipes Editor	33	6.3.3 Script validation	71
3.6 Summary of the SPipes Editor Original State	34	6.3.4 Modules grouping and collapsing	71
4 Review of related technologies	37	6.3.5 Module transfer	72
4.1 Graph-based RDF visualization tools	37	6.3.6 Script to form	72
		6.3.7 The actual state of the system for multiple users	73
		7 Implementation	75
		7.1 Legacy SPipes editor update	75
		7.1.1 Technology stack update	75
		7.1.2 Testability	76

7.1.3 Development simplification ..	77
7.1.4 UI Layout	77
7.2 Dockerization of SPipes editor ..	78
7.2.1 Docker and Docker compose ..	79
7.2.2 Containers and Docker compose	79
7.2.3 Reverse proxy	80
7.3 Requirements implementation ..	80
7.3.1 Implementation of Functional Requirements	81
7.3.2 Implementation of Non-functional Requirements ...	88
8 Testing and Evaluation	91
8.1 Testing	91
8.1.1 Unit tests	91
8.1.2 Browser testing	92
8.1.3 User testing	92
8.1.4 User testing evaluation	97
8.2 Comparison with original SPipes Editor	100
8.2.1 Implementation Comparison	100
8.2.2 Features Comparison	101
9 Conclusion	103
9.1 Summary	103
A Code Attachments	105
B Installation guide	107
B.1 Installation via Docker	107
C Attachment	109
D Bibliography	111

Figures

2.1 Domain model diagram	3	8.2 Hello-world3.sms.ttl script execution detail	94
2.2 Layered architecture of the Semantic Web [96]	6		
2.3 An example of informal graph [97]	8		
2.4 An example of SPARQLMotion script visualization [89]	16		
2.5 Hello-world.sms.ttl visualization as a graph	18		
3.1 Original SPipes editor	22		
3.2 Script editing	22		
3.3 Notification message	23		
3.4 Script execution	23		
3.5 High level deployment diagram .	24		
4.1 Linked Data Visualization Tools Timeline [3]	38		
4.2 CytoScape visualization example [21]	39		
4.3 Collapsing example	45		
5.1 SPipes language terminology . . .	52		
5.2 Function execution	59		
5.3 Module debugging proposal	60		
5.4 Script validation	61		
5.5 Use-case diagram	64		
6.1 Application structure	65		
6.2 Spring Boot Flow Architecture [85]	67		
6.3 Skosify script grouping and hierarchy example. Source is SPipes documentation.	71		
6.4 Module transfer. Source is adjusted from SPipes documentation	72		
7.1 Life cycle of the integration test	77		
7.2 Hello-world3.sms.ttl script execution detail	78		
7.3 Deployment to the cloud with Docker [61]	79		
7.4 Deployment diagram	80		
7.5 Scripts page	81		
7.6 SForms modal widows	82		
7.7 Modul menu	83		
7.8 List of executions	87		
8.1 Test coverage	92		

Tables

4.1 Visualization libraries usage in Visualization tools, Graph database, and Standalone solutions	42
4.2 ✓- supported feature \$ - payed feature ? - not exactly support requirements ✗- not supported . . .	47
7.1 Best practise to write SPipes language and custom best-practice rules to write scripts. The * is out of scope SHACL validation	88
8.1 Browser support	92
8.2 Participants profile	93
8.3 Scenarios duration for every participant	97
8.4 Implementation comparison between original and new SPipes Editor	101
8.5 Original and new SPipes editor functions comparison	102



Chapter 1

Introduction

In recent decades, the amount of information available on the World Wide Web (WWW) has increased dramatically. However, it was clear from the beginning that to exploit the Web's potential, information needed to be stored in a way that computers could retrieve and interpret it in a similar way that humans do. The Semantic Web is an extension of the WWW and consists of structured or semi-structured data that are semantically annotated, thus meaning of the data is recorded and available.

An important concept of the Semantic Web is the Resource Description Framework (RDF) – a general framework for description, exchange and reuse of metadata so that information can be processed across the Web without the loss of its context. On the basis of RDF, several languages have been created, one of being the SPipes. SPipes is a scripting language which allows to visualize data processing through its graphical notation representing pipelines as scripts consisting of modules (processing nodes) and dependencies between them showing the dataflow direction.

One of the limitations of SPipes is the absence of reliable and user-friendly tool aiming for simpler development of SPipes scripts that would be also able to execute them and which would provide debugging and validation capabilities.

The main objective of this thesis is to improve the existing SPipes editor – a Scala/React web application that allows the user to visualize and edit the associated folder with SPipes RDF Turtle(.ttl) scripts. The SPipes editor identifies functions in the script and subsequently runs them via SPipes engine.

This work aims to address known problems of the previous version of the SPipes editor, as well as providing in-detail analysis in order to reveal other issues making further development complicated. The contribution of this thesis encompasses re-implementing of the editor in order to mitigate the problems arising from the interoperability issues between Scala and the Spring framework. Furthermore, the formerly neglected issue of tests or notifications is addressed and overall simplification of the application is achieved by design changes and its containerization. Nevertheless, the main contribution should be seen in the addition of new advanced features such as script validation and script debugging.

The structure of this thesis is as follows. First, we provide a background of SPipes editor and its related technologies in Chapter 2. We give an introduction to Semantic Web and languages for work with Semantic web. Then, SPARQLMotion and its dialect the SPipes language are introduced.

In Chapter 3, the original state of the SPipes editor is reviewed. We briefly explain the editor functionality and describe the architecture and implementation details. Next, the state of the editor is summarized, shaping the next direction for the development.

Chapter 4 reviews related technologies of the RDF data visualization tools, which helps to select a proper visualization library for the new editor.

Chapter 5 describes the functional and non-functional requirements of the new editor.

In Chapter 6 the architecture of the editor and technologies selected for implementation are proposed. The solution and design of the more complex requirements left for future implementation are described.

Chapter 7 describes the implementation of the new editor based on the previously defined requirements.

Chapter 8 summarizes application testing and evaluates user testing based on created test scenarios. Finally, the comparison of the new editor with the old one is provided highlighting the benefits of the new one.

Chapter 2

Background

This chapter provides a brief overview of the technologies related to the main topic of this thesis – the SPipes editor. It introduces the term ontology, the very concept of the Semantic Web and associated technologies and languages such as Resource Description Framework (RDF), RDF Schema, and the Web Ontology Language (OWL). Further, selected serialization formats of RDF are presented herein. As the SPipes are heavily based on the SPARQLMotion scripting language, a separate chapter is devoted to its description. Finally, the SPipes scripting language itself is introduced.

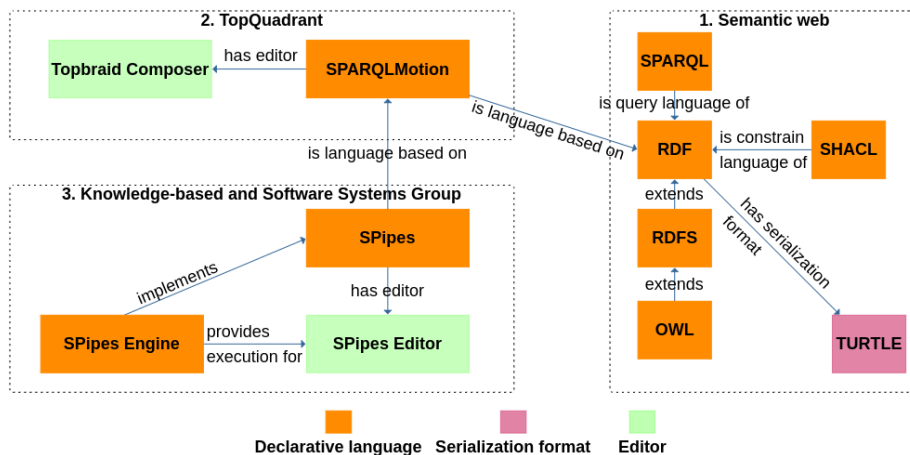


Figure 2.1: Domain model diagram

2.1 Ontology

An ontology, in its original meaning, is a philosophical discipline that deals with being or existence. For the field of computer science, the term was borrowed Gruber et. al. [32] as an “explicit specification of a conceptualization”. In other words, ontologies define relationships between concepts from a certain domain of knowledge or discourse. Ontologies can be represented by semantic networks, ontological dictionaries, and models, as well as thesauri

and classification schemes.

Each ontology formally defines terms and relationships. Most commonly, ontology is represented by a taxonomy, a scheme of classification in which things are classified into groups or types, and a set of derivation rules. The ontology data model generally consists of four basic types of elements [63]:

- **An individual** (object, instance) is the basic building block of an ontology data model. An entity can be specific (human, table, molecule) or abstract (number, concept, event)
- **A class** is a set of entities of a certain type. A subset of a class is a subclass. A class can contain both entities and subclasses. A taxonomy is usually defined on a set of classes (using multiple inheritance) [86].
- **An relation** describes a property, characteristic, or parameter of an entity. Each relation of a particular entity contains at least a name and a value. The relation is used to store certain information related to a given entity.
- **An axiom** is used to place constraints on the meanings of definitions or instances in order to ensure that ontology is consistent.

In summary, as described in [86], an ontology is an set of representational primitives consisting of:

- A set of terms or vocabulary
- A set of relationships between the terms
- A set of logical axioms that defined the desired vocabulary

The possible applications of ontologies are wide and apply in many fields. They expand the functionality of the World Wide Web – for example, they increase the accuracy of search engines. In e-commerce, they ensure a better understanding between the seller and the customer, and terminological ontologies can help in text translation or summarization.

Ontologies can be represented by formal, semi-formal, or informal languages – on the web, they are defined by artificial formal (ontological) languages. The first “web” ontological languages were represented by SHOE (simple HTML Ontology Extension) and Ontobroker developed in the mid 1990s, which enabled incorporation into the source code of web pages both metadata about the objects to which these pages relate, as well as the ontology defining the semantics of this metadata itself [86]. In the late 1990s, the RDF specification of metadata standard was published by the W3C consortium. From this moment ontological languages which provide RDF-compatible metadata were used – RDF in XML syntax, RDF Schema, OWL, and Turtle (Terse RDF Triple Language). Also, an alternative to ontological languages was developed – the Topic Maps technology, which enables the creation and management of the so-called thematic maps.

2.2 Semantic Web

The semantic web can be described as a way to implement the ontological approach to networks of computers, i.e., to represent the network content in a machine-processable form. The idea of ontological approach appears in seminal 1994 when Berners-Lee et al. [12] in a proposal for World Wide Web (WWW) future development stated the need for “Evolution of objects from being principally human-readable documents to contain more machine-oriented semantic information”. Further interpretation of the semantic web varied; however, the main concept of the global Web of machine-readable data remained unified [13]. Nowadays, the term semantic web is generally used for describing both the technologies which handle the data and the repository of datasets introduced by these technologies – the Semantic Web [22].

The original WWW primarily consists of unstructured data which are linked together using plain references (hyperlinks). On the contrary, the Semantic Web, being the extension of WWW, consists of structured or semi-structured data that are semantically annotated. It implies that information about the relations and interconnections between them is recorded on this network [20]. The Semantic Web presents a view of documents not only as text and graphic files, but also carries the information about their meaning. As all of the information is clearly-defined and computer interpretable, it enables easier manipulation and interpretation of real-world domain knowledge [81].

The Semantic Web is created and structured according to certain rules and standards such that the required information could be effectively obtained. The Semantic Web expects implementation of standards for semantic (RDF), structural (XML), and syntactic (URI) components in the web documents architecture. The Semantic Web framework has a layered architecture (also known as Semantic Web Stack, Semantic Web Cake or Semantic Web Layer Cake [22]) where various technologies are layered in a way that each layer is based on the lower layer (example in Figure 2.2). Each layer (i.e., technology) in the Semantic Web framework has its own purpose (data networking, semantic annotation, knowledge representation, query of semantic data, reasoning and inferencing, software user agents, etc. [20]) enabling creation of Semantic Web application.

2.2.1 Layers

Let us now briefly describe the layers of the Semantic Web on a Figure 2.2, which is based on a survey presented in [81]. The bottom layer consists of **IRI** (Internationalized Resource Identifier) to identify individual objects from the international Unicode character set, which is the base for both the standard Web and Semantic Web. IRI is a Unicode string that conforms to the syntax defined in RFC 3987, and it is a generalization of URI (Universal Resource Identifier) [40].

The following layer includes **XML markup language, XML Schema and Namespaces**. The XML technologies enable the creation of individual

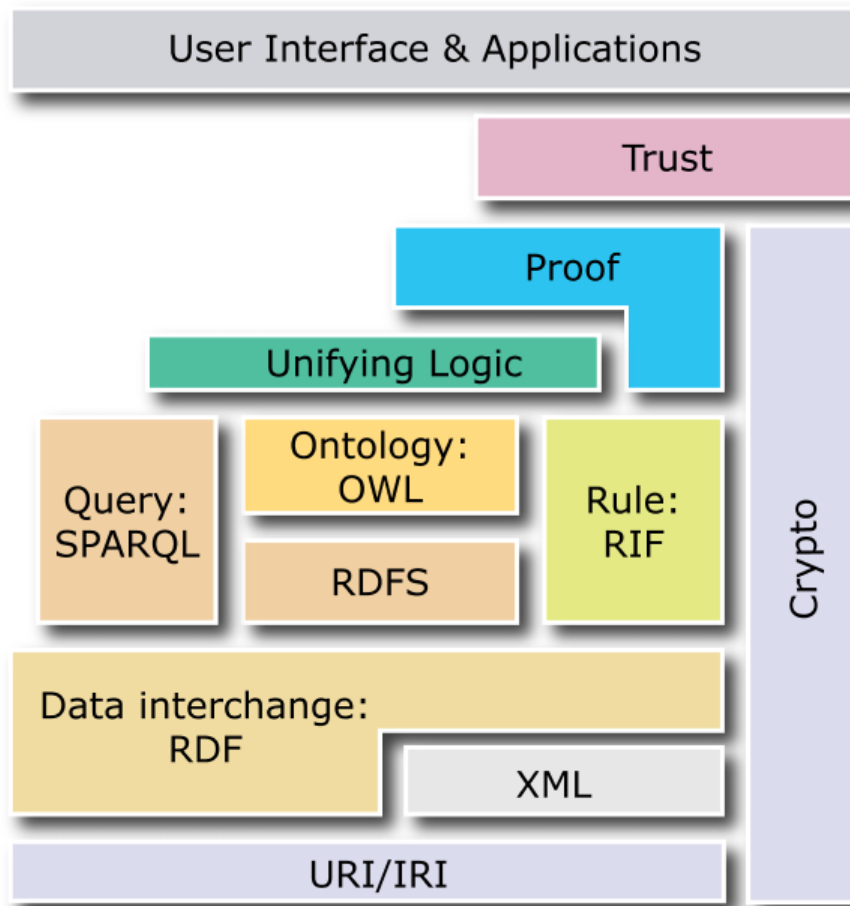


Figure 2.2: Layered architecture of the Semantic Web [96]

descriptions in the Semantic Web in XML-based standards. This brings a benefit of having the created documents in form of structured data. The Namespaces in XML enable the use of tags, thus facilitating readability of the documents.

As previously mentioned, the RDF standard is a kind of extension of XML. In the following layer, the combination of XML and RDF technologies defines one of the basic languages for storing information on the Semantic Web – XML/RDF. The XML/RDF language provides the ability to assign certain properties to a specific web source, or to describe relationships between selected web resources. In the Semantic Web, a resource is any object which has an URI/IRI identifier assigned. When using only XML without the RDF standard, the need for URI/IRI identifier is not required. Thus, the requirement is not generally met in the standard Web as it is written using HTML which originates in XML.

Next layer consists of the already mentioned **RDFS language** which enables creation of simple RDF schemes – classes, properties and taxonomies can be defined here.

All the remaining technologies are based on the **RDF format**. RIF (Rule Interchange Format) is a format for creation and modification of rules on the Semantic Web. **OWL** is an ontological language that defines dictionaries for interpretation of information semantics and derivation of further information using applicable logic. It is a more expressive alternative to RDFS and it allows creation of more complex domain models. **SPARQL** is a query language which evaluates queries over RDF.

Encryption and digital signatures are applied in order to ensure credibility and authenticity of the documents (referenced as Crypto in Figure 2.2). Upper layers are still not standardized by W3C consortium, thus not completely resolved. Unifying logic serves for automatic derivation of information from ontologies and semantic data. The Proof layer is meant to determine credibility of the obtained information. In a similar way, the Trust layer is based on the verification that the information comes from a trusted source. The User Interface and Applications is the final layer in the Semantic Web and it allows users to employ the described technologies and principles.

Several of the layers in the architecture are already standardized by W3C, for example the RDF (Resource Definition Format), OWL (Web Ontology Language), or SPARQL (SPARQL Protocol and RDF Query Language). Others, especially the upper layers, are still waiting to be standardized.

Selected technologies mentioned in this chapter will be further described in more detail.

2.3 RDF

The Resource Description Framework (RDF) is, as already mentioned in 2.1, a recommendation of the W3C consortium for representing the structure of web metadata [65]. It is a general framework for description, exchange and reuse of the metadata so that information can be processed across the Web without the loss of meaning.

The abstract RDF syntax (i.e., a data model that is independent of a particular concrete syntax) has two key data structures: RDF graphs and RDF datasets. RDF graphs are sets of triples (statements) consisting of subject \rightarrow predicate \rightarrow object. In each statement, the subject is the resource, predicate is the property (i.e., what we claim about the subject), and object is the property value [86]. The basic syntax of RDF recommended by W3C is based on XML. By the so-called serialization, the individual elements of the RDF statement are arranged in a specific way into XML elements and attributes.

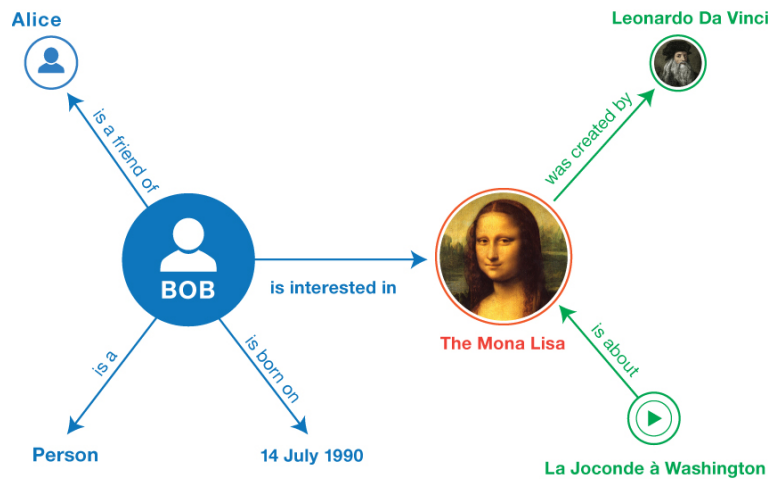
RDF datasets organize collections of RDF graphs, and comprise a default graph and zero or more named graphs [65]. An example of an informal graph of triples can be found in [66]. The triples defining the graph in Listing 1 are following:

An important ability of RDF graphs is evident from the node “The Mona Lisa”. In two triples, the Mona Lisa represents the subject, but in one triplet, it is the object. This ability to have the same resource as subject and object

```

1 <Bob> <is a> <person>.
2 <Bob> <is a friend of> <Alice>.
3 <Bob> <is born on> <the 4th of July 1990>.
4 <Bob> <is interested in> <the Mona Lisa>.
5 <the Mona Lisa> <was created by> <Leonardo da Vinci>.
6 <the video 'La Joconde à Washington'> <is about> <the Mona Lisa>

```

Listing 1: Example of triples [66]**Figure 2.3:** An example of informal graph [97]

makes it possible to find connections between the triples [66]. Thus, based on these relationships, a query language SPARQL can be subsequently used over the graph to obtain information, e.g., people interested in Leonardo Da Vinci.

Now, let us discuss the possible data types that can occur in triples: IRIs, literals, and blank nodes [67]. International Resource Identifier (IRI), a generalization of URI as mentioned above, identifies a *resource*. RDF conceptualizes anything (and everything) in the universe as a resource. A resource is simply anything that can be identified with the IRI. The URLs (Universal Resource Locators), which are used to identify where digital information can be retrieved, are one form of IRI. While URLs tell you where to find specific information, they also provide a unique identifier for the information. IRIs generalize this concept further by saying that anything, whether you can retrieve it electronically or not, can be uniquely identified in a similar way. Thus, other forms of IRI provide an identifier for a resource without implying its location or how to access it [66]. IRIs can appear in all three positions in a *triple*.

Literals are, simply said, values that are not IRIs. Thus literals are primitive literal values such as strings, integers, decimals, booleans, etc. For example, in the graph in Figure 2.3, "the 4th of July, 1990" is a literal. In RDF triples,

literals can appear only in the object position.

Blank nodes are used when we want to talk about resources without using IRI. Blank nodes then serve as variables in algebra; we can use them for representing things without assigning them specific values. Blank nodes can be subjects or *objects*.

■ 2.3.1 RDFS

RDF Schema (Resource Description Framework Schema, or abbreviated as RDFS) is an ontological language developed by W3C in late 1990s. In this chapter, description of RDFS will be provided based on the official W3C Recommendation [68].

RDFS represents a semantic extension which adds mechanism to RDF for distinguishing types, basic manipulation with classes and properties. Thus, it provides mechanisms for describing groups of related resources and the relationships between these resources. To put it simply, RDFS allows designers to create their own ontologies and RDF vocabularies.

The class and property system of RDFS have many similarities with traditional object-oriented languages such as Java. The main difference of RDF Schema is that instead of defining a class in terms of the properties its instances may have, RDF Schema describes properties in terms of the classes of resource to which they apply. For this reason, domain and range mechanisms are applied. An illustrative example is provided in [68]: the `eg:1author` property has a domain of `eg:Document` and a range of `eg:Person`, whereas a classical object oriented system might typically define a class `eg:Book` with an attribute called `eg:author` of type `eg:Person`. The main advantage of this property-centric approach is that it is easy to subsequently define additional properties with a given domain or range.

RDFS, unlike richer vocabulary or ontology languages such as OWL, does not attempt to enumerate all the possible forms of representing the meaning of RDF classes and properties. The RDFS language consists of a collection of RDF resources that can be used to describe other RDF resources in application-specific RDF vocabularies. The core vocabulary is defined in a namespace identified by IRI <http://www.w3.org/2000/01/rdf-schema>, conventionally associated with the prefix `rdfs:`.

Resources can be grouped into units, in RDFS called *classes*. The resources are then instances of the given class. Several key classes in RDFS are:

- **rdfs:Resource** All things described by RDF are called *resources*, and are instances of the class `rdfs:Resource`. `rdfs:Resource` is an instance of `rdfs:Class`.
- **rdfs:Literal** The class `rdfs:Literal` is the class of literal values such as strings and integers. Property values such as textual strings are examples

¹The *eg* is a *prefix*, which allows you to write prefix names instead of having to use full URIs everywhere.

of RDF literals. `rdfs:Literal` is an instance of `rdfs:Class`. `rdfs:Literal` is a subclass of `rdfs:Resource`.

- **rdf:Property** `rdf:Property` is the class of RDF properties. `rdf:Property` is an instance of `rdfs:Class`.

All properties used in the RDF document are automatically members of the `rdf:Property` class. Some of the basic properties defined in RDFS are:

- **rdfs:range** is used to state that the values of a property are instances of one or more classes.
- **rdfs:domain** is used to state that any resource that has a given property is an instance of one or more classes
- **rdf:type** is used to state that a resource is an instance of a class.

RDF Schema is a basic ontological language and comes with several limitations. To name a few [68], the following constraints cannot be expressed in RDFS:

- *cardinality* constraints on properties, e.g., that a Person has exactly one biological mother.
- defining property to be *transitive*.
- defining that a given property is a unique identifier for instances of a particular class.
- defining that two different classes (different IRIs) represent the same class.
- defining that two different instances (different IRIs) actually represent the same individual.
- defining constraints on the range or cardinality of a property that depend on the class of resource to which a property is applied, e.g., being able to say that for a soccer team the `ex:hasPlayers` property has 11 values, while for a basketball team the same property should have only 5 values.

In order to describe data in such a way, an ontological language with greater expressive ability is needed. One of such languages is introduced in the next section.

■ 2.3.2 OWL

OWL (Web Ontology Language) is a language standardized by the W3C consortium for publishing and sharing ontologies. OWL is based on RDF and RDF Schema and serves as an extension of RDFS vocabulary for describing properties and classes. By providing additional vocavular terms and formal semantics, it is a much more effective tool for interpreting information on the Web than the XML, RDF and RDFS technologies [72].

The main uses of OWL can be summarized in the following three points [62]:

1. Creating a domain by defining classes and their properties.
2. Defining instances (elements of classes) and their properties.
3. These classes and instances to an acceptable degree using the formal semantics of the OWL language.

When comparing RDFS and OWL as its extension, several points can be observed [64]:

- OWL uses all structures from RDFS (domain, range, etc.)
- The most significant extension is the ability to use not only named classes, but also anonymous classes defined by a logical expression.
- Range of properties are explicitly distinguished as objects (the value is an instance of a class) and data (the value is a literal).
- OWL gives the possibility to formulate a negative statement.

Herein, selected terms defined by OWL: *owl: imports* and *owl: Ontology*, will be briefly described, based on [62], as they are key elements in the SPipes scripts modularization. Ontologies can be grouped under the *owl: Ontology* tag. The *owl: Ontology* element is a place to collect much of the OWL metadata for the document. The *owl: imports* statement provides an include-style mechanism. *Owl: imports* inserts ontology into a document, specified by an XML attribute *rdf: resource*, possibly accompanied by a declaration of namespace.

An example can be seen in 2:

```

1 <owl:Ontology rdf:about="">
2   <rdfs:comment>An example OWL ontology</rdfs:comment>
3   <owl:priorVersion rdf:resource="http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine"/>
4   <owl:imports rdf:resource="http://www.w3.org/TR/2004/REC-owl-guide-20040210/food"/>
5   <rdfs:label>Wine Ontology</rdfs:label>
6   ...

```

Listing 2: *OWL:Ontology* example [62]

There are several different syntaxes available for persisting, sharing, and editing OWL ontologies [36]:

- the Functional OWL syntax,
- RDF-based syntaxes (RDF/XML, Turtle),
- OWL/XML,
- the Manchester OWL syntax

It is important to point out that the OWL language is not defined using a particular concrete syntax; however, RDF/XML is specified by W3C as the default exchange syntax, while some of the other syntaxes are mentioned in the W3C notes.

There are several species of OWL language, collectively referred to as OWL *Family*. OWL is used to denote the 2014 specification, OWL2 the 2009 specification. OWL specification includes three sublanguages with different levels of expressiveness: OWL Lite, OWL DL and OWL Full. OWL 2 is defined by three sublanguages: OWL 2 EL, OWL 2 QL, and OWL 2 RL. In this thesis, we will not describe individual sublanguages in greater detail; however, when creating ontologies using OWL, requirements and the specific sublingual relationship with RDF need to be taken into account in order to choose the most suitable option.

■ 2.3.3 RDF Serialization Formats

As mentioned in previous chapters, RDF, unlike some other data models, is not bound by a single serialization format. The *triples* statements (i.e., *subject, predicate, object*) can be represented using various languages, e.g., Turtle, N-Triples, JSON-LD, and RDF/XML. Given that each language has its own unique syntax, it depends on the specific purpose for which we serialize the linked data. In this chapter, selected serialization formats important for this project will be briefly introduced.

■ Turtle

Turtle (Terse RDF Triple Language) is a syntax and file format for describing the information in RDF data model. Standardized version of Turtle was published in 2014 by W3C. A Turtle document provides a way to represent RDF graphs as text [93]. Compared to other formats, Turtle is easily readable by humans, making comprehension and processing simple. One of the features that facilitate the more user-friendly form is the possibility of truncating triples using prefixes at the beginning of a turtle file. Furthermore, it allows grouping of triples with an identical subject into blocks, making the code more structured(object oriented) and clearer [1]. Apart from the readability of a Turtle file, thanks to the lack of closing lines at the beginning and end of the file, the data can be streamed in blocks (unlike in RDF/XML) [93]. An example of a Turtle file is provided in Listing 3:

■ JSON-LD

JSON-LD (JavaScript Object Notation for Linked Data) is a data serialization format based on JSON language, standardized by the W3C consortium [45]. The syntax was designed to facilitate a smooth upgrade path in JSON systems – it allows existing JSON to be interpreted as linked data with minimal changes. JSON-LD is primarily intended to build interoperable Web services, as well

```

1  BASE    <http://example.org/>
2  PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
3  PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
4  PREFIX  schema: <http://schema.org/>
5  PREFIX  dcterms: <http://purl.org/dc/terms/>
6  PREFIX  wd: <http://www.wikidata.org/entity/>
7
8  <bob#me>
9      a foaf:Person ;
10     foaf:knows <alice#me> ;
11     schema:birthDate "1990-07-04"^^xsd:date ;
12     foaf:topic_interest wd:Q12418 .
13
14 wd:Q12418
15     dcterms:title "Mona Lisa" ;
16     dcterms:creator <http://dbpedia.org/resource/Leonardo_da_Vinci> .
17
18 <http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619>
19     dcterms:subject wd:Q12418 .

```

Listing 3: Example of Turtle [66]

as store the linked data in storage engines based on JSON [45]. JSON-LD provides all features available in JSON and further introduces [45]:

- identification of JSON objects with IRIs
- clearly distinguishing shared keys in different JSON documents using IRIs and *context* (a set of rules for JSON-LD document interpretation)
- the way a resource on different site on the Web can be assigned to JSON object value
- annotation of string with language
- coercion of values to specific data types
- the way of writing down more than one directed graph in a single document

The concept of *context* is the key functionality of JSON-LD as it facilitates mapping from JSON to RDF. It links the concepts in ontologies to object properties in a JSON document.

JSON-LD combines three ways of data processing: accessing of raw triples, use of graph processing API, and converting linked data to tree structures [46]. An example of a JSON-LD document can be found in Listing 4:

```
1 {
2   "@context": "example-context.json",
3   "@id": "http://example.org/bob#me",
4   "@type": "Person",
5   "birthdate": "1990-07-04",
6   "knows": "http://example.org/alice#me",
7   "interest": {
8     "@id": "http://www.wikidata.org/entity/Q12418",
9     "title": "Mona Lisa",
10    "subject_of": "http://data.europeana.eu/item/04802/243F",
11    "creator": "http://dbpedia.org/resource/Leonardo_da_Vinci"
12  }
13 }
```

Listing 4: Example of JSON-LD [66]

2.3.4 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is an ontological query language for RDF and it is standardized by W3C [82]. Queries are, like in RDF, expressed using *subject-predicate-object* triples, called here basic graph patterns. In SPARQL, a variable can be used in any of the three positions. SPARQL can return none, one, or more answers. The query is typically performed on one RDF dataset – a set of RDF documents belonging to a specific endpoint. Such an endpoint is the URL address to which queries can be sent and results received via the HTTP protocol.

Compared to query languages for relational databases, there are similarities in keywords (e.g., SELECT, WHERE, and FROM). SELECT query consists of two main components: a list of selected variables and a WHERE clause which specifies the graph pattern to match (although WHERE clause can be omitted). The SELECT query returns a table with the selected variables as columns and rows representing the matched graph patterns. By the query, we are searching for matching *triples* using the *triple* patterns and the variables serve as wild cards that match any node. An example is given below in Listing 5 where we make a query using a single triple pattern `?picture rdf:type :Picture`. In this case, `?picture` is the variable, `rdf:type` is the *predicate*, and `:Picture` is the *object*.

```
1 SELECT ?picture
2 WHERE {
3   ?picture rdf:type :Picture .
4 }
```

Listing 5: SPARQL SELECT example

SPARQL provides many other expressions with similar functionality as query languages for relational databases, such as `FILTER`, `ORDER BY`, `DISTINCT`, `GROUP BY`, `UNION`, `DESCRIBE`, etc. Some additional expressions, however, provide advantageous mechanisms for dealing with the RDF data model. For example in some situations, we would like to construct a new graph from the solution set but the standard `SELECT` expression returns a table. In SPARQL, it is facilitated by the `CONSTRUCT` clause which replaces the `SELECT` clause. The other clauses work in exactly the same way as in the `SELECT` form. An example of the `CONSTRUCT` clause is provided in Listing 6:

```

1 CONSTRUCT {
2   ?picture a :Picture
3 }
4 WHERE {
5   ?pCollection a :PictureCollection ;
6   :picture ?picture
7 }

```

Listing 6: SPARQL `CONSTRUCT` and `WHERE` example

The `CONSTRUCT` expression returns read-only access to data. If we want to alter the data (e.g., add or delete triples into/from the RDF database), the `INSERT` and `DELETE` clauses can be used, such as in this example provided in Listing 7:

```

1 DELETE {
2   ?picture :style ?s1
3 }
4 INSERT {
5   ?picture :style ?s2
6 }
7 WHERE {
8   ?picture a :Picture ;
9   :price ?s1
10  BIND("landscape" AS ?s2)
11 }

```

Listing 7: SPARQL alter data example

2.4 SPARQLMotion

SPARQLMotion [83] is a scripting language based on RDF which allows to visualize data processing pipelines through its graphical notation. It

interconnects individual processing steps (queries and data transformations) in such a way that the output of the previous step represents the input for the next one. Generally, RDF graphs are passed between the steps; however, named variables pointing to RDF nodes and XML documents can be also used.

In SPARQLMotion, processing steps can be chained together to create complex processing pipelines which are able to assemble different sources of data. Thus, new applications such as reports, information dashboards, or data exchange between the backend systems can be created [92].

SPARQLMotion scripts are composed of modules, where each module represents one processing step. A relationship between the modules (i.e., processing steps) can be established. It is possible to visualize these connections shown in Figure 2.4; however, the scripts are ultimately stored as RDF models [83].

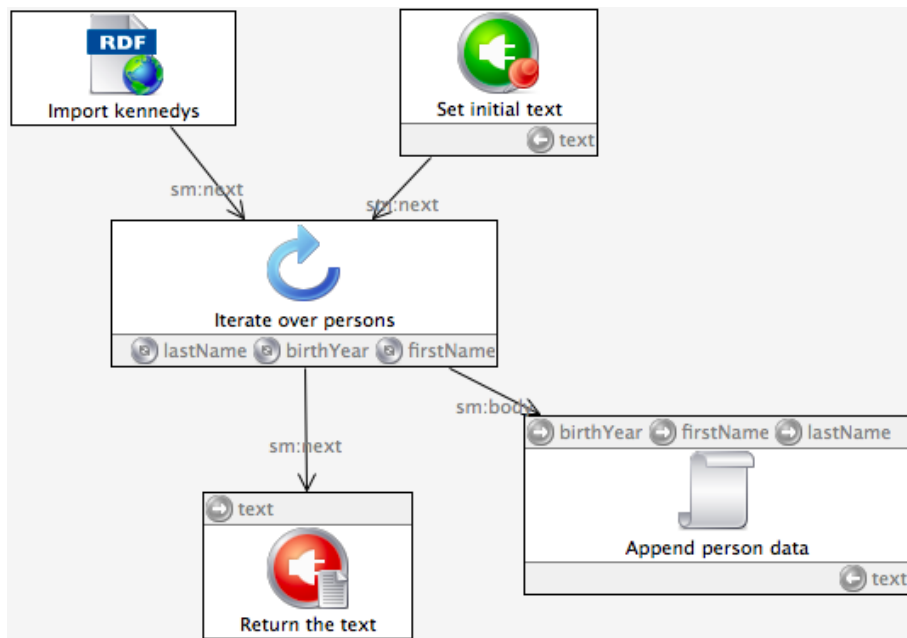


Figure 2.4: An example of SPARQLMotion script visualization [89]

In order to facilitate better understanding of SPARQLMotion, its key concepts are herein described:

- **Script** - A script is typically an RDF file in a format composed of modules that are connected by relations. Every well-formatted SPARQLMotion script should have at least one module without successors. These modules are called target modules that can be executed. The SPARQLMotion script can be saved in any format, and it is composed of triples that represent the ontology.
- **Variable** - Modules can also communicate using variables, where the variable is represented as a name-value pair. The variables can then be

used in SPARQL queries or SPARQL expressions. An example of the variable *text* can be seen in Figure 2.4, in the module *Set initial text*. This module only initializes *text* variable, which is later on passed into another module *Iterate over persons*.

- **Modules** - The individual modules can be seen in Figure 2.4. Their behavior depends on a `ModuleType` and the specific implementation of the module. The modules can be further interconnected by means of links. The output of a specific module is used in another module, using the *sm: next* binding.

2.5 SPipes

SPipes is an RDF language based on SPARQLMotion language that is restricted to semantic data flows as an acyclic graph of modules. It's the main difference from SPARQLMotion. The modules could be defined in Java or declaratively in RDF. The only supported serialization format of the execution is Turtle. The implementation of SPipes language is SPipes engine, which allows serializing data of the execution into an ontology. The data could be saved into the files or RDF4j server repository. The example script is shown in Listing 10 with the explanation is described in the following section:

1. The script contains only one pipeline and one input variable *varName* visible on line 16. The pipeline is visualized as a graph on Figure 2.5.
2. The script IRI of the pipeline is defined on line 8 as *a owl:Ontology*. Also, the *s-pipes-lib* is imported, which allow us to use modules such as *sml:BindWithConstant* or *sml:ApplyConstruct*. It is important to note that both modules are from SPARQLMotion, but as mentioned at the beginning of the chapter, SPipes is built on top of SPARQLMotion, but it can not use all its modules. For a complete list of SPARQLMotion modules, see [88], or SPipes modules[73].
3. Lets move to execution of the pipeline. Every script has to have a least one *sm:Function*, which is used as trigger. In our case the *:execute-welcome* on line 37.
4. Lets call *:execute-welcome* and set the input variable *varName=Petr* via REST-API (the documentation of the endpoints is in SPipes documentation²). You need to realize how the script is run and how it actually works. *:execute-welcome* returns *:welcome_Return*, which nevertheless depends on *:construct-welcome*, which depends on *:bind-person-name*. So the script is executed from the end(*:welcome_Return*) - for a better visualization, look at Figure 2.5. It is important to realize that a module can have more than one input, so we have to build the execution from the end.

²<https://github.com/kbss-cvut/s-pipes>

5. Thus `:execute-welcome` really starts on the module `:bind-person-name`, where the output variable `sm: outputVariable "personName"` with the value `"Petr"` is set. This module calls the `:construct-welcome` module, which uses the SPARQL CONSTRUCT query to build the resulting *RDF triplet*.
6. The result of the pipeline in RDF format is in Listing 8; however the SPipes returns result as JSON-LD shown in Listing 9.

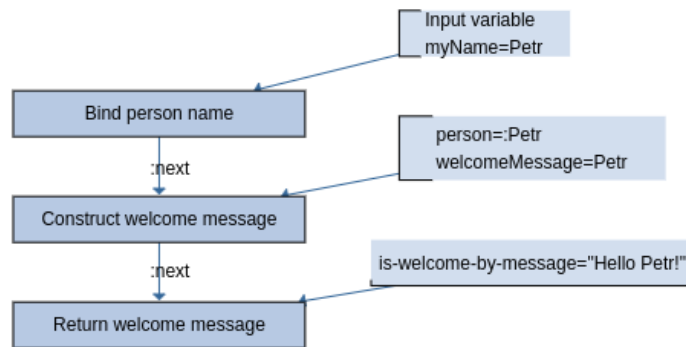


Figure 2.5: Hello-world.sms.ttl visualization as a graph

```

1 <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world/Petr>
2 <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world/is-welcome-by-message>
3 "Hello Petr!" .

```

Listing 8: SPipes hello-world.sms.ttl as RDF

```

1 {
2   "@id": "http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world/Petr",
3   "is-welcome-by-message": "Hello Petr!",
4   "@context": {
5     "is-welcome-by-message": {
6       "@id": "http://onto.fel.cvut.cz/ontologies/s-pipes/hello-worl
7         d/is-welcome-by-message"
8     },
9     "@vocab": "http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world/"
10  }
11 }

```

Listing 9: SPipes hello-world.sms.ttl as JSON-LD

```

1  @prefix : <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world> .
2  @prefix owl: <http://www.w3.org/2002/07/owl#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4  @prefix sm: <http://topbraid.org/sparqlmotion#> .
5  @prefix sml: <http://topbraid.org/sparqlmotionlib#> .
6  @prefix sp: <http://spinrdf.org/sp#> .
7
8  <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world>
9    a owl:Ontology ;
10   owl:imports <http://onto.fel.cvut.cz/ontologies/s-pipes-lib> ;
11 .
12 :bind-person-name
13   a sml:BindWithConstant ;
14   sm:next :construct-welcome ;
15   sm:outputVariable "personName" ;
16   sml:value [sp:varName "myName" ; ] ;
17   rdfs:label "Bind person name" ;
18 .
19 :construct-welcome
20   a sml:ApplyConstruct ;
21   sm:next :welcome_Return ;
22   sml:constructQuery [
23     a sp:Construct ;
24     sp:text ""
25 PREFIX : <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world>
26 CONSTRUCT {
27   ?person :is-welcome-by-message ?welcomeMessage .
28 } WHERE {
29   BIND(iri(concat(str(:), ?personName)) as ?person)
30   BIND(concat("Hello ", ?personName, "!") as ?welcomeMessage)
31 }
32 "" ;
33   ] ;
34   sml:replace true ;
35   rdfs:label "Construct welcome message" ;
36 .
37 :execute-welcome
38   a sm:Function ;
39   sm:returnModule :welcome_Return ;
40   rdfs:subClassOf sm:Functions ;
41 .
42 :welcome_Return
43   a sml:ReturnRDF ;
44   sml:serialization sml:JSONLD ;
45   rdfs:label "Return welcome message" ;
46 .

```

Listing 10: SPipes hello-world.sms.ttl example

Chapter 3

The Original State of the SPipes Editor

In this section, we summarize the original state of the SPipes editor – the subject of this diploma thesis. The main functionality and supposed applications of the SPipes editor are presented, together with the official assignment of the previous work. Next, we discuss the design of the application as well as the design of its main components and their relationships. The next part is devoted to a discussion about the suitability of used technologies and made design choices and its possible shortcomings. We close this section by listing the bugs and other issues found in this project.

The following section reviews key parts of SPipes editor and summarizes shortcomings of its implementation.

3.1 Functionality

The SPipes editor is a Scala/React web application that allows the user to visualize and edit the associated folder with SPipes RDF Turtle(.ttl) scripts. The SPipes editor identifies functions in the script and subsequently runs them via SPipes engine.

3.1.1 Script Editing

The user interface (UI) of the application is provided by a web page where users can browse stored scripts and script functions. The script view contains script layout options and visualization of the acyclic graph of the corresponding script. The layout options (Box, Layered, Radial, etc.) define the rendering strategy of the graph. The most useful visualisation is often given by the layered graph, where vertices of the directed graph are represented by horizontal (vertical) rows or layers, and the edges between them are generally directed downwards (or from left to right). [52]

The UI Figure 3.1 also offers the possibility of editing the graph. It is possible to edit every individual module (node) and its edges. It is important to mention that all of these operations could be done directly inside the .ttl script; however, it is much more convenient to use a graphical interface. Also, the possibility of any kind of syntactic error is reduced.

3. The Original State of the SPipes Editor

The editing consists of two separate operations. The first operation is related to the module's relationship. Users can add edges between nodes with drag and drop or delete edges in a popup context menu. The resolution of this operation is deletion or creation of the `sm:next` property between two modules.

The second operation is editing the module (node) parameters via the module context menu 3.2a and SForms ¹ 3.2b. Based on the module type, the SForms framework is used to provide a form for editing of this module. It should be noted that the module type cannot be changed. The workaround is to create a new module with a new type and delete the old one. Later on, the edges can be added.

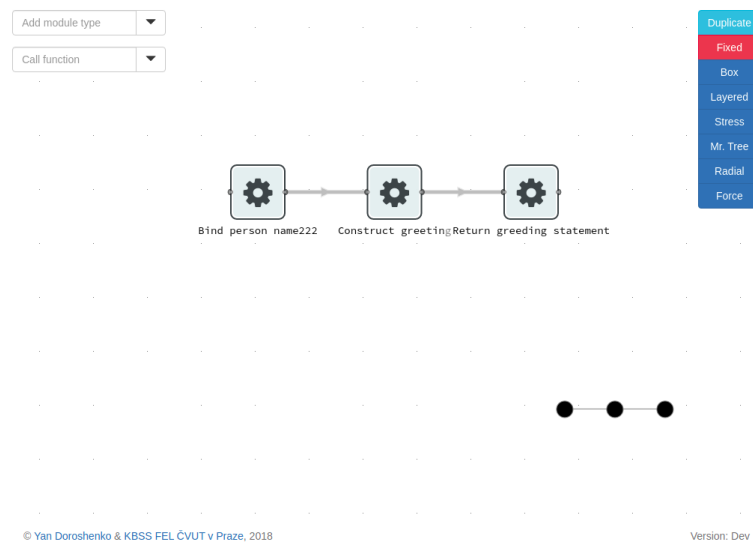
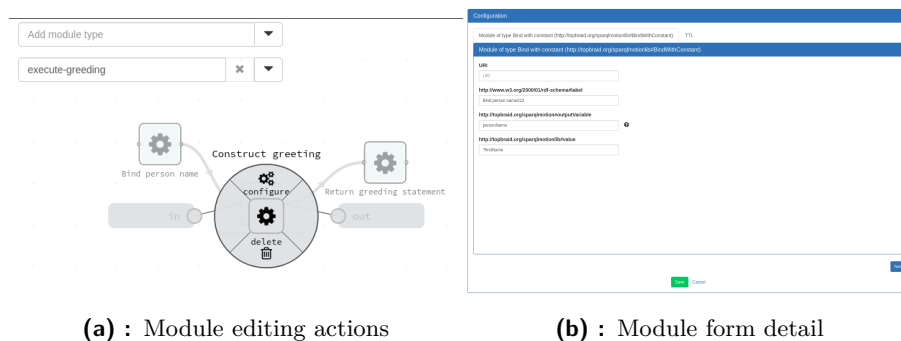


Figure 3.1: Original SPipes editor



(a) : Module editing actions

(b) : Module form detail

Figure 3.2: Script editing

¹<https://github.com/kbss-cvut/s-forms>

3.1.2 Notifications

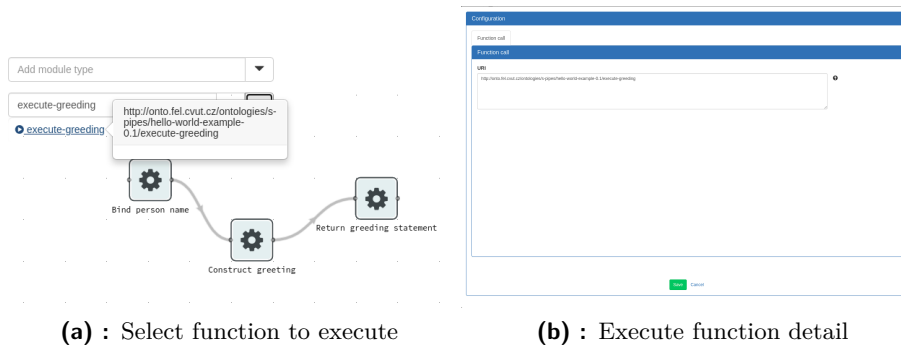
SPipes editor notifies the users by a popup window when a script is edited or deleted. Detail of the notification message is in Figure 3.3.. This allows multiple users to edit one script at the same time. The notification appears even in the case the change is made directly in a script file. The notification only informs about the change event without providing detail and offers users to reload the script page.



Figure 3.3: Notification message

3.1.3 Script Execution

Scripts that contain a function can be browsed on the function page. The function is defined as an instance of type *sm:Function* described in section 2.5. All the functions that are defined in scripts containing suffix *sms.ttl* are listed in the drop-down menu and they can be called by setting up parameters through SForm form. The execution is possible in detailed view of the *.ttl* script and is done by a REST-API call. The outcome of the execution is, however, not visualized in the editor.



(a) : Select function to execute

(b) : Execute function detail

Figure 3.4: Script execution

3.2 Implementation of the SPipes Editor

As was already stated, the SPipes editor is a standard web application and hence composed of the backend and the frontend part. The backend is implemented in Scala as the primary programming language, and Java uses the Spring framework and provides a *REST-API* and *WebSocket* communication with the frontend. REST-API(RESTful API in web communication) is an

architectural style for an application program interface that uses HTTP requests to access and use data[71]. WebSocket is a computer communication protocol that provides a full-duplex (two-way) communication channel over a single TCP connection[100]. Also backend side communicates with SPipes engine, which executes SPipes scripts. The frontend side is built on React [69] library and provides means of visualization of the application.

The high level deployment diagram of the application is shown in Figure 3.5.

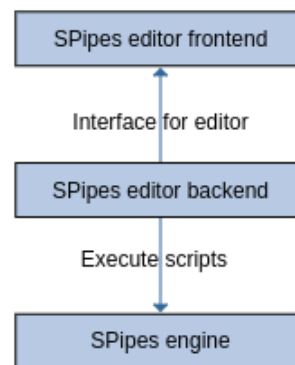


Figure 3.5: High level deployment diagram

3.2.1 SPipes Editor Backend

The design follows the Model-View-Controller (MVC) pattern [59] which is the recommended [60] use of the Spring framework. However, the use of Scala slightly directs the overall shape of the design of the whole backend application. This problem is described in detail in section 3.5.

The Model layer of the application is the only part of the application which is written in Java. The necessity comes from the dependency on the library JOPA (Java OWL Persistence API) which provides object access to ontologies, similarly as ORM frameworks provide an interface to relational database objects. The library is explained in detail in Figure 6.2.1. The functionality is essential for the project and JOPA is one of the leading libraries providing it. Further Model layer communicate via REST-API with SPipes engine.

The Controller layer provides data for the React application and receives data. In most cases the format used for communication is JSON-LD to unite with the Semantic Web domain.

The View layer was originally part of the application and was designed to be run together with the backend on one server. As the communication between layers is performed by REST-API calls, this will be moved to a separate project. It will facilitate greater flexibility and lead to better separation of concerns [57].

Unfortunately, the use of Scala is difficult due to interoperability problems

between Java and Scala. As Spring framework also suffers from compatibility issues with Scala, this led to the decision to rewrite the whole application to Java. The Scala-related challenges are described and discussed in greater detail in section 3.5.

■ 3.2.2 SPipes Editor Frontend

As was already stated in subsection 3.1.1 frontend part of the application is interface for the script editing and execution built on javascript framework - React. The front-end part communicates with the back-end part and handles rendering components such as a .ttl script list or a list of functions. However, it also takes care of rendering, editing the graph using SForms, and notifying the user of a script change on the File System.

Rendering is solved using the React component library The Graph Editor ², which is described in more detail in the section 4.3. This library provides visualization and editing of node-based graphs. It is essential for the application, and the individual requirements for the library are described in the subsection 4.3.4. Unfortunately, the library does not meet some of the conditions and will therefore be replaced in the future.

Another essential component is the SForms library, which provides forms for editing SPipes modules. The library is explained in subsection 6.2.2. The main advantage of SForms is the automatic rendering of the form based on JSON-LD data. The last important part of the application is the WebSocket client, which receives messages about changes in scripts on the File System.

■ 3.3 Tests

As admitted in the bachelor thesis which dealt with the original SPipes editor [104], automated tests were not done properly due to time pressure. Some manual testing was performed; however, manual approach is only sufficient for demo projects. A problematic aspect of manual testing is that the tests are not carried as a repeatable part of code. Furthermore, the tests should serve as holders of the component behavior, thereby allowing programmers to better understand the code. This deficiency of tests is fully described in following chapter.

■ 3.3.1 Backend Tests

The application backend contains, as already emphasized, only a limited number of tests. These tests are not, in the majority cases, related to the code itself. Meaning, while many of these tests monitor the behavior of used libraries, they do not test the code (i.e., classes and its method) used in the project. This absence of sufficient testing of the code is highly impractical as it does not verify the application functionality.

²<https://github.com/flowhub/the-graph>

The only part of the application backend which seems to be tested is services. However, the content of these tests is rather incomprehensible. The content of some of these tests is only mocked, which only increases code coverage but does not bring any benefits. As we can see in Listing 11 the `createFile` method create instance of `File` class and create new empty file on file system. Nonetheless `f.createNewFile()` is part of standard library which expect to be working. Also, the assertion should be part of the test [47]. Another example of an unnecessary test is `jenaTest`, which calls a third-party library and does not assert anything. This type of test is essentially unnecessary and should be removed from the application. An example of this inadequate code is in Listing 11.

```

1  @Test
2  def createFile: Unit = {
3      val script = getClass().getClassLoader().getResource("scripts/")
4      .getFile()
5      val f = new File(script + "test")
6      f.createNewFile()
7      println(f.getAbsolutePath())
8  }
9
10 @Test
11 def jenaTest: Unit = {
12     val m1 = ModelFactory.createDefaultModel()
13     m1.add(
14         ResourceFactory.createResource("http://uri.org/s1"),
15         ResourceFactory.createProperty("http://uri.org/p"),
16         ResourceFactory.createResource("http://uri.org/o1")
17     )
18     val m2 = ModelFactory.createDefaultModel()
19     m2.add(
20         ResourceFactory.createResource("http://uri.org/s2"),
21         ResourceFactory.createProperty("http://uri.org/p"),
22         ResourceFactory.createResource("http://uri.org/o2")
23     )
24     val union = m1.union(m2)
25     union.listStatements()
26 }

```

Listing 11: `OntologyHelperTest.scala` from `s-pipes-editor`³

Other main parts of the backend (model and controller), do not contain any tests. In general, the lack of testing makes applications extremely vulnerable to errors. For example, the edition of a specific node is done via REST-API and this edition had to work during the testing. Nevertheless, further project adjusting had to corrupt this operation. It could be easily prevented if the endpoint, which is responsible for the update operation, would be tested.

The same argument could be used for every other endpoint. Also, this test could serve as documentation of the endpoints. The absence of dynamically generated documentation of the REST-API such as Swagger and typeless endpoints does not provide any information about what is provided by the application.

Lastly, the reason for this unsatisfactory state is not caring about the tests from the beginning of development. The third-party libraries could be simply wrapped and later on mocked. The actual situation is much more complicated because it is very hard to satisfy all the actual dependencies.

3.3.2 Frontend Tests

Automated tests of the frontend part of the application were not established. This section was tested only manually. It means the manual testing of the scenario is required over and over again. Concrete scenarios based on the most common graph manipulations are already defined in the thesis. However, it would be highly convenient for the project if these kinds of tests were automated. For example, a Selenium framework could be used.

3.4 Design Related Issues and Bugs in the SPipes Editor

In this section, we explain issues related to the design of the application, which was described in section 3.2. Some of the components suffer from the problematic design, which could cause unexpected behavior of the program. Further, some of the discovered bugs with explanation are listed.

3.4.1 Design Issues

In this section, parts of the whole SPipes editor design and its individual components which have been evaluated as problematic are described.

Synchronization and Transaction Problem

Although the SPipes editor allows editing of the .ttl scripts for several users, it does not handle the atomicity of individual operations. This issue concerns both reading and writing of the files. This can lead to a race condition - it is an error in a system or process in which the results are unpredictable with the wrong order or timing of its operations. For example, by deleting a node in thread A and editing it in thread B in the same file. Even though thread A deletes the node, thread B may not notice the change and write the node to the file again.

■ Changes in Edited Files Notifications

Changes in files are monitored by a *NotificationController.scala* (NC), which is tied to consumers by a session. Consumers are notified of the change via a web socket. It is important to mention that the name of the component is rather misleading. As it only takes care of notifications when changes appear on the filesystem a different name (e.g., *FileSystemNotificationControler.scala*) would be more accurate.

Another issue with the notifications is that the notify function is incorrectly called in several places in the code. These calls are unnecessary as the root directory with all the files is already monitored using NC. These redundant function calls can result in creation of an erroneous notification or two identical notifications, as you can see in Listing 12. The critical section of the code is *NotificationController.notify(scriptPath)*. This call is not necessary because files are already monitored. To avoid this issue, the *NotificationController.notify(scriptPath)* method should be private or private to the package.

The last problem with the notification component is the management of sessions that receive notifications. The management of sessions is done via Java HashMap in which the file absolute path is the key and the list of sessions are values. However, Java HashMap has no synchronization mechanism. One of the race condition examples is the *register* method is in Listing 13. If two messages for the same file come at the same moment, it is not guaranteed that both of them would be saved. Therefore, when NC interacts with multiple users, the behavior may not be correct.

```

1  def deleteModule(scriptPath: String, module: String): Try[_] = {
2    log.info(f"""Deleting module $module from $scriptPath""")
3    helper.getFileDefiningSubject(module)(new File(scriptPath)) match {
4      case Success(file) =>
5        val m = ModelFactory.createDefaultModel().read(file)
6        m.removeAll(m.getResource(module), null, null)
7        m.removeAll(null, null, m.getResource(module))
8        cleanly(new FileOutputStream(file))(_.close())(os => {
9          m.write(os, FileUtils.langTurtle)
10       })
11       .map(_ => NotificationController.notify(scriptPath))
12     case f => f
13   }
14 }

```

Listing 12: deleteModule method from s-pipes-editor⁴

```

1 @OnMessage
2 def register(script: String, session: Session): Unit = {
3   Try {
4     log.info("Session " + session + " registered on " + script)
5     if (NotificationController.subscribers.keySet.contains(script))
6       NotificationController.subscribers(script) = NotificationController
7       .subscribers(script) + session
8     else
9       NotificationController.subscribers(script) = Set(session)
10  } match {
11    case Failure(e) =>
12      log.warn(e.getLocalizedMessage(), e.getStackTrace().mkString("\n"))
13    case _ => ()
14  }
15 }

```

Listing 13: *register* method from s-pipes-editor⁵

■ Direct Manipulation with File System

Another problematic part is direct manipulation with the file system which is not in line with ACID (atomicity, consistency, isolation, and durability) principles. Implementation of a mediator, so the file system handling would not be direct, could ensure some of these principles.

■ 3.4.2 Bugs

Several bugs were discovered in the current implementation of the application. Namely:

- Update of modules whose output is not used by other modules does not work entirely. The server returns Internal server error (500) and the frontend does not notify the user about the error. The detail of the error in the application log is the *Null pointer exception*.
- Update of the configuration module corrupts the .ttl script which is not valid anymore. Websocket notifies users about the change in the File System; however, after the reloading of the script, the application notifies the user about server Internal server error (500). The analysis of the corrupted script is complicated, because the result is an extremely long file (around 4.5k lines) which is immensely hard to analyze.
- Adding an edge works correctly but it allows the user to create a cycle. However, the SPipes requirements do not allow cycles. It could be easily prevented on the JS side of the application. The Graph library allows cycle detection, thus cycle creation could be easily prevented.

- Adding a new module works as expected from the frontend perspective. However, the *owl:import* property is not longer in the script and the script can not be longer loaded.
- Execution of any function returns Internal server error (500) even if SPipes engine running as described in subsection 3.1.3.

3.5 Problems Related to the Scala Language

In this section, Java and Scala languages in detail to be able to understand their interoperability. Further, the Scala with the Spring framework is described because Scala language is not supported by Spring framework officially. Finally, the last part of the section describes related issues to SPipes editor.

3.5.1 Java

Sun Microsystems created the Java programming language at the beginning of the 1990s and nowadays, it is one of the most used programming languages in the world [7]. While it is used mainly for Internet applications, Java is a simple, efficient, general-purpose, portable language. Following the “Write Once, Run Anywhere” (WORA) paradigm [6], Java applications are typically compiled to Java bytecode that can run on any JVM regardless of the underlying computer architecture. Thus, the same Java program will run on any platform, irrespective of hardware or operating system, provided it has a Java interpreter.

As Java is largely derived from C++, its syntax does not largely differ from C/C++ syntaxes. Java is class-based object-oriented language, thus all data structures are represented as java objects. However, Java cannot be considered to be purely object oriented as it has predefined types as non-objects (primitive types such as integers, floating points numbers, boolean values, and characters) [18]. Unlike C++, Java does not contain pointers and operator overloading (operator ad hoc polymorphism) and multiple inheritance is not enabled. However, definition and implementation of interfaces (i.e. collection of abstract methods) by classes is possible, thus, in some sense, a certain level of multiple inheritance can be achieved. As Java is a statically-typed programming language, prior declaration of the variable data type is necessary.

3.5.2 Scala

Scala is a modern multi-paradigm general purpose programming language which combines features known from functional languages such as Haskell, or Standard ML with object oriented programming. Scala was designed by Martin Odersky from École polytechnique fédérale de Lausanne and was first released in 2003 [17] Many of Scala’s architecture design choices were made

such that it is concise while maintaining readability [48] and were intended to counter critique of Java.

As described in Tour of Scala [39], unlike Java, Scala is a pure object-oriented language in the sense that every value is an object. Objects are defined by classes and traits. Multiple inheritance is in Scala replaced by subclassing and using a flexible mixin-based composition mechanism. Same as Java, Scala implements singleton classes, i.e., classes that can only have one object (one instance of the class) at a time, which provides a convenient way to group functions that are not members of a class.

As mentioned previously, Scala is also a functional language in the sense that every function is a value [39]. Using a lightweight syntax, definition of anonymous functions and higher-order functions is enabled. Nesting of functions and currying (i.e., methods with multi-parameter lists) is also supported. Scala provides the functionality of algebraic types, used in many functional languages, by case classes and their built-in support for pattern matching. General extension of pattern matching using extractor objects then facilitates processing of XML data, thus making Scala highly suitable for web services applications.

Scala is a statically typed language as Java. Scala type system supports generic classes, variance annotations, upper and lower type bounds, inner classes and abstract type members as object members, compound types, explicitly typed self references, implicit parameters and conversions, and polymorphic methods.

The name Scala comes from the words *scalable* and *language* [55]. Thus, the name itself suggests that the language is extensible and provides a straightforward way of implementing new custom language constructs in the form of libraries.

Scala code compiles to the same bytecode as Java, and therefore runs on the Java Virtual Machine (JVM). Scala libraries can be referenced directly from Java [28]. This makes it easier to work with these languages. It also supports multiple inheritance, interfaces, virtual functions, and generic types. The main benefits of Scala over Java include: operator overloading, optional parameters, named parameters, and raw strings. Moreover, Scala allows the use of checked exceptions.

■ 3.5.3 Interoperability between Scala and Java

As previously mentioned, one of the most notable features of Scala is that it runs on the JVM, which brings many benefits in the form of easy use of Java's rich ecosystem with a lot of development frameworks and software [94]. As the compiled bytecode runs on the JVM regardless of the language that was used to produce it, the integration of Java libraries into Scala and vice versa is, in the majority of cases, smooth.

However, some corner cases are known [38] in which issues may arise when using Java code with Scala or vice versa. It is thus important to keep in mind that these languages are not equal. For example, Java does not have an equivalent for Scala's trait which is, on the other hand, distinct from Java's

interface. Interface and trait have many similarities; however scala interface allows implement members.

The full list of interoperability problems is not officially known and as they come from fundamental differences between the two languages, they tend to manifest themselves in various ways. Let's see which issues can arise when using a *ScalaTrait* in a Java code. Suppose we have a Scala trait declaring some function *foo* and a Java class that implements it in Listing 14.

```

1 //Scala code
2 trait ScalaTrait {
3     def foo = {}
4 }
5
6 //Java code
7 class JavaClass implements ScalaTrait {
8 }

```

Listing 14: ScalaTrain implements in Java

As one can see, this leads to an error. Java complains that *ScalaTrait* is not an abstract class and it can not override the *foo()* abstract method in *ScalaTrait*. However, *ScalaTrait* is compiled to Java bytecode, so where is the problem? The key to this question is to understand how the compiled code looks. The equivalent for *ScalaTrait* is shown in Listing 15.

```

1 public interface ScalaTrait {
2     public void foo();
3 }
4
5 public class ScalaTrait$class {
6     public static void foo(ScalaTrait self) {}
7 }

```

Listing 15: ScalaTrain implements in Java

ScalaTrait is compiled to the *ScalaTrait* interface from the Java perspective. Therefore, *ScalaTrait* is interpreted in Java as an interface - which explains the inability of Java to interpret the *ScalaTrain* as an abstract class.

Some other problems such as primitive values, objects in collection or primitive types are described in detail in *Java interoperability: Kotlin vs Scala* [42].

■ 3.5.4 Scala with the Spring framework

As mentioned previously, the Spring framework is a Java platform. Unfortunately, Spring does not have any official Scala support and the only

community developed project Spring Scala⁶ is no longer maintained (the last commit was on October 03, 2015).

However, thanks to the mentioned interoperability between Scala and Java, the Java Spring Framework is usable in Scala as well. Although its application is possible, the differences between the two languages, as outlined in previous chapters, may cause various issues. Alvin Alexander, for example, pointed out quite problematic casting of objects from the Spring application context to Scala [2].

The analysis revealed a number of other issues that, especially due to the lack of official documentation, is time consuming to address. A good practical example can be found in the article *Basic Spring web application in Java, Kotlin and Scala - comparison* [84] by Radosław Skupnik. In this project, a simple Spring Boot app⁷ was developed using Java and alternative JVM languages (Scala and Kotlin) and the approaches have been further compared. The comparison revealed several problematic factors when using Scala to create a simple Model-View-Controller (MVC) application with a trivial Entity, Repository and Controllers. For example, in the Entity class when using the Scala *case class*⁸, it is necessary to use `scala.meta.annotation`⁹ so that the code can work.

3.5.5 Issues in Implementation with Spring Framework and SPipes Editor

In this section, let us briefly describe the Scala-related problems on the backend side of the application. As already mentioned, Spring framework does not have native Scala support which causes an obligation to mix up Scala and Java code in SPipes editor. Some of the issues are:

- **Avoiding return types** - Controller layer is avoiding return types. Even if the DTO is created in Java, the REST endpoint is unable to match it. It makes code incomprehensible and breaks type convention. One of the common approaches to handling exceptions [9] is not working due to the use of Scala.
- **Mixing Java and Scala Collections** - In most cases projects use Java libraries for working with ontologies. These libraries return Java Collections which are later on casted to Scala collections for more convenient work in application. However, to return a response via REST-API, Spring requires a Java collection. This problem manifests itself in many places in the code for Example in ViewService¹⁰: load data wrapped in Java collection, cast them to Scala collection and cast them again to Java collection without any other use in application.

⁶<https://hub.darcs.net/psnively/spring-scala>

⁷<https://github.com/rskupnik/pet-clinic-jvm>

⁸<https://docs.scala-lang.org/tour/case-classes.html>

⁹<https://www.scala-lang.org/api/2.12.0/scala/annotation/meta/index.html>

¹⁰<https://bit.ly/3xgIOKR>

- **Specific libraries for projects** - such as JOPA vocabulary, Jena Ont-DocumentManager cache, and Scala collection wrappers require specific solutions in Scala or implementation in Java to avoid compatibility issues.
- **Mutability** - Scala preferable implementation way are immutable objects and data structures. The immutable objects avoid failures on distributed networks and have thread-safe results. Nevertheless, language still allows the use of mutable objects if they are required.
- **Exception Handling** - Handling of exceptions can also be problematic in case we try to combine Scala with the Spring framework (as described in the subsection 3.5.4). The core of the problem lies in different approaches to exception handling in Scala and Spring. Scala does not require to treat the exception. However, the Spring framework expects the same exception handling as in Java. It means we have to handle every error instead of using fundamental functionality for exception handling by Spring.

■ 3.6 Summary of the SPipes Editor Original State

The application contains several critical bugs that make it practically impossible to use as described in subsection 3.4.2. The script visualization works fine; however, critical errors occur during its editing which damages the script. After that, it is no longer possible to edit it. Combined with unpacking the imports into a given script, it is then virtually impossible to detect the error. This brings us to another very problematic part, and that is the unpacking of imports. This problem makes it very difficult to work further with the script, even though the script is still readable from the machine's point of view. For a human, it is impossible to spot an error.

Unfortunately, script execution doesn't work at all. REST endpoints were provided nevertheless the implementation is not working. SPipes editor requires a running instance of the SPipes engine. Even though the engine is running the script is not executed. Also the possibility of getting results from the execution is not implemented.

I assume that a large part of the application worked during development. Unfortunately, the adding of new features damaged existing functionality. This could be prevented if tests are added from the beginning of the development as stated in subsection 3.3.1. At this stage, it is very complicated to edit existing code or even add new features. The developer does not know if everything is working correctly. Also, the synchronization topic is overlooked subsection 3.4.1. The absence of this topic could cause serious errors in further development.

The last problematic part is the use of Scala as described in section 3.5. With the combination of the Spring framework, it is not possible to use the full potential of any technology. Scala does not enable types of endpoints, creating a model layer must be written in Java, complicate dependency injection and,

last but not least, handle exceptions by REST-API endpoints. In addition, key libraries such as Jopa OWL annotation can not suffice to work with Scala.

Despite the fact, part of the SPipes editor functions are not working application is very well designed. Script module editing via SForms is fundamentally correct. It is working as expected in some cases—unfortunately, the absence of tests very complicated refactoring and understanding the application. In other parts of the application, notifications working as expected in most cases, but the synchronization is not handled correctly. The execution of the script is also well designed, but the implementation is not working.

As an outcome of this analysis, we do not use Scala in further development because it brings more negatives than positives. Also, the tests of the application have to be implemented. The frontend part of the application will be separated from the core of the application to decrease coupling.

Chapter 4

Review of related technologies

This chapter reviews the available graph-based RDF visualization tools and summarizes the visualization technologies. Further, it describes data pipeline editors and compares the visualization libraries and SHACL validation engines.

4.1 Graph-based RDF visualization tools

This section analyses tools capable of visualization of RDF triples via graph shown in Figure 4.1. The aim is to provide a comprehensive overview of available graphic libraries and identify their features and limitations. Herein, only the most crucial part of the libraries - visualization of RDF data as graphs, is described, but other essential features like graph editing, nodes collapsing, and license could be mentioned. The visualization part is a crucial part of the editor; hence the analysis is needed.

4.1.1 Visualization tools

This section briefly describes the majority of the tools that can visualize RDF data. The timeline showing the development of tools for linked data can be seen in Figure 4.1.

- **CytoScape** - Cytoscape is an aggregation, interpretation, and visualization platform for data networks. Several extensions on the App Store in CytoScape, such as **SemScape**¹ and **Vital AI Graph Visualization**², support Semantic web technologies like RDF[78]. The used graphical library for the visualization is Cytoscape.js[27] with an **MIT license**.
- **Fenfire** - Fenfire was an instrument for displaying and modifying RDF graphs to explore the graph interactively. However, the project is no longer active and it does not use any specific library for visualization[33].
- **Gephi** - Although Gephi has been developed to reflect semantic networks, it also works with different networks and other types of graphs. Exter-

¹<http://apps.cytoscape.org/apps/semscape>

²<http://apps.cytoscape.org/apps/vitalaigraphvisualization>

4. Review of related technologies

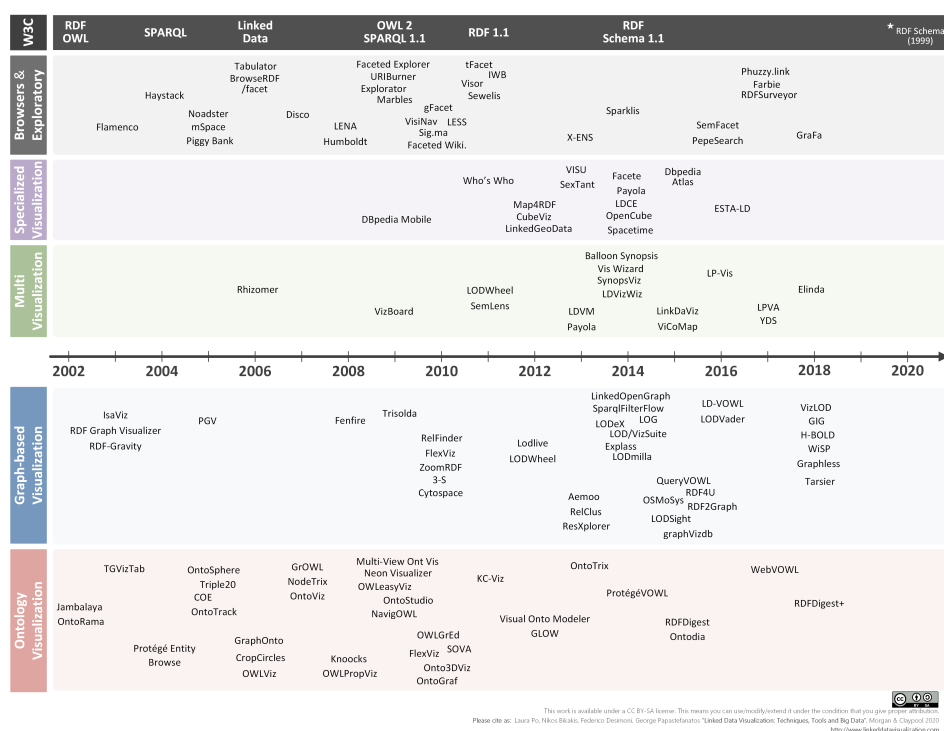


Figure 4.1: Linked Data Visualization Tools Timeline [3]

nal plugins include **SemanticWebImport**³ and **VirtuosoImporter**⁴ support RDF representation. Unfortunately, Gephi is written in **Java** which handles the visualization[11]. It implies that no specific library is used.

- **IsaViz** - IsaViz is a **Java** application for creating RDF data models. Data can be browsed and exported/imported in many different formats, including RDF[41]. However, its development is no longer active as the last version was published in 2007.
- **LodLive** - LodLive project offers browsing RDF data in a very user-friendly web-based GUI. This tool aims to prove how simple understanding robust Semantic Web standards can be and it promotes the spread of massive data. Any LOD Live resource is surrounded by a series of symbols which are various forms of relationships between nodes[19]. LodLive uses a jQuery plugin **lodlive-core.js**⁵, and the project is under **MIT license**.
- **RelFinder** - RelFinder provides an overview of RDF data. It allows extractions, visualization, and interactive exploration of links among provided objects in RDF data[34]. Library is part of **Java Visual Data**

³<https://github.com/gephi/gephi/wiki/SemanticWebImport>

⁴<https://github.com/avens19/virtuosoimporter>

⁵<https://github.com/LodLive/LodLive>

Web group⁶, which is no longer developed.

- **WebVOWL** - WebVOWL is a web framework, which enables to display ontologies interactively. It implements the OWL Ontologies (VOWL) visual notation by supplying graphical representations of OWL components combined with a force-driven ontology graph structure. Interaction techniques allow exploration of ontology and simulation to be adapted[54]. VOWL uses its own implementation built on **D3.js**, which is under **BSD license**.
- **Protégé** - Protégé provides two solutions - Desktop **Java-based** Protégé Desktop and WebProtégé for the Web, which is more convenient for ontology editing[58]. Both solutions support editing of the ontologies. Desktop visualization allows adding or deleting the relations among the nodes, but no specific visualization library is used.

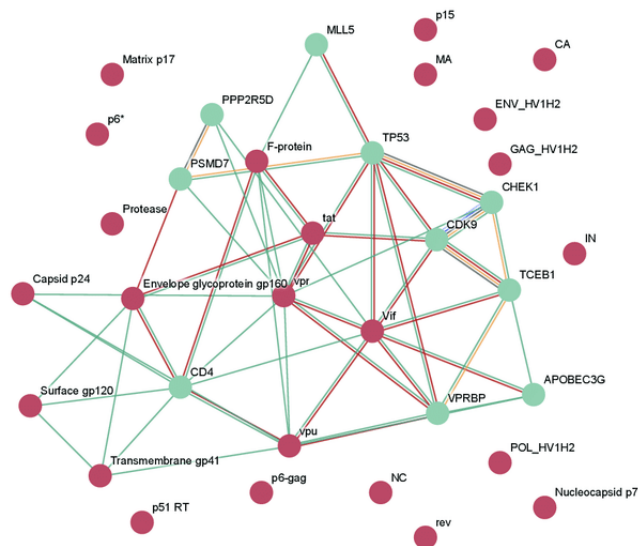


Figure 4.2: CytoScape visualization example [21]

■ 4.1.2 Graph database

A graph database is a database designed to handle data relations as essential as the data themselves[14]. Data could be handled without being limited by the predefined model. Instead, the data is stored as we first draw it - demonstrating how every single object relates or is connected to another object.

Graph databases typically allow browsing data via queries that return text represented results such as JSON[99]. Graphical representation is not so

⁶<https://github.com/VisualDataWeb>

common, but some of the graph databases provide the possibility of graph visualizations. This section analyses only the databases with the graphic visualization option, so it tries to find used visualization libraries.

- **ArangoDB** - ArangoDB is an open-source (**Apache License 2.0**) multi-model database supporting three models of data (graph, document, and key-value data models), enabling users to freely integrate all data models in a single query (ArangoDB Query Language, AQL). Integrating multiple NoSQL types into a single infrastructure using a multi-model database will simplify your architecture. The query language is declarative, allowing various data access patterns to be merged in a single query. While ArangoDB is a NoSQL database, AQL is similar to SQL in many ways[5]. ArangoDB has several options in the area of graph javascript visualization. The most used ones are **D3.js** and **Sigma.js**.
- **DGraph** - Dgraph is a GraphQL [31] database with a graph runtime that is horizontally scalable and distributed under **Apache 2.0 license**. Dgraph is designed to handle the high-volume transactional workloads demanded by today's apps and websites, but it is not exclusive to them. It is used for application backend, search engine, or only for data analysis [24]. DGraph develops its own web visualizer and cluster manager Ratel⁷, which uses **D3.js** for graph visualization.
- **Neo4j** - Neo4j (Network Exploration and Optimization for Java) is a graph database with native graph data structure and processing under **GPL v3 license**. It is a high-performance graph store with all of the functionality expected from a mature database, such as a primary query language and ACID (atomicity, consistency, isolation, durability) transactions [99]. The Neo4j offers two embeddable libraries **Neovis.js** built on **Vis.js**, and **Popoto.js** make on **D3.js**.
- **JanusGraph** - JanusGraph is an open-source, scalable graph database that is customized for work over giant data sets containing billions of nodes and edges. It is licensed under the **Apache 2.0 License**. JanusGraph allows graph data to be distributed to a multi-node cluster as the usual chart database supports competitive transactions and the creation of custom indexes [80]. JanusGraph works with a variety of storage backends (Apache Cassandra, Google Cloud Bigtable, Scylla, etc...). JanusGraph supports **Cytoscape.js** and other visualization applications.
- **RDF4J** - RDF4J is a robust Java framework for handling of RDF data. The functionality of the database is extensive from comprises RDF and Linked Data creation, parsing, reasoning, querying, and others. It comes with a simple API that connects to all meaningful RDF database solutions. It enables you to connect to SPARQL endpoints with the

⁷<https://github.com/dgraph-io/ratel>

repository and build an application that takes advantage of linked data and the Semantic Web[23].

4.1.3 Standalone solutions for data visualization

This chapter focuses on general solutions for data visualization. Graphical visualization tools are an essential and unique source of insights into results. In recent years, researchers in some of the most creative firms worldwide have explored graphic techniques to achieve a more detailed data view [15]. Some of these tools are presented here in order to get insight into which libraries are used for visualization.

- **yFiles** - yFiles from yWorks⁸ company is a programming library for free and commercial usage specifically built for diagram visualization. It is a perfect match for the complexities of graph database visualization. yFiles advanced architecture algorithms can efficiently turn data into a readable and informative network. The visualization is provided by desktop application yed⁹ or website application yed-live¹⁰[103]. Both applications use private visualization libraries.
- **KeyLines** - Cambridge Intelligence¹¹ offers two commercial libraries - javascript library **KeyLines** and **ReGraph**, which allow the use KeyLines for React developers. Both of the libraries provide a powerful API for developers and offer very well documentation. The library used for the visualization is private.
- **GraphXR** - GraphXR a visual analytics platform from KINEVIZ¹² company that gives anybody working with linked, high-dimensional, and big data unparalleled speed, control, and fluidity to explore the graph data in both 2D and XR. GraphXR visualizes data as nodes bound by edges in a graphical 3D graph space and offers a powerful range of tools for exploring and modifying them. The technology used for visualization is **Java-based** private solution.
- **GRAPHLYTIC** - Graphlytic is a web framework for graph visualization, interpretation, and automation that can be customized. The usage of Graphlytic has a variety of purposes, including code refactoring, spam prevention, communication visualization, process analysis, and modeling of IT infrastructure. The library is built on Cytoscape.js[30].

4.1.4 Summary

As we can see in Figure 4.1, graph-based visualization tools are still under development; however, they do not get so much interest as Graph databases

⁸<https://www.yworks.com/>

⁹<https://www.yworks.com/products/yed>

¹⁰<https://www.yworks.com/yed-live/>

¹¹<https://cambridge-intelligence.com/keylines/>

¹²<https://www.kineviz.com/>

or standalone data visualization solutions. Many of the libraries are no longer developed, such as IsaViz, Fenfire, and RelFinder.

The problem of tools designed specifically for use on RDF data is, at the same time, its main feature - the specialization solely on RDF data. Contrary to this, the problem with the solution only for RDF visualization such as WebVOWL is the concretization only on RDF data. In contrast, CytoScape is more general and solves another range of problems for which it provides visualization. Visualization in CytoScape for RDF-based data is solved using the RDFScsape plugin¹³.

Modern graph databases can often work with RDF-based data and then visualize them using visualization plugins. For example, Neo4j can consume RDF data and then visualize and browse it very well. All the above-mentioned graphic libraries have the possibility of high-level visualization, as mentioned in subsection 4.1.2. However, compared to RDF-based visualization tools, Graph databases are more general and RDF-based, and their visualization is only a small part of their whole functionality. Standalone solutions for data visualization focus on general data analysis and possibly their visualization. However, these solutions are often paid for, and visualization is part of the know-how. As shown in Table 4.1 the most used library for visualization is D3.js, Java-based, and Cytoscape.js, where **D3.js** and **Cytoscape.js** offer great visualization options and are in addition OpenSource.

Framework	Visualization tools	Graph database	Standalone solutions	#
Cytoscape.js	CytoScape	JanusGraph	GRAPHLYTIC	3
lodlive-core.js	LodLive			1
D3.js	WebVOWL	Neo4j, DGraph, ArangoDB, RDF4J		5
Java-based	Protégé, RelFinder, IsaViz, Gephi			4
Vis.js		Neo4j		1
KeyLine			ReGraph	1

Table 4.1: Visualization libraries usage in Visualization tools, Graph database, and Standalone solutions

4.2 Data pipeline editors

This section briefly describes existing solutions of data pipeline editors. The used technology and main features are presented.

¹³<https://apps.cytoscape.org/apps/rdfscsape>

- **LinkedPipes ETL** - LinkedPipes ETL is actively developed RDF-based, lightweight ETL tool under **MIT licence**. The tool is composed of REST-API backend and frontend. Both of the components could be built separately, where the backend provides REST-API. It means everyone can create their UI. It is important to mention that the project offers a demo application¹⁴.

The LinkedPipes allows creation of a custom pipeline, where the pipeline is defined as data transformation operations compounded from the modules. The pipelines can be executed and the result downloaded. Also, significant part is the editor, which allows users to create or adjust the existing pipelines. A different variety of features is provided, such as personalizing the UI, pipeline import, or debugging of the pipeline. The UI uses the JointJs¹⁵ visualization library.

- **UnifiedViews** - UnifiedViews offers an editor for managing, debugging, and monitoring ETL pipelines. UnifiedViews is one of the base part of Open Data Node - a publishing platform for Open data. This platform provides extraction, transformation, and publishing open data[50]. The modules of the pipelines are called DPU, which allows load RDF data from the database or call SPARQL query. Unfortunately, the editor is no longer being developed.
- **TopBraid Composer** - TopBraid Composer is a commercial tool from TopQuadrant company. The application is distributed as a Java-based desktop application which allows working with ontologies and RDF data. The tool is primarily oriented to RDF-based data; however, it can work with XML and XML Schemas, JSON, Spreadsheets, and much more. Additional functionality enables the user to create SHACL constraints, read/write by GraphQL, create SPIN constraints, and most importantly, **SPARQLMotion** scripting language, which is the base for the SPipes language. The TopQuadrant offers paid solution *TopBraid Composer Maestro Edition* with advanced features and a free version which has a limited scope of functions [90].

As can be seen from the list of editors, only LinkedPipes ETL and TopBraid Composer are actively being developed. Both solutions offer relatively good documentation and many examples. However, it is necessary to mention that the editors do not get much interest, and no other editors have been created in recent years.

4.3 Visualization libraries

This section introduces the evaluation criteria for the visualization libraries, describes the original analysis, and evaluates the selected libraries.

¹⁴<https://etl.linkedpipes.com/>

¹⁵<https://www.jointjs.com/>

■ 4.3.1 Original evaluation criteria

The original analysis has two types of evaluation criteria for visualization libraries - Critical and Nice-to-have. These criteria are still very relevant to the new SPipes editor. However, the new editor is focused on script debugging. Thus, some requirements will be reprioritized, deleted(~~Strikethrough~~), or added(*italics*).

Critical

- **Automatic layout** - The graph has to be rendered with a readable design as described in subsection 3.1.1.
- **Collapsing** - Nodes could be collapsed(minimalized) into more minor visualization elements for better orientation in large graphs. The sample of collapsing is shown in Figure 4.3.
- **Overall view of graph** - Orientation in the large graph could be the issue; thus, a navigator element with bird's eye view pan and zoom is necessary.
- **Module type icons** - Modules have different types, which another kind of icon could easily visualize.
- ~~Module parameter visualization~~ - This requirement is no longer valid because visualization could be done via external CSS style.
- **Graph live editing** - The library has to be capable of adding/deleting the edge from one node to another. Also, adding/deleting nodes to graphs is required.
- **Custom node arrangement** - When the graph is rendered, nodes could be drag and drop to another position.
- **Node/edge context menu** - Context menu with the possible action simplify usage of the application such as trash icon for the node/edge deletion.

Nice-to-have

- **License** - Library must be under a free license such as Open Source licenses¹⁶. This requirement is promoted to the critical section because further use of libraries in development is crucial.
- ~~Visualization of node/edge state~~ - This requirement is no longer valid because visualization could be done via external CSS style.
- **Documentation** - The library should have actual and updated documentation.
- **Technology and active development** - Technology has to be modern and actively developed - only the javascript-based libraries are considered.

¹⁶<https://opensource.org/licenses>

The original criteria groups (Critical and Nice-to-Have) will not be further used. All of the criteria are marked as mandatory. Nice-to-have criteria are essential for the project; hence, it is unnecessary to create two groups of measures.

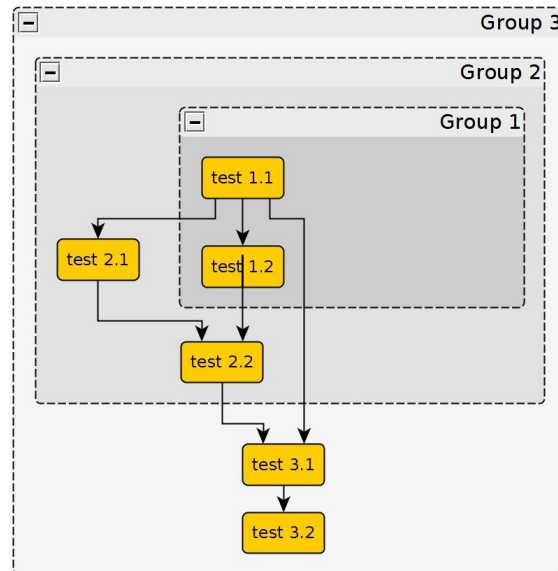


Figure 4.3: Collapsing example

4.3.2 Original analysis results

The initially selected library, The Graph Editor, was chosen based on original evaluation criteria. However, one of the initial requirements (collapsing) is not provided by the selected library. The collapse is a crucial feature because it significantly simplifies orientation in the graph. Moreover, The Graph Editor is not much actively being developed - (Yan's Doroshenko issue - `editor.addErrorNode`)¹⁷ is not still closed or commented. Therefore a new library survey is needed to find the most suitable library that meets all of the criteria.

4.3.3 Libraries analysis

This section briefly describes used visualization libraries from previous analysis of Graph-based RDF visualization tools in subsection 4.1.1. As we know, these libraries can visualize RDF-based data, so we check their properties in the Feature matrix subsection 4.3.4 - acceptance criteria are listed in Original evaluation criteria in subsection 4.3.1. Another source of visualization libraries is from Big Data — Graph Visualisations article[43] and Drawing graph library trend[101]. All of the sources partly overlap; hence we select the most popular ones.

¹⁷<https://github.com/flowhub/the-graph/issues/378>

- **D3.js** - D3 is the most popular drawing graph library on github[101]. It is significantly better rated than the second one, Sigma.js. However, D3 is a general-purpose visualization library, and graph visualization is only its small part. As we can see in Table 4.1 D3 is the top used library among Graph Databases. Moreover, it is used in the very popular RDF visualizer WebVOWL. Excellent documentation is a matter of course; however, the only concern for D3 is its complexity.
- **Sigma.js** - This library is purely dedicated to graph drawing. It is open-source with excellent documentation and a lot of samples. Sigma.js is built on Canvas & WebGL. Sigma.js is focused on colossal graph drawing, and the examples for graph editors are missing. However, Sigma.js is still a compelling library for data visualization.
- **Cytoscape.js** - Cytoscape is used in identically named RDF visualization tool CytoScape, as we can see in subsection 4.1.1. The library allows the import of a wide range of data and their subsequent visualization which shows the vast possibilities of the library. In addition, as can be seen in Table 4.1, it is used in all types of visualizations. Another advantage is the excellent documentation with lots of demo samples and a large community.
- **Vis.js** - Vis.js is a visualization library for visualization of DataSet, Timeline, Network, Graph2d, and Graph3d. The library is developed to be easy to use and manage large amounts of complex data. As we can see in Table 4.1, the Vis.js is used in Neo4j. The official documentation provides many showcases and good documentation. However, the library is not maintained anymore and it is split into many different community modules¹⁸.
- **Dagre.js** - Dagre is mainly used for the lay-out directed graphs. It means the library itself can not visualize data but another visualization tool has to be used for rendering. The supported renderers are, for example, D3, Cytoscape, or JointJs. As Dagre cannot be used for data visualization, it is not considered in the Feature matrix subsection 4.3.4.
- **Keylines** - Keylines is a versatile toolkit for network visualization and browsing. The library is able to visualize a large amount of data in a very efficient way. Nevertheless, the solution is paid, and examples are accessible after registrations. As we can see in subsection 4.3.4, it is used in ReGraph, which is also fee-based. Besides, the library has a limited scope of customization unless it is provided via functions.
- **JointJs** - As we can see in section 4.2, JointJs is used in a LinkedPipes ETL. It proves that the library is fulfills the requirements of the data pipeline editor. The library is well documented with a lot of demos.

¹⁸<https://github.com/almende/vis/issues/4259>

However, the library is composed of two libraries JointJs and Rappid¹⁹. JointJs is free to use but Rappid is fee-based.

- **The Graph Editor** - The Graph Editor is an originally selected React component which provides a wide range of features from a live adjustment of the graph such as drag and drops node, context menu, menu icons, and others - as we can see in subsection 4.3.4. However, the library is no longer being developed and missing the key features such as collapsing.

■ 4.3.4 Feature matrix

Feature matrix check options of selected libraries.

	D3.js	Sigma.js	Cytoscape.js	Vis.js	Keylines	JointJs	The Graph Editor
Automatic layout	✓	✓	✓	✓	✓	✓	✓
Collapsing	✓	✓	✓	?	✓	\$	✗
Overall view of graph	✓	?	✓	✗	✗	✗	✓
Module type icons	✓	✓	✓	✓	✓	✓	✓
Graph live editing	✓	✓	✓	✓	✓	✓	✓
Custom node arrangement	✓	✓	✓	✓	✓	✓	✓
Node/edge context menu	✓	✓	✓	✓	✓	✓	✓
License	✓	✓	✓	✓	\$	✓	✓
Documentation	✓	✓	✓	✓	✓	✓	✓
Technology and active development	✓	✓	✓	✗	✓	✓	✗

Table 4.2:

✓ - supported feature

\$ - payed feature

? - not exactly support requirements

✗ - not supported

■ 4.3.5 Evaluation of the results

As we can see in the matrix, only two libraries, D3.js and Cytoscape.js, meet all of the requirements. However, it is important to mention that some other libraries, such as Sigma.js or Vis.js, are able to meet most of the requirements, if the Overall view of graph requirement is not taken into account. All of the other libraries are also valid options but some work would have to be invested into their extensions. However, extension of the libraries is not the subject of this thesis.

¹⁹<https://resources.jointjs.com/docs/rappid/v3.3/index.html>

The finally selected library is **Cytoscape.js**. In terms of features, D3.js and Cytoscape.js are equal. The D3.js has the only problem with the general purpose of this library. Its generality could be problematic when looking for a specific feature which could be too broad for other solutions. Cytoscape.js is only one-purpose library for graph visualization, which makes it suitable for our application. Also, in terms of performance, Cytoscape.js library is optimized for larger graphs.

■ 4.4 Validation

The original concept of the semantic web expected that the data could be published without any restrictions and the application would be able to handle them using derivation based on ontologies. It was proven that checking the integrity of data before using it in the application is necessary in most cases [79]. This section introduces the SHACL language and compares its execution engines.

■ 4.4.1 SHACL

The Shapes Constraint Language (SHACL) is an RDF-based language for validating RDF graphs against a set of constraints. These criteria are given in the form of an RDF graph as shapes and other components. In SHACL, these RDF graphs are referred to as "shapes graphs," while RDF graphs that are verified against a shapes graph are referred to as "data graphs." SHACL shape graphs are used to check that data graphs meet a set of criteria; they may also be thought of as a description of the data graphs[75].

■ 4.4.2 SHACL execution engines

The SHACL language only defines the vocabulary and different implementations of this language exist. Implementations vary based on the programming language or multiple implementations for a specific language such as Java. However, the implementations are not always equaled. For example SHacLEX the Scala implementation of SHACL does not have equal comparison results as dotNetRDF [77; 76; 25].

However, we are primarily only interested in Java implementations where the main ones are *Apache Jena SHACL* and *TopBraid SHACL API*. Both of the implementations provide a command-line interface or could be imported as Java dependency. The main difference is that TopBraid is implemented on Apache Jena, thus it is actually an extension [4; 91]. The libraries do not behave equally and could return different results in the evaluation. The problematic behavior of Jena SHACL implementation is demonstrated on Listing 16. Jena evaluation engine can not properly deal with *sml:ApplyConstruct*, which is an imported class. Thus it does not correctly evaluate the result. On the other hand, TopBraid SHACL is working as expected. Hence the **Top-**

Braid SHACL API will be used in implementation as SHACL execution engine.

```
1 @prefix : <http://onto.fel.cvut.cz/ontologies/shapes/form/> .
2 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3 @prefix sh: <http://www.w3.org/ns/shacl#> .
4 @prefix sml: <http://topbraid.org/sparqlmotionlib#> .
5 @prefix sp: <http://spinrdf.org/sp#> .
6
7 :apply-construct-check
8   a sh:NodeShape ;
9   rdfs:comment "sml:ApplyConstruct must have at least 1 sml:constructQuery
10  with exactly one sp:text value"@en ;
11   sh:targetClass sml:ApplyConstruct;
12   sh:property [
13     sh:path sml:constructQuery ;
14     sh:minCount 1 ;
15     sh:property [
16       sh:path sp:text ;
17       sh:minCount 1 ;
18       sh:maxCount 1 ;
19     ] ;
20   ] ;
21 .
```

Listing 16: SHACL rule example

Chapter 5

Requirement Analysis

This chapter describes the features of the new editor and introduces terminology for accurate clarification of requirements. The MoSCoW prioritization technique is given for the requirements, which determines the importance of individual requirements. Further, the chapter analyzes functional and non-functional requirements for tasks or actions that the SPipes editor must meet. Finally, use-cases are introduced to help determine the boundaries of the system.

5.1 New SPipes Editor

The original functionality is described in section 3.1 where the editing and execution of scripts, and notification of changes of individual files is explained. As described later in subsection 3.4.2, preliminary analysis shown that neither the script editing nor execution works correctly in specific scenarios. The notification about script changes works without any problems; however, its behavior could also be improved to resolve the synchronization issues.

The goal of the new editor is to reimplement and correct its original functionality with the main focus on advanced features to manage SPipes scripts. The new editor introduces validation and it is more focused on debugging the scripts. Validation will be used to check semantic constraints of the SPipes language and custom best-practice rules for writing scripts. Debugging will allow defining test-cases to validate the pipeline, set up inputs and output of modules manually, reuse outputs of previous executions, or query execution history.

Some of the requirements which will be introduced in this chapter overlap with the original implementation; however, as mentioned in section 3.6, previous SPipes editor contains several critical bugs that are difficult to solve in the absence of tests. We considered the rewrite problematic parts of a code and added tests; however, the code is not written in a testable way; thus, a rewrite would be very complicated. Further, Scala interoperability complicates the rewrite. Problematic cases will be distinctly marked in subsection 5.2.4 or discussed in more detail in chapter 7 to clearly identify what precisely has been changed and what has been kept the same.

Moreover the terminology is introduced for a better clarity of the require-

ments in Figure 5.1. Terminology that defines how the script will be executed is colored as orange, while terminology related to a concrete execution is colored as red.

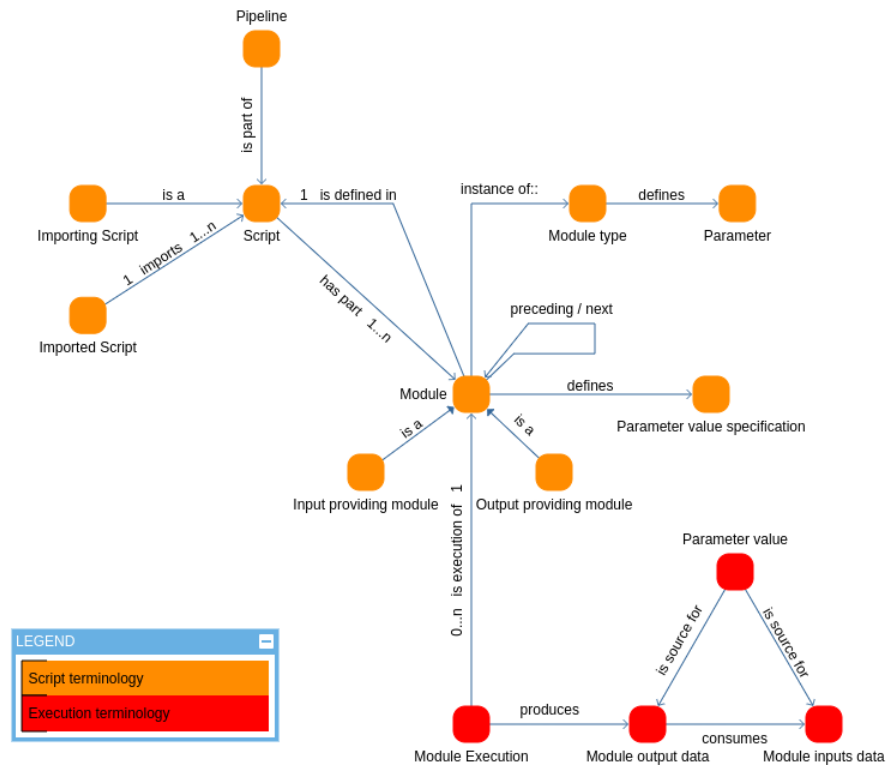


Figure 5.1: SPipes language terminology

5.2 Analysis of the SPipes editor requirements

This chapter introduces a prioritization technique for evaluating functional and non-functional requirements. These requirements will help to precisely define the criteria for the new system and bring its functionality closer [26]. The requirements were obtained after consultation with the supervisor of the diploma thesis, analysis of the original thesis, LinkedPipes ETL and TopBraid Composer.

5.2.1 Prioritization Technique

MoSCoW prioritization introduces a clear definition of categories for each type of requirement regarding their importance to the final product. The prioritization method removes the element of personal preferences. It replaces it by assessing the impact of the implementation/non-implementation of the functionality on the usability/benefit of the resulting solution. The name of the MoSCoW prioritization method is derived from the naming of individual

priorities of the requirements of the created product/service, which are sorted according to importance[16]:

- **(M) Must Have:** These requirements represent the so-called Minimum Usable Subset (MUS), i.e., the minimum requirements that the project must deliver. If these requirements are not implemented, the project/solution as such will lose its meaning. If another solution to the requirement can be found, it is a Should Have or Could Have requirement, even if it is less practical. Moving a request to the Should or Could Have category does not mean that it will not be delivered, but only that delivery is not guaranteed[56].
- **(S) Should Have:** A requirement that is very important and should be part of the solution even if only partially. This requirement is important but not critical for the question of the completion or continuation of the project. It may be disadvantageous to omit this requirement, but the solution will still make sense.
- **(C) Could Have:** Requirements marked Could are desirable but not necessary. They usually help to improve user-friendliness or customer satisfaction with minimal development costs. These requests are typically implemented when there is time left.
- **(W) Won't have (this time):** Requests that have been agreed to be out of scope for the current time window. It is necessary to distinguish between the requirements that will be taken into account for the next time window and permanently excluded.

■ 5.2.2 Functional Requirements

Functional requests define the necessary tasks or actions that an application must perform[87]. Requests are uniquely identified by $FRx (Y)$, where x is the request number and the Y is priority based on MoSCoW. Further, in a more complex task, the activity diagram is used or more precise clarification. An activity diagram is one of the UML diagrams that describes behavior of a system. This diagram is used to model procedural logic, processes, and workflow capture [26]. Each process in the activity diagram is represented by a sequence of individual steps which are plotted in the model as an action or nested action.

Furthermore, special symbols and colors are introduced for more precise clarification of requirements.

Symbols

- ✓ - fully met requirements
- * - partially met requirements
- not marked* - not met requirements

Colors

- Completely implemented
- Partial implementation
- Not implemented

■ **Script adjustment**

This section describes requirements related to work with the scripts.

FR1 CRUD of scripts

- FR1.1 (M) System allows the user to create a new script✓
- FR1.2 (M) System allows the user to list a scripts in a particular folder✓
- FR1.3 (M) System allows the user to visualize script as a graph✓
- FR1.4 (M) System allows the user to update a script✓
- FR1.5 (S) System allows the user to delete a script✓
- FR1.6 (C) System allows the user to move a script to another directory
 - This requirement will be fully implemented in the prototype version except for FR1.6, which requires a better visualization component, which allows the drag and drops feature. The workaround for move script operation is direct manipulation within the file system, which is recommended only for advanced users.

FR2 CRUD script modules

- FR2.1 (M) System allows the user to create a new module with specific module type✓
- FR2.2 (M) System allows the user to update a module✓
- FR2.3 (M) System allows the user to delete a module✓
- FR2.4 (M) System allows the user to show module parameter✓
- FR2.5 (S) System allows the user to read a module source code related to a module✓

- This requirement is essential for the project, and it has to be implemented in the prototyped version. The CRUD modules operations were implemented in an original editor; however, it does not work as expected. The problematic behavior is explained in detail in subsection 3.4.2.

FR3 CRUD Module parameter value

FR3.1 (M) System visualize module parameter value✓

FR3.2 (M) System allows the user to add a parameter value*

FR3.3 (M) System allows the user to remove a parameter value*

FR3.4 (M) System allows the user to update a parameter value✓

- This requirement is vital to work with Module, and it has to be implemented in the prototyped version. The CRUD operations allow defining the behavior of the Module, which gives a whole meaning to SPipes editor.

FR4 Module type parameter template

FR4.1 (S) System defines mandatory module type parameters for every module✓

FR4.2 (C) System defines optional module type parameters for every module

- FR4.1 helps the user to fill up the necessary parameters for module functionality. Thus, this requirement will be implemented in the prototype version. The FR4.2 requires modification in SPipes library and visualization, so it will not be implemented in the prototype version.

FR5 Search for a module type

FR5.1 (S) System allows the user to search for specific module type✓

- As the number of modules types increases, an easy search of available modules type is necessary. For this reason, this requirement will be implemented in the prototyped version.

FR6 Manage script imports

FR6.1 (M) System allows the user to show imported scripts✓

FR6.2 (S) System allows the user to add existing ontology✓

FR6.3 (S) System allows the user to delete imported ontology✓

FR6.4 (S) System allows the user to rename ontology✓

FR6.5 (S) System recursively propagates the ontology adjustment to other scripts✓

FR6.6 (C) System checks if all the scripts are still valid import after adjustment*

- FR6.1 is a crucial requirement for the project so that it will be implemented in the prototype. Other requirements improve the reusability of the scripts; thus, they will be part of the implementation.

FR7 Module relations

FR7.1 (M) System allows the user to add an input/output module to a module✓

FR7.2 (M) System allows the user to delete a input/output module to a module✓

FR7.3 (S) System not allows the user to add dependency which create a cycle

- The modules relations are essential for executions, so they will be part of the prototyped version. The FR7.3 is related to 5.2.2 and it will not be implemented due to high time requirements.

FR8 Module transfer in related scripts

FR8.1 (C) System allows the user to move a module to a related script via modal dialog✓

FR8.2 (W) System has defined integrity constraints for a module move

FR8.3 (W) System allows the user to move a module to related script via drag and drop

- The transfer of modules helps users in module arrangement so that it will be implemented in the prototyped version. FR8.2 and FR8.3 will not be implemented due to low priority.

■ Script visualization

FR9 Module type identification

FR9.1 (M) System renders a module with a distinct icon based on the module type✓

FR9.2 (S) System allows the user to change default module types icons*

FR9.3 (C) System allows the user to assign a custom icon to module

- FR9.1 simplifies orientation in large graphs and identifies modules so it will be implemented in the prototype. The FR9.2 could be changed in the configuration file, but the prototype does not provide any configurable visualization. The last requirement

could be beneficial in some cases when the user has to identify a particular module. Still, it will not be in the prototype because of the low priority of the requirement.

FR10 Module grouping and collapsing

- FR10.1 (S) System displays a module in groups based on a file✓
- FR10.2 (S) System allows user to collapse modules based on a file✓
- FR10.3 (C) System allows user to select multiple modules and collapse them
- FR10.4 (C) System allows user to create and assign a module to a custom group*
- FR10.5 (S) System distinguishes the groups hierarchy*
- FR10.6 (S) System allows a module to be assigned to exactly one group✓
- FR10.7 (C) System allows modules to be assigned to more than one groups
- FR10.8 (S) System allows user to redirect to module script✓

- The grouping and collapsing facilitate orientation in large graphs. The basic grouping will be implemented in the prototyped version. The FR10.4 will be partly implemented based on custom assignment to the group in the script. The FR10.3, FR10.5, FR10.7 will not be implemented due to lack of support on selected visualization library.

FR11 Script overview

- FR11.1 (S) System allows user to view a graph from bird's eye perspective✓

- Script could contains more then 100 modules and the orientation in this graph could be problematic. Thus FR11.1 will be implemented in the prototype version to simplify the orientation in large graphs.

FR12 Script rendering

- FR12.1 (M) System allows user to render a graph in predefined layouts✓
- FR12.2 (S) System allows user to fix the positions of the module*
- FR12.3 (S) System allows user to arrange a custom layout of the graph✓

- For better orientation in the graph, it is necessary to provide the user with intelligent visualization of the graph to implement the prototype.

FR13 Notification

FR13.1 (M) System notifies the user about an adjustment on the currently edited script and offers to refresh the page✓

FR13.2 (S) System notifies the user about a graph appearance adjustment and offers to refresh the page because the script could be edited by a different user or directly in the file system.

FR13.3 (C) System automatically re-renders the graph after the adjustment

- The FR13.1 will be implemented to allow multiple users to edit scripts. The FR13.2 and FR13.3 will not be implemented due to lack of time.

■ Execution and debugging

FR14 Script execution

FR14.1 (S) System allows the user to list script execution history✓

FR14.2 (S) System allows the user to show an execution report✓

FR14.3 (S) System allows the user to download a module execution input/output✓

FR14.4 (M) System allows the user to list a function✓

FR14.5 (M) System allows the user to execute a function✓

FR14.6 (S) System allows the user to see the successful and the unsuccessful status of the execution*

FR14.7 (C) System allows the user to see the progress of execution

FR14.8 (S) System can handle different versions of the same script

- **FR14** is one of the thesis requirements, so that will be implemented in the prototyped version. The FR14.7 requires modification on the SPipes engine side, so it will not be implemented in the prototyped version. FR14.8 will not be implemented due to its complexity and lack of time. The design of the function execution is proposed in Figure 5.2 and described in 7.3.

FR15 Module execution

FR15.1 (S) System allows the user to show a module input/output✓

FR15.2 (S) System allows the user to show variables assigned to a module✓

FR15.3 (S) System allows the user to edit a module input/output✓

FR15.4 (S) System allows the user to execute a module with a mocked input✓

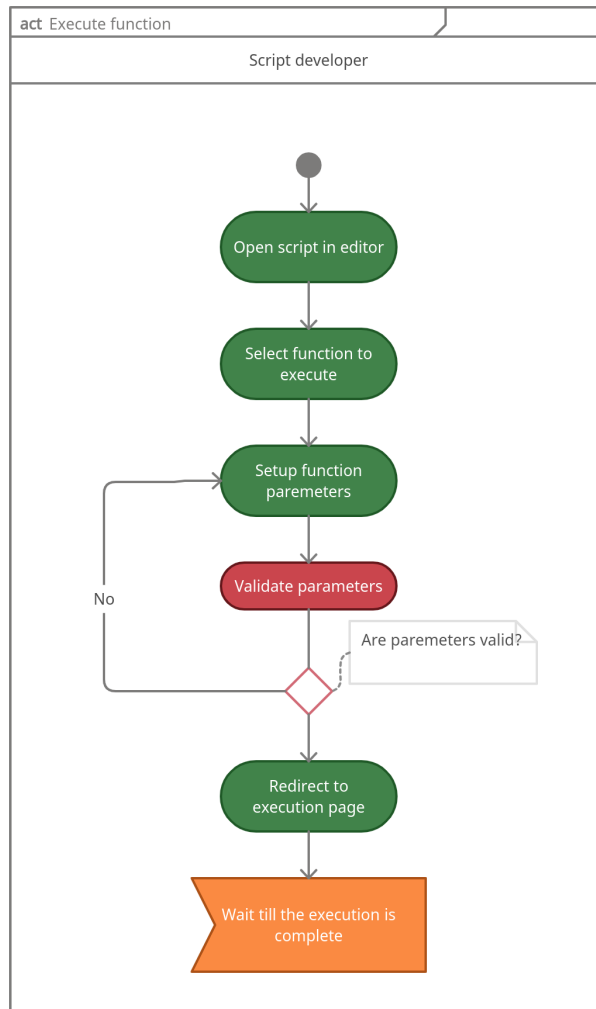


Figure 5.2: Function execution

FR15.5 (S) System allows the user to execute a module with a indirectly mocked input

- This requirement is an essential part of the debugging process and will help the user debug a specific module. FR15.5 will not implement due to unimplemented functionality on the SPipes engine side. The design of the debugging of the module is proposed in Figure 5.3 and described in 7.3.

FR16 Module execution status

FR16.1 (S) System distinguish by color the status of the module after execution

- This requirement will not be implemented due to a lack of time.

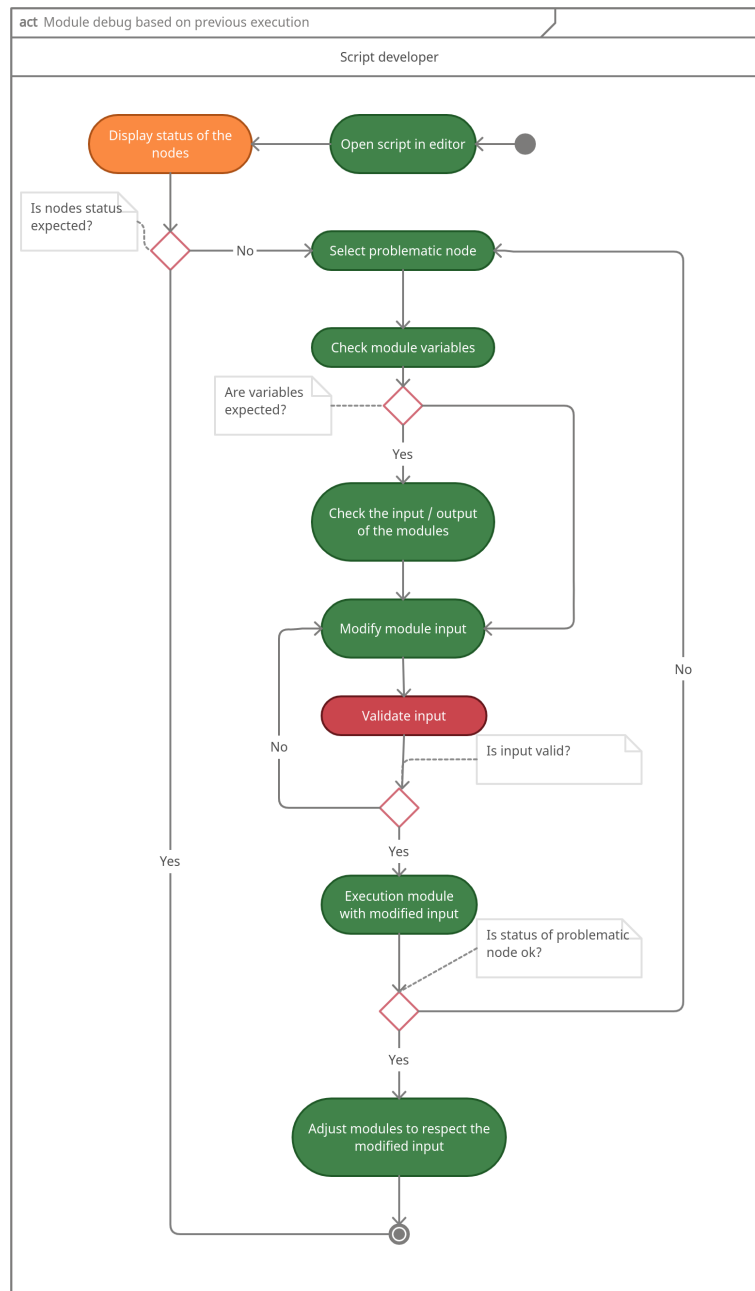


Figure 5.3: Module debugging proposal

■ Validation

FR17 Script integrity constraints

FR17.1 (M) System has defined set of custom best-practice rules to write scripts✓

FR17.2 (S) System allows the user to show a validation report✓

FR17.3 (S) System allows the user to navigate to an invalid module✓

- All requirements are crucial to validation so that they will be implemented in the prototyped version. The design of the validation is proposed in Figure 5.4 and described in 7.3.

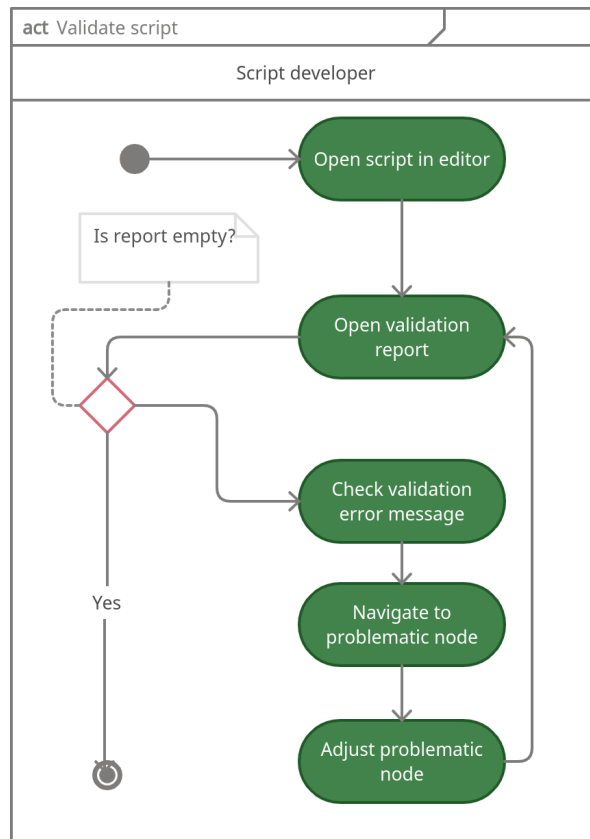


Figure 5.4: Script validation

FR18 Module integrity constraints

FR18.1 (M) System has defined set of semantic constrains of the SPipes language✓

FR18.2 (C) System does not allow the user to upload invalid module data

- The FR18.1 is part of the thesis assignment, so it has to be implemented in the prototyped version. The FR18.2 has to check integrity constraints before uploading, which not be implemented due to lack of time.

Others

FR19 Browser compatibility

FR19.1 (C) System detects unsupported browser

- The requirement will not be implemented due to low priority.

FR20 Newest version of SPipes library

FR20.1 (S) System has always newest version of SPipes library ✓

FR20.2 (S) System informs user if newer version of SPipes library exists*

- This requirement will be implemented in the prototyped version.

FR21 System synchronization

FR21.1 (S) System handles the transactions and does not enter into an unexpected state*

- The synchronization prevents the unexpected states of the application so that it will be implemented in the prototyped version.

5.2.3 Non-Functional Requirements

This chapter introduces non-functional requirements (NFR) that will be taken into account during the development. Non-functional requirements are an addition to functional requirements. They describe other necessary properties needed for the environment and context [29]. Requests are uniquely identified by $NFR_x(Y)$, where x is the request number and the Y is priority based on MoSCoW.

Logging

NFR1 (M) The application will save the log to the application server log file. CRUD operations with entities and any unexpected states of the application will be logged. ✓

Localization

NFR2 (M) All user environments of the application will be localized into English. ✓

Configuration

NFR3 (M) Configuration of all the applications will be accessible via an external file. ✓

Licensing

NFR4 (M) All components of the system used will meet the license conditions for the free creation and subsequent operation of the application. ✓

Compatibility and portability

NFR5 (M) The application will offer a responsive user interface adapted for display on tablet devices and desktop computers. The user interface will be compatible with Mozilla Firefox version 89, Chrome version 91.✓

NFR6 (M) System will use the newest version of used technologies.✓

Maintainability

NFR7 (M) At least 75% server part of the application will be covered by unit tests.✓

NFR8 (M) The application will be tested by user testing at least on 3 people.✓

NFR9 (M) All of the service will be containerized for easy execution of the whole application.✓

Performance

NFR10 (M) System will allow concurrent use of the application at least for 5 people.✓

NFR11 (M) The loading of the component will not take longer than 5 seconds.✓

Deployment

NFR12 (S) The entire application will be easy to run and ready for production deployment✓

■ 5.2.4 Use-cases

The usage diagram captures the external view of the modeled system and thus helps to reveal the boundaries of the system and serves as a basis for range estimates. It is a sequence of related transactions between a participant and a system during a mutual dialogue. The primary purpose is to capture the actors who communicate with the system and the relationships between services and customers. In both ways - visually and textually, understandable to system developers and customers.

The system has two groups of users.

- **Script viewer:** A user typically wants to run only a script and is not expected to have a more profound knowledge of creating scripts. The user should be familiar with the individual SPipes modules.
- **Script developer:** An expert user who can execute, debug and create SPipes scripts. The requirement from the user is to know SPipes module types and the SPARQL language.

As mentioned in section 5.2 the colors are used for better clarification; however, the original SPipes editor is partially overlapping with the new editor, so it is vital to determine what the original functionality was and what is new. Thus the state of the original cases is marked by a colored border, and the current state fills the body. It is essential to mention the black color border, which is not part of the color palette(introduce in subsection 5.2.2), is working as an expected state. The list of the requirements is in Figure 5.5.

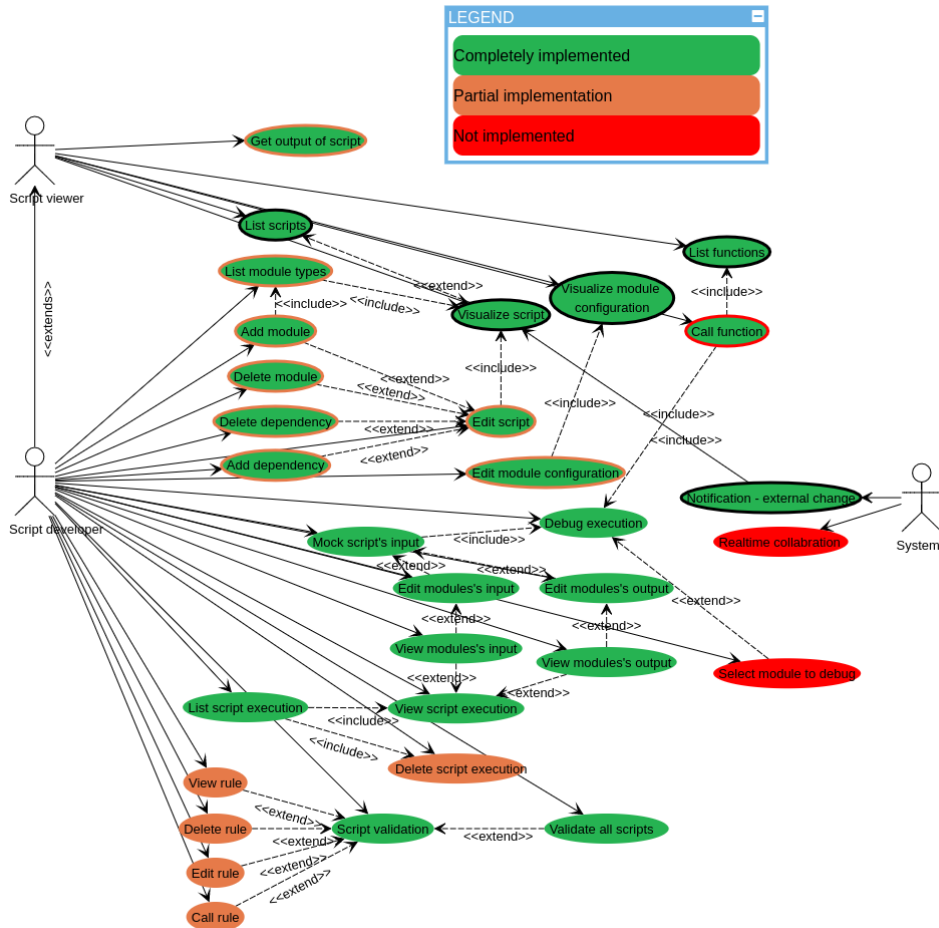


Figure 5.5: Use-case diagram

Chapter 6

Architecture Design and Technologies

In this chapter, the reasons that led to the choice of used technologies and the application architecture design are described. The presented solution of the application takes inspiration from the legacy S-Pipes editor, see chapter 3. Final architecture aims to extend the legacy architecture with the respect to the new functionality of the editor. Due to the acquired requirements, see chapter 5, and analysis of the related technologies, see chapter 4, a web application was determined to be the most suitable solution option.

6.1 Application structure

This section explains the application design and the functions of the main components. The system is divided into two main parts – the server part, which takes care of the application logic, and the front end, which handles the UI. In this section, only the application design is explained – the implementation detail is described in chapter 7. The interconnection of the whole system is described in more detail in section 7.2 which deals with the dockerization of the entire system.

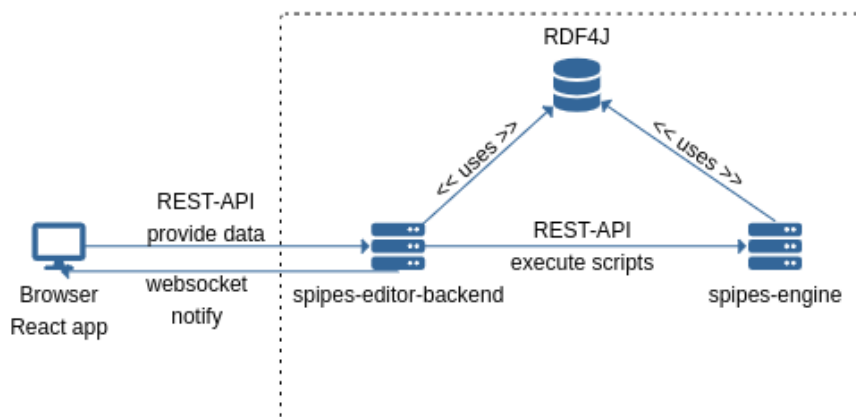


Figure 6.1: Application structure

- Spring Boot¹** - Spring Boot framework is an extension of the Java Spring framework with more straightforward configuration and a built-in application server. This significantly reduces the required configuration, and the resulting application is effortless to run. Spring Boot is based on the MVC architecture providing many modules that offer a wide range of services such as Authentication and authorization or Data access, which enables working with relational databases or NoSQL databases. Another essential feature is the support of testing via unit tests and integration tests [98]. The basic architecture of the Spring Boot is shown in Figure 6.2.

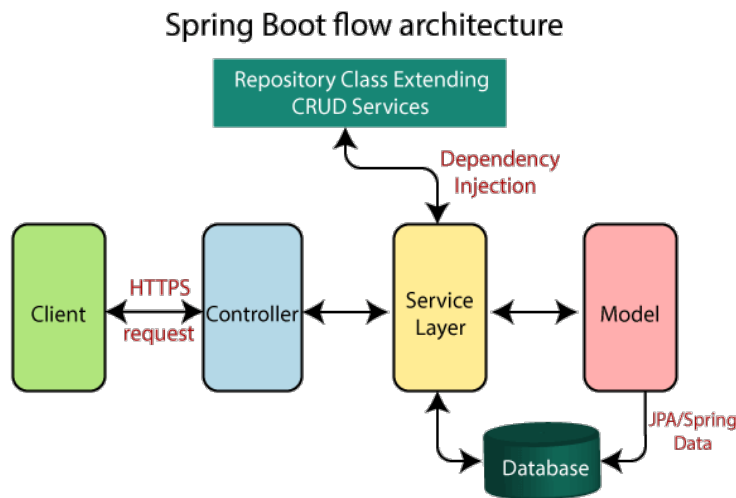


Figure 6.2: Spring Boot Flow Architecture [85]

- Apache Jena²** - Apache Jena is an open-source Java framework for working with semantic web and Linked Data. Jena offers a wide range of APIs to work and process RDF data [4]. The concept of the semantic web is already explained in chapter 2, which is required to work with Jena. It gives access to querying and updating RDF models using SPARQL, comparing two RDF models, and much more. We introduce only a small part of the Jena functionality, which is relevant to the thesis.

We firstly introduce the representation of a *triple* as a *Statement*. It allows user to access the *subject*, *predicate* and *object*. The *Statements* are accessible via the *Model* interface, which is a set of *Statement* and it provides a CRUD operation.

The second significant part is the loading of RDF-based files such as SPipes scripts. As described in section 2.5, the SPipes contains the *owl:import*; thus, Jena has to be capable of working with imports. Class critical for working with the scripts is *OntDocumentManager*, which loads all necessary files, and it is able to represent script data as *Statements*.

¹<https://spring.io/projects/spring-boot>

²<https://jena.apache.org/>

- **JOPA**³ - JOPA is a Java OWL persistence framework for programming access to OWL2 ontologies and Java RDF graphs. In order to achieve the contract between a JOPA-operated Java application and an OWL ontology, the system is built on the OWL integration restrictions [53]. The System and API architecture follows the JPA 2.1, which allows specifying the details of the table in the database [44]. The example of both technologies is shown in Listing 17 and Listing 18, which maps a table *PERSON* with *id* and *name* properties.

```

1      @Entity
2      @Table(name = "PERSON")
3      public class Person {
4          @Id @GeneratedValue
5          @Column(name = "id")
6          private int id;
7
8          @Column(name = "name")
9          private String id;
10     }

```

Listing 17: JPA example

```

1      @OWLClass(iri = "person")
2      public class Person {
3          @Id(generated = true)
4          private URI uri;
5
6          @OWLDataProperty(iri = "person_name")
7          private String name;
8      }

```

Listing 18: JOPA example

- **RDF4J** - RDF4J is an open-source framework for working with RDF data. It provides storing, querying, and analyzing the RDF data. The supported query language is SPARQL.

6.2.2 Client side

The client-side is build on the Javascript React framework; the most notable technologies are mentioned. The technology which is not discussed here is CytoscapeJs which provides the visualization, as it is already described in subsection 4.3.3.

³<https://kbss.felk.cvut.cz/web/kbss/jopa>

- **React**⁴ - React is a JS open-source library from Facebook for creating a user interface (UI). It focuses on one specific area, and if we look at it from the point of view of the classic MVC architecture, it forms only the *view* layer that presents data to the user [69].

The basic building block consists of *components* that are various reusable HTML elements with encapsulated functionality, the assembly of which creates a complex UI application. These components then have their own properties and manage their internal *state*. This declarative way of working with application data leads to more predictable behavior and easier debugging.

- **Webpack**⁵ - The primary purpose of Webpack is to work with javascript modules, create packages for the browser, and facilitate the work of developers. It is an open-source module bundler for JavaScript. When we have javascript code written in a modular way, Webpack processes it and creates a js package from it. Also, it can import assets that are not JavaScript into the modules, such as images, styles, etc.

Webpack also provides a development server that facilitates development.

- **SForms**⁶ - SForms allows to visualize a dynamic form from the provided ontological data in the JSON-LD format. Data contains the collection of the form's fields and the behavior of the form. Formula functions include dividing a form into several steps, which are displayed based on filled fields. Furthermore, it supports various input types such as Input/Textarea, Datetime picker, Checkbox, etc.
- **react-treebeard**⁷ - React-treebeard is a React component for directory structure visualization. In the project it the library visualizes the directories with the scripts.

6.3 Design of non-trivial requirements

This chapter describes solutions for non-trivial parts of the application, which refer to functional requirements in subsection 5.2.2.

6.3.1 Execution

This design issue is related to the Script execution (**FR14**) and module execution (**FR15**). The problem is related to the SPipes engine. The dependencies of every module are displayed in Figure 7.4. As it can be seen, if we want to debug the script, the engine has to have access to the location of the scripts, or we have to provide the scripts to the engine in some other way.

⁴<https://reactjs.org/>

⁵<https://webpack.js.org/>

⁶<https://github.com/kbss-cvut/s-forms>

⁷<https://github.com/storybookjs/react-treebeard>

6.3.3 Script validation

The open-world nature of the RDF language concept implies that it is difficult to enforce a module schema. The introduction of validation via SHACL is described in section 4.4. The proposed solution is to define best practice rules to help users check the correctness of the module.

The proposed design of the validation can be seen in Figure 5.4. The main feature is an overview of all problematic modules with a short description of the problem. Further, the user could localize the module and modify it. That is, the user could iteratively improve the quality of the questionable modules.

6.3.4 Modules grouping and collapsing

This section describes the issues related to modules grouping and collapsing (**FR10**). This problem is illustrated in Figure 6.3, where four files are displayed, each with a different color. The yellow file is the main script that is visualized. This file imports a green and a blue file, and these two files import a purple file. Our goal is to find a solution that can respect this fact in the resulting visualization.

One of the possible solutions for the file is to remember all of the imported files, which are collapsed during the collapse operation, recursively for all their imports. However, it is important to mention the technology restriction on the side of the selected Cytoscape.js library. Each node in the graph can belong to exactly one group, so it is not possible to enter information directly into the graph that the yellow file imports blue and green. However, it is possible to add anonymous nodes to the graph that are able to reflect this fact. However, this solution requires a complex implementation modification on the part of the library.

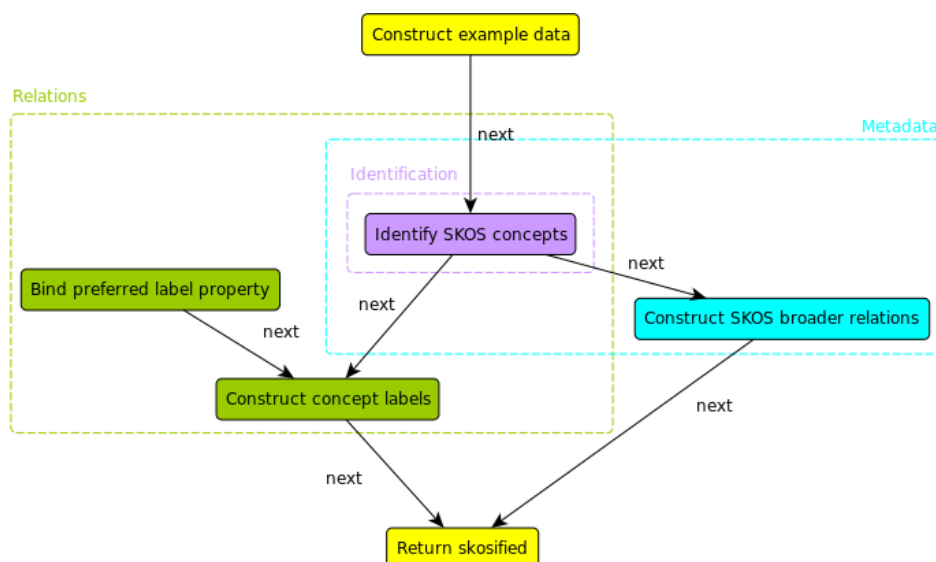


Figure 6.3: Skosify script grouping and hierarchy example. Source is SPipes documentation.

6.3.5 Module transfer

Transferring a module in a related script (**FRS**) could be highly problematic and could corrupt script that is no longer valid and executable. An example of a complex operation is moving a module that is imported in multiple scripts. Thus, if the module *Identify SKOS concepts* was moved to the green file, the blue file would lose this module because it does not import the green file. Therefore, this move would have to import the green file into the blue script, and the purple file would be useless. However, it is necessary to realize that the purple file can be used in other scripts, and the issue would have to be addressed in them as well. Thus, this problematic move is considered invalid.

An example of a non-problematic operation is to move the yellow module *Construct example data* to any file. The problem could only occur if this module used a different script, but we do not assume this case in this example.

The result of the design is to make non-problematic operations automatically valid and inform the user about complex operations that could cause an invalid state of the scripts. As described before, the result of the procedure could be an empty file. Also, the circle reference of the imports could be an issue.

The sample valid transfer of the *Construct example data* from Figure 6.3 is shown in Figure 6.4. The problematic scenario 6.4b shows a transfer of the purple module *Identify SKOS concepts* to a different script. The problem is similar to importing a class in Java. The purple module does not know it is imported to the green or blue file. Thus, if we transfer the purple module to the blue script, the green file loses track because it only imports the purple file, which does not contain the *Identify SKOS concepts* anymore.

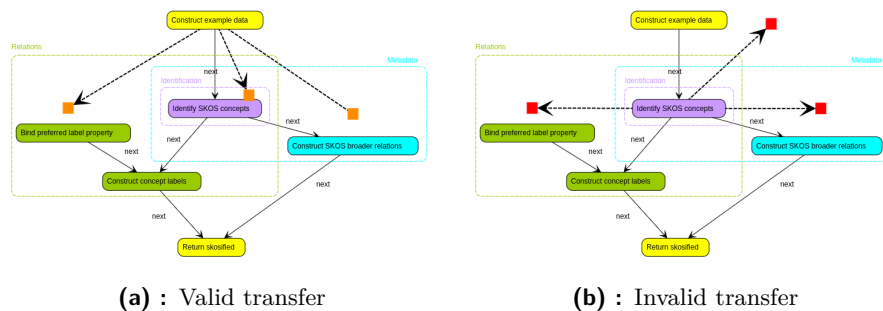


Figure 6.4: Module transfer. Source is adjusted from SPipes documentation

6.3.6 Script to form

The original solution has already been proposed in the legacy SPipes editor. This is an implementation problem rather than a design problem. All of the issues are closely related to the SForms framework and SPipes scripts. Firstly the module is taken, and its properties are parsed into *Question* class, which is the SForms class for data representation, and serialized class is visualized in SForms. The form is rendered to the user who can update it. The form is

submitted, the response is parsed to a *Question*, and changes are written to the module.

■ 6.3.7 The actual state of the system for multiple users

The basic design of this solution was designed in the legacy editor in subsection 3.1.2. The UI part of the application could receive messages about the changes from the server in real-time via WebSocket. An improvement is the introduction of the type of adjustment. If a graphical change occurs, the system redraws the graph instead of refreshing the page(FR13.1).

Chapter 7

Implementation

This chapter describes the technology stack update in the original editor. Further, the design for testing the application is explained, simplifying development by replacing the Spring framework with Spring Boot and the layout of the individual pages in the UI. Docker technology is further presented for easier deployment and deployment of the new editor. Next, the communication of particular services behind the reverse proxy is described in detail. The last part describes the implementation of the requirements defined in chapter 5.

7.1 Legacy SPipes editor update

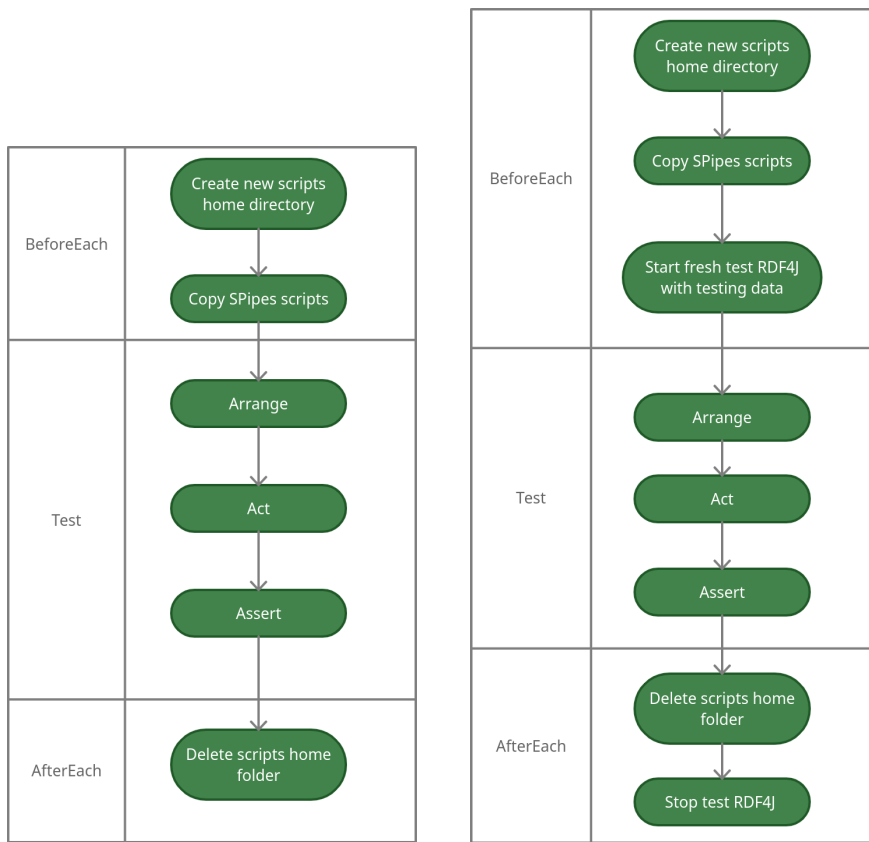
This section describes technology updates on both the server and the client-side. Further, the topic of testability and simplification of the development is described. Lastly, the layout of the UI is introduced.

7.1.1 Technology stack update

The current application architecture differs from the original one, introduced in subsection 3.2.1, primarily in the implementation detail. As described in subsection 6.2.1, the main difference is the use of Java by which the interoperability issues mentioned in section 3.5 are excluded.

Another significant change is the replacement of the Spring framework with the Spring Boot framework. Spring Boot is just a Spring extension to make development, testing, and use more convenient [8]. This change is mainly reflected in the development and deployment. Other library update was Jopa from 0.13.1 to 0.13.5. The most important libraries are libraries supporting junit5 and mockito testing libraries. The project focuses on the application tests from the beginning of the development. Thus every component had to be written in a testable fashion, more in chapter 8.

A fundamental change occurred on the UI side. It was separated out as an independent application and completely re-implemented. The applied technologies are described in section 6.2. The modified UI is built on the React technology, identically as the original UI, but has been updated from version 15.3.2 to 16.13.1, which fixes minor bugs and offers better error handling



(a) : Basic schema of the integration test

(b) : Test with database

Figure 7.1: Life cycle of the integration test

7.1.3 Development simplification

As stated in subsection 7.1.1, the Spring framework is replaced by Spring Boot. This replacement allows the developer to run a project without caring about the execution server. Further, the testability of the application is one of the main requirements, so the user can freely change the code and see the consequences of the changes [57]. The last in simplification is the introduction of Docker, which is described in detail in section 7.2.

7.1.4 UI Layout

The proposed design of the application is divided into three parts:

- **Scripts** - User can list the folders with scripts and perform CRUD operations on the scripts.
- **Executions history** - User can see history of the execution and navigate into its detail.

- Scripts detail** - User can edit the script, execute function, check execution and validate script.

The Scripts detail page is shown in Figure 7.2. In the upper part, there is a menu bar with Scripts and Executions bars, which redirect the user to other pages. The left side contains operation for *Add module*, *Function call* for calling script function, *Graph render strategy* allows different render strategies. Further, the buttons are self explanatory - *Execution report*, *Manage script's ontology* and *Validate report*. The last component is *Variables info*, which displays the name of called function and values of variables based on *mouseoverhttps://www.w3schools.com/jsref/event_onmouseover.asp* action. The right side contains the script as the graph visualization.

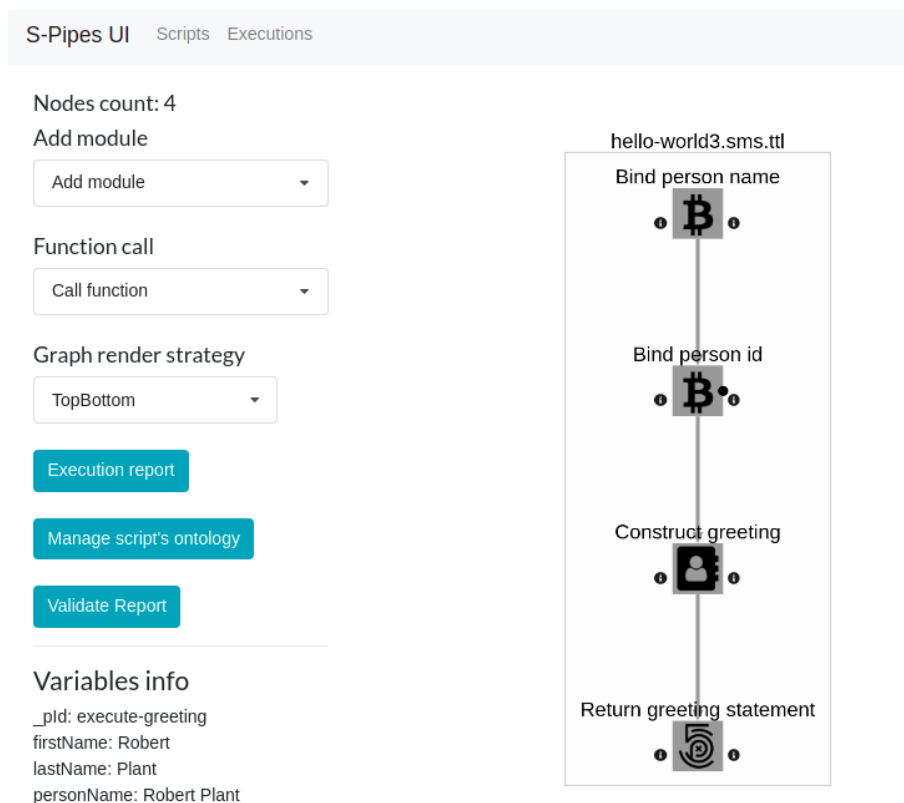


Figure 7.2: Hello-world3.sms.ttl script execution detail

7.2 Dockerization of SPipes editor

This chapter introduces the reader to the Docker technology, i.e., dividing each service into a container. It also explains the deployment of the entire system and the communication of individual services with each other. The last section explains the concept of a reverse proxy.

7.2.1 Docker and Docker compose

Docker is an open-source project to automate the deployment of applications as portable and custom containers that can run in the cloud or locally. Its goal is to provide a unified interface for isolating applications into containers in mainstream operation systems macOS, Linux, and Windows.

The container contains only the required applications and their specific files but does not contain a (virtualized) operating system. This significantly reduces overhead compared to traditional virtual machines. Therefore, Docker's advantage is a much smaller size, greater flexibility, and thus lower operating costs. On the contrary, the disadvantage is the connection with the guest operating system, which is directly used for running applications in containers [61].

The example of the Dockerized app, which could run in different environments, is shown in Figure 7.3.

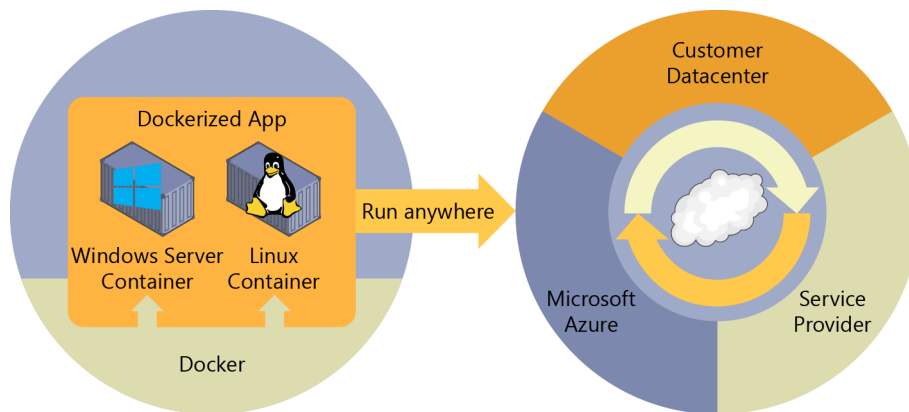


Figure 7.3: Deployment to the cloud with Docker [61]

7.2.2 Containers and Docker compose

The SPipes editor requires two other services: *RDF4J* and the *SPipes engine* with a rather complicated configuration. Further, the *SPipes engine* is based on the Spring framework, which requires an application server such as Tomcat and the *RDF4J* for logging the executions. Lastly, React requires the nginx¹ server.

In order to solve the complex configuration, all of the services are dockerized, and the whole application is started via the Docker compose. Docker compose is a tool for defining and running multi-container Docker applications specified in *docker-compose.yml*. It shows configuration options for each service and enables running of the SPipes editor.

The proposed architecture of the system and communication between the containers is shown in Figure 7.4 and the *docker-compose.yml* is shown in Appendix A. The SPipes editor UI is marked as *spipes-editor-ui*, the SPipes

¹<https://www.nginx.com/>

editor as *spipes-editor-rest* and the SPipes engine as *spipes-engine* and RDF4J as *RDF4J*.

The Docker compose immensely helps with the parametrization of the project. Moreover, it allows the user with local execution and further development.

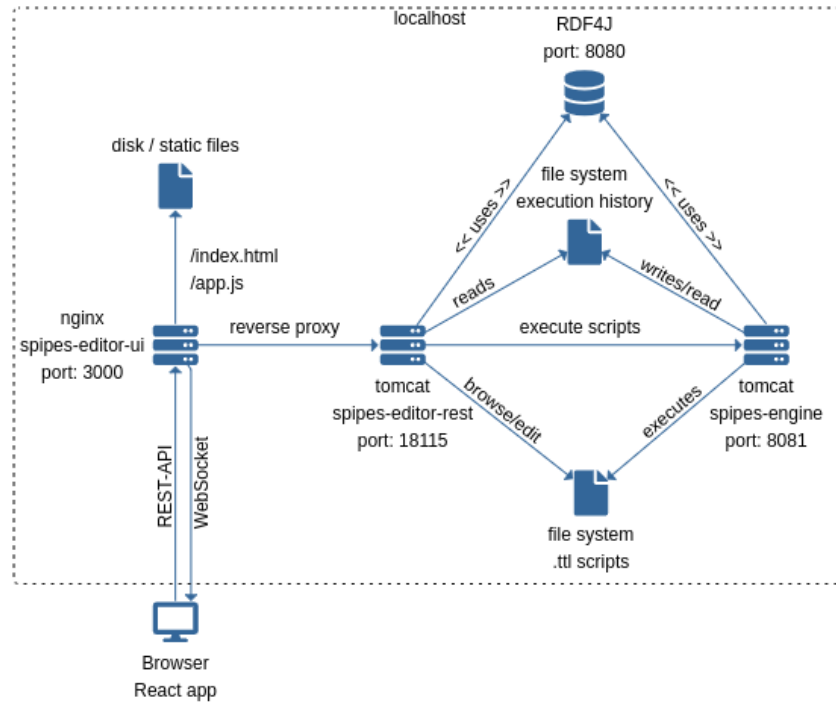


Figure 7.4: Deployment diagram

7.2.3 Reverse proxy

In order to avoid security risks such as CORS² attack and simplification of local development, a Reverse proxy was chosen. A reverse proxy divides incoming traffic across multiple servers, maintaining a single external interface for the client [70].

7.3 Requirements implementation

This chapter describes the implementation of the functional requirements described in subsection 5.2.2 and the non-functional requirements described in subsection 5.2.3. The structure of every requirement begins with the status of implementation of every sub-requirement.

²<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

7.3.1 Implementation of Functional Requirements

Every functional requirement has a following structure: the description starts with an overview, explains where it could be found in UI, if possible, and describes the implementation detail.

Script adjustment

FR1 CRUD of scripts

- **Implemented** - FR1.1, FR1.2, FR1.3, FR1.4, FR1.5
- **Not implemented** - FR1.6

The home page consists of a list of folders and scripts displayed in 7.5a. The list of scripts is received via a REST-API endpoint. The structure of the files is represented by *react-treebeard*, described in 6.2.2. Each folder has an *onClick* action that displays a modal dialog with possible actions that are listed in 7.5b. Furthermore, the user can create a new script in the folder or delete an existing one.

The FR1.6 (moving a script to another directory) lacks support on the *react-treebeard* side. The framework does not provide a drag and drop feature. The workaround is simple, and it is described in **FR1**; however, the direct manipulation with files is always dangerous as we can no apply any integration constrain and could break the pipeline.

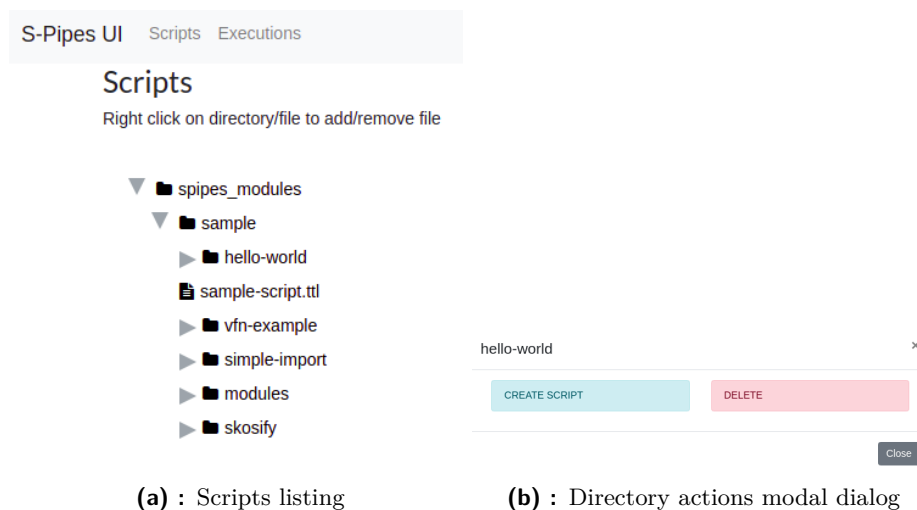


Figure 7.5: Scripts page

FR2 CRUD script modules

- **Implemented** - FR2.1, FR2.2, FR2.3, FR2.4, FR2.5

The module creation FR2.1 is possible via *Add model* with a dropdown menu (on the left side in 7.2), where the user selects a module type and the *SForms* modal window shows up after the selection. The dropdown menu allows the user to filter the desired module type which is related to **FR5**. The *SForms* modal is shown in Figure 7.6. The *SForms* has two modes. The graphical mode in which the user can read the properties of the module as input fields. The second option is an editable text representation of the module in Turtle, which is recommended for advanced users only.

The module can be removed via the *cytoscape-cxtmenu*³ plugin, as shown in 7.7.

All module-related data are received via REST-API endpoints.

The screenshot shows a modal window titled "Modal heading" with a close button (x). The main content area is titled "Module of type Apply Construct (http://topbraid.org/sparqlmotionlib#ApplyConstruct)". On the left, there is a sidebar with a blue header "Module of type Apply Construct (http://topbraid.org/sparqlmotionlib#ApplyConstruct)" and a "TTL" button. The main form contains the following fields:

- URI:**
- Label:** (URI: http://www.w3.org/2000/01/rdf-schema#label)
- SPARQL Query:** (URI: http://topbraid.org/sparqlmotionlib#constructQuery)
- Replace:** (URI: http://topbraid.org/sparqlmotionlib#replace)

At the bottom right of the form is a "Next" button. Below the modal window are "Close" and "Save Changes" buttons.

Figure 7.6: SForms modal widows

FR3 CRUD Module parameter values

- **Implemented** - FR3.1, FR3.4
- **Partly implemented** - FR3.2, FR3.3

All of the adjustments in the module parameter values are related to *SForms*, which is shown in Figure 7.6. The modal window is triggered by a click on the cogwheel icon shown in Figure 7.7. The visualization (FR3.1) and update (FR3.4) can be directly performed by input variables. The creation of the form is described in subsection 6.3.6. The add or remove parameter values is marked as partly implemented because it is possible only via direct

³<https://github.com/cytoscape/cytoscape.js-cxtmenu>

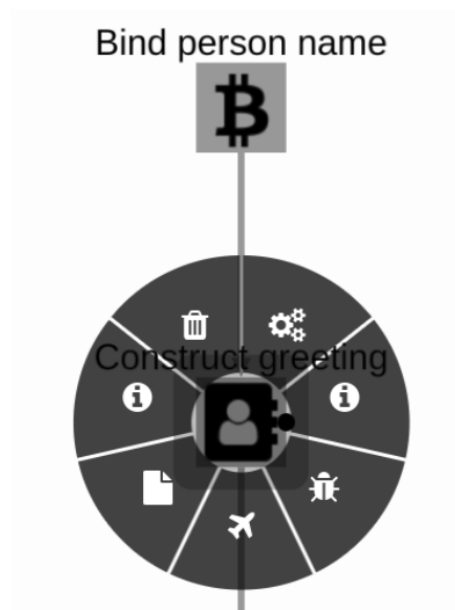


Figure 7.7: Modul menu

edition of the Turtle representation of the module. The access to the turtle representation possible by the *TLL* bar on the left side, and it is recommended for advanced users only.

The implementation uses the *SForms* library, which receives parsed module in JSON-LD via REST-API.

FR4 Module type parameter template

- **Implemented** - FR4.1
- **Not implemented** - FR4.2

Based on the request with a module type, the server provides data for *SForms*. It is necessary to mention that the *SForms* only visualizes data but it does not handle the logic of the required input fields. The method of the module data generation is described in subsection 6.3.6.

The optional fields (FR4.2) require adjustment in both libraries, and it was not implemented due to the lack of time.

FR5 Search for a module type

- **Implemented** - FR5.1

The *Add module* element on top left side of Figure 7.2 is a dropdown element from Semantic UI React⁴, which provides a filtering option.

⁴<https://react.semantic-ui.com/modules/dropdown/>

- **Partly implemented** - FR9.2
- **Not implemented** - FR9.3

The Cytoscape allows assigning a custom icon to a node, which can represent different module types. FR9.2 does not have a convenient graphical interface, but the configuration for module type and icon is hardcoded in the configuration. This configuration can be changed; however, it is recommended for the advanced user only. The custom icon is not implemented due to low priority.

FR10 Module grouping and collapsing

- **Implemented** - FR10.1, FR10.2, FR10.6, FR10.8
- **Partly implemented** - FR10.4
- **Not implemented** - FR10.3, FR10.5, FR10.7

Grouping and collapsing is described in subsection 6.3.4. Modules are grouped according to the file that can be collapsed. An example of such a group is in Figure 7.2. Collapsing into a single node is possible by hovering the mouse over the group, where a plus icon appears in the upper right corner. Navigation to the module source script is possible using the file icon in Figure 7.7. Collapse is implemented using the Cytoscape extension *cytoscape-expand-collapse*⁶.

Grouping based on multiple groups is unfortunately impossible due to the Cytoscape library, which allows to assign only one group to a node.

FR11 Script overview

- **Implemented** - FR11.1

Cytoscape has *cytoscape-navigator*⁷ extension, which provides a component with an overview of the graph. The overview of the graph is in the right corner.

FR12 Script rendering

- **Implemented** - FR12.1, FR12.3
- **Partly implemented** - FR12.2

The render strategies options are located on left side *Graph render strategy* in Figure 7.2. The possible options are from *TopBottom*, *LeftRight* and *Custom*. FR12.2 is implemented by a *Custom*; however, the default choice for all nodes is the same. This means that for larger graphs, all nodes are in the same

⁶<https://github.com/iVis-at-Bilkent/cytoscape.js-expand-collapse>

⁷<https://github.com/cytoscape/cytoscape.js-navigator>

place, which causes confusion when rendering. The rendering options can be set by Cytoscape.

FR13 Notification

- **Implemented** - FR13.1
- **Not implemented** - FR13.2, FR13.3

If the user opens the script, the server saves his session. When a change occurs in the file, the user is notified via WebSocket. The change appears as a modal window on the UI side, informing that the script has been changed and offering to refresh the page.

It is necessary to mention that the session-user pair is stored in *SynchronizedMap* to avoid the synchronization problems described in subsection 3.4.1.

■ Execution and debugging

FR14 Script execution

- **Implemented** - FR14.1, FR14.2, FR14.3, FR14.4, FR14.5
- **Partly implemented** - FR14.6
- **Not implemented** - FR14.7, FR14.8

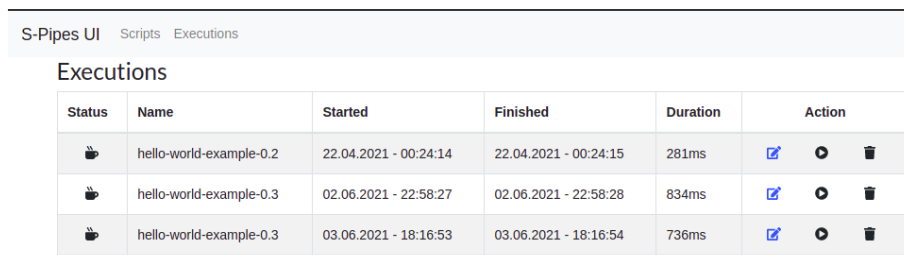
The list of the executions (FR14.1) is shown in Figure 7.8. The execution report (FR14.2) and the execution and listing of the function (FR14.5, FR14.4) is located on the left side labeled as *Function call* in 7.2 and button *Execution report*. The execution input/output (FR14.3) can be downloaded via the report.

The status of the execution (FR14.6) is visible if the execution is finished without an error; however, if the execution fails due to a fatal error, the execution can not be written in the database. More robust writing of the execution logs is required in the SPipes engine.

The flow of the execution is shown in Figure 5.2. The implementation is straightforward - the SPipes backend calls the function execution with parameters via the SPipes engine. When the execution is complete, the engine logs the data to RDF4J.

FR15 Module execution

- **Implemented** - FR15.1, FR15.2, FR15.3, FR15.4
- **Not implemented** - FR15.5



Status	Name	Started	Finished	Duration	Action
	hello-world-example-0.2	22.04.2021 - 00:24:14	22.04.2021 - 00:24:15	281ms	
	hello-world-example-0.3	02.06.2021 - 22:58:27	02.06.2021 - 22:58:28	834ms	
	hello-world-example-0.3	03.06.2021 - 18:16:53	03.06.2021 - 18:16:54	736ms	

Figure 7.8: List of executions

The module input/output from a execution is visible via the Figure 7.7 by clicking on the *info* icon. The left one is for the input and the right one for the output. Variable assignments are located on the left side in Figure 7.2 labeled as *Variables info*. It provides a piece of quick information about the assigned variables and what was executed. The module input and the parameters could be edited, and the module could be debugged.

Indirect module execution is not implemented because of the missing functionality on the SPipes engine part. However, the necessary preparation on the UI side is complete.

Validation

FR17 Script integrity constraints

- **Implemented** - FR17.1, FR17.2, FR17.3

The list of the best practices is shown in Table 7.1. The rules are related to the SPipes language and best practices for writing scripts. The validation report is shown in Figure 7.2 on the left side as a *Validate Report*. On click, the button shows up in the modal window with a list of errors and it allows the user to navigate the problematic module. The module is also coloured red for better recognition. The script validation progress is shown in subsection 6.3.3.

FR18 Module integrity constraints

- **Implemented** - FR18.1
- **Not implemented** - FR18.2

The module integrity constraints are merged with best practices for script writing. The approach is the same as in **FR17**. The list of the best practices is also shown in Table 7.1.

Others

FR20 Newest version of SPipes library

- **NFR1** System logs all of the events via the *SLF4J*⁹ library.

Localization

- **NFR2** All of the parts of the UI are in English language.

Configuration

- **NFR3** The configuration is achieved via the 7.2.

Licensing

- **NFR4** All of the libraries meet the license conditions.

Compatibility and portability

- **NFR5** UI is working on Mozilla Firefox version 89, Chrome version 91. Further the display on tablet and monitor is supported.
- **NFR6** The newest technologies were introduced in section 6.2.

Maintainability

- **NFR7** This requirement is explained in chapter 8. Testing is a vital part of the application, so it was taken care of throughout the development.
- **NFR8** The user testing is performed in chapter 8 based on defined scenarios.
- **NFR9** The containerization is discussed in section 7.2 and it was developed during the development process.

Performance

- **NFR10** During the development, more than five requests were sent to simulate the behavior of multiple users.
- **NFR11** All of the methods are implemented in efficient way, so the performance is not problem.

Deployment

- **NFR12** The deployment of the application is described in section 7.2

⁹<http://www.slf4j.org/>

Chapter 8

Testing and Evaluation

This chapter describes and summarizes the final quality of the unit tests of the application. Further, the browser testing of the editor is introduced. Next, user testing is prepared with the predefined scenarios, evaluation, duration of the testing process and summarizes findings from the users. The next part is devoted to a comparison with the original SPipes editor. The comparison firstly compares the implementation differences; secondly, the features between are compared.

8.1 Testing

As stated in the original work, there was not enough time for testing [104]. The topic of tests is also discussed in section 3.3. The new editor tries to follow the principles of Test-driven development (an approach to software development where test cases are created to specify and verify what the code will do [57]) and have the written code tested. Testing is primarily focused on the server side of the application.

The React part of the application is not tested due to time constraints and lack of experience with the React framework on the author's side.

This section discusses the quality of unit tests, browser testing in different browsers, and various devices. In the last part, the user testing with its evaluation is described.

8.1.1 Unit tests

The term *unit test* in the context of information technology refers to the automatic testing and verification of the functioning and correctness of the system implementation [37].

The **NFR7** requires unit tests that will cover at least 75% of the server part of the application. However, it is not simple to choose the methodology for the evaluation. The metrics are based on *Class*, *Method* or *Line* coverage [35]. Moreover, more complex methods such as *pairwise testing*, also known as *all-pairs* testing, is possible [51]. The selected one is *Line coverage*, which is the percent of lines executed by this test run.

The final test result is displayed in Figure 8.1. The average of the tested lines is 62%; however, without the *model* package it is slightly more than 77%. The *model* package is used to access the persistence layer or data transfer classes. This points to the problem of the line coverage methodology, where the developer is forced to write a test just to have line coverage. These classes do not need to be tested.

Element	Class, %	Method, %	Line, %
config	100% (4/4)	100% (7/7)	100% (22/22)
model	93% (27/29)	41% (103/248)	37% (209/560)
persistence	85% (6/7)	74% (29/39)	72% (142/197)
rest	87% (7/8)	75% (27/36)	68% (101/148)
service	81% (13/16)	69% (70/101)	78% (447/573)
shacl	100% (3/3)	90% (10/11)	97% (37/38)
websocket	100% (3/3)	76% (13/17)	69% (37/53)

Figure 8.1: Test coverage

8.1.2 Browser testing

As stated in **NFR5** the application will offer a responsive user interface adapted for display on tablet devices and desktop computers. The UI is tested on *Ubuntu 20.10* in following browsers *Google Chrome 91.0.4472.77* and *Mozilla Firefox 90.0*. The width is set to 1600px, and everything is working as expected in both browsers. The table of tested browsers is shown in Table 8.1

Further, the application is tested via the Chrome developers tools¹, which enable mock tablet. The selected table is *iPad Pro*, which is working as expected.

Device	OS	Browser	Width	Supported
PC	Ubuntu 20.10	Google Chrome 91	1600px	Yes
PC	Ubuntu 20.10	Mozilla Firefox 90.0	1600px	Yes
Mobile	Ubuntu 20.10	Google Chrome 91	1024px	Yes

Table 8.1: Browser support

8.1.3 User testing

This section describes the selected users for testing. Further, the scenarios for testing are presented. The last part collects answers from the participants.

Selected users

The information about the participants is shown in Table 8.2. The following list explains information in a table:

¹<https://developer.chrome.com/docs/devtools/device-mode/>

- High education includes the participants with bachelor or higher degrees.
- The Ontology knowledge and Coding skill has the following options:
 - **None** - The None level of ontology knowledge or coding skills means that participants do not know anything about the domain.
 - **Basic** - The Basic level represents the junior level in the domain. Participant knows basic of some programming language and knows basic about the semantic web.
 - **High** - The High level represents rich practical experiences in the domain.
- Job describes the background of the participant profile. If the Private is filled, the participant does not wish to answer.
- The Installation has two options:
 - **Prepared** - The running instance of SPipes editor is prepared on the author's computer
 - **Own** - Participant runs own instance following the guide in Appendix B

	Education	Ontology knowledge	Coding skill	Job	Installation
P0	High	Basic	None	Private	Prepared
P1	High	None	None	Ph.D. student of VSCHT	Prepared
P2	High	Basic	High	T-mobile full-stack developer	Prepared
P3	High	Basic	High	Ataccama full-stack developer	Own

Table 8.2: Participants profile

■ Introduction

Before the testing process the basic information about SPipes editor is explained to participant.

The tested application is an editor for developing and debugging SPipes scripts. As you can see, an example of such a script on Figure 8.2 is visualized as a graph in the right part of the image, and it contains the critical component for use called Pipeline. This Pipeline defines a transformation process that is composed of Modules. The Module can be added by added via the left menu or edited by right-clicking on the Module. Each Module could have an input Module and output Module that can be added with the left mouse button. Based on Module types, the functionality is defined. Further, the editor has the validation rules for the Pipelines and Modules creation.

Unfortunately, the documentation for the individual modules is not available; though, for test scenarios, you will suffice with the following modules:

- **BindWithConstant** - Binds data to variable
- **ApplyConstruct** - Execute SPARQL query
- **ReturnRDF** - Return data in desired format

The Pipeline contains functions that can be run. The function call is called Execution. The Executions page shows information about Execution, such as start time, the name of the called function, etc. Execution can be opened, and the inputs and outputs of individual modules can be checked. Further, the user can modify the module properties. It is possible to debug modules using this procedure.

S-Pipes UI Scripts Executions

Nodes count: 4

Add module

Add module

Function call

Call function

Graph render strategy

TopBottom

Execution report

Manage script's ontology

Validate Report

Variables info

_pid: execute-greeting
 firstName: Robert
 lastName: Plant
 personName: Robert Plant

hello-world3.sms.ttl

Bind person name

Bind person id

Construct greeting

Return greeting statement

Figure 8.2: Hello-world3.sms.ttl script execution detail

■ Test scenarios

The scenario is designed as an introductory task for the beginning user. Participant should be able to execute the function and check the execution output.

TC1 Hello world scenario

- 1 Open *hello-world3.sms.ttl* script
- 2 Change render strategy

- 3 Call the function `execute-greeting`
- 4 Fill up the `firstName` and `lastName` and submit form
- 5 Select your execution
- 6 Check if you input variables are `firstName="YOUR-VALUE"` and `lastName="YOUR-VALUE"`
- 7 Download the input and output of `Return greeting statement` module
- 8 Check if the output of the execution is in *JSON-LD* format and the output message is **Hello Your Name**

The scenario shows validation and work with modules.

TC2 Validation scenario

- 1 Open `hello-world2.sms.ttl` script
- 2 Collapse `hello-world2.sms.ttl`
- 3 Open validation report
- 4 Select problematic modules and fix them
- 5 Delete module with `Empty-bind-constant` label
- 6 Add new module of type `Bind` with `constant` with `label`, `outputVariable` and `value`
- 7 Add edge from your new module to `Construct greeting`
- 8 Delete module with `Bind person name`
- 9 Update `Construct greeting` to consume variable from new module
- 10 Call the function `execute-greeting`
- 11 Download the input and output of `Return greeting statement` module

The scenario shows the possibility of debugging.

TC3 Debug scenario

- 1 Open `hello-world.sms.ttl` script
- 2 Execute the `execute-greeting`
- 3 Check the execution report
- 4 Fix the `Construct greeting` module to return you *Hello VARIABLE*
- 5 Execute the `execute-greeting`

■ Post scenario questions

- Which steps in the test scenarios were not apparent what you were looking for the longest?
 - P0** - The question for **TC1** in steps 4 and 5 was asked quite incorrectly because I first called the execution and only then entered the parameters and started.
 - P1** - I was puzzled in scenarios 2 and 3 and had to ask for help in both. In fact, **TC3** seemed impossible to me because I don't know the required programming language, and I don't know the terms as a variable.
 - P2** - I found the **TC3** scenario very difficult, as it was necessary to work with a module that required knowledge of the SPARQL language, which I have only seen; however, I never work with it.
 - P3** - I was a little confused when working with modules. The knowledge or SPARQL was required at **TC3**, but the solution was the same as in **TC1**, so it was a simple task.

- What do you think about the application? You can rate it from 0 to 10, where the 10 is top grade.
 - P0** - With good documentation, the app could be used. The rating of the app is 6 out of 10.
 - P1** - I did not enjoy the testing process, so I abstain.
 - P2** - I guess the application is focused on SPipes scripts developers. As I am not a script developer, it is hard to rate it; thus, I give it 5.
 - P3** - I really enjoy it - 9/10!

- Do you have any real case scenario for the application?
 - P0** - With this application, I see a fundamental problem with the need to know the basic programming concepts, including knowledge of the SPARQL language. Another tricky part is the individual modules, which require knowledge. So I can't think of any scenario. I do not want to go into intense detail; however, the RDF technologies suffer from a lack of popularity and significant divergence from popular technologies.
 - P1** - Honestly no. I guess the application requires programming and domain knowledge.
 - P2** - If I understand the application correctly, it is actually the ETL editor, where the individual modules define the transformation. It seems interesting to me, but the actual usage would be pretty challenging. Using the tool would be relatively complex for an inexperienced user, and for the programmer, it is a minimal tool.
 - P3** - I hope, if some training with the application expert will be provided to users, the application could be beneficial.

■ 8.1.4 User testing evaluation

This section describes the severity of the problems found. It also tells how long it took the participants to go through the scenarios and finally discuss the discovered issues.

Severity of the findings

- High - The problem severity is critical and significantly limits the usage of the application. The finding should be fixed immediately.
- Medium - The problem limits the usage of the application; however, it is still possible to fulfill the task. The finding should be fixed as soon as possible.
- Low - The problem does not limit the application usage. It is typically the visualization problem. The finding should be fixed later.

■ Duration

The duration is measured from the introduction of the application. It resulted in answering the questionnaire, which was not included in the total time. The average time was 58m. It is important to comment on the remarks on P0 and P3. P0 was a bit biased because the participant saw the application before, and the difficulty of the scenarios was consulted with him. P3 had problems because the participant forgot how to use Docker, but he wanted to try the installation of the application. It was successful; however, the Docker itself added 45m to the duration. The participant still needed to figure out the configuration for the Docker itself and the necessary variables. The configuration of the Docker is shown in the duration column in brackets. The final duration of every participant is shown in Table 8.3.

Participant	Duration
P0	30m
P1	84m
P2	56m
P3	53m+(45m)

Table 8.3: Scenarios duration for every participant

■ Findings

F1 Multiple modal windows opened at once.

- **Severity** - High
- **Found by participant** - P0
- **Found in test scenario** - TC1

- **Description** - If the participant opened the modal window and wanted to close it and open another, the previous opened still appear. It has to fix immediately, and it was no longer issue for other participants.

F2 Spelling problem execute-greeding

- **Severity** - Low
- **Found by participant** - P0
- **Found in test scenario** - TC3
- **Description** - The label of the module had a spelling problem; instead of *execute-greeding* it should be *execute-greeting*.

F3 The bitcoin icon is used for module type of `BindWithConstant`.

- **Severity** - Low
- **Found by participant** - P0, P1, P2, P3
- **Found in test scenario** - TC1
- **Description** - The icons were chosen randomly from Font Awesome² library. The icon will be changed after testing.

F4 The execution of the function is not convenient.

- **Severity** - Medium
- **Found by participant** - P0
- **Found in test scenario** - TC1
- **Description** - The parameters for the execution were not suggested and had to fill as a text value. It was fixed immediately after PO testing for more friendly usage of the editor.

F5 Function form submit button text.

- **Severity** - Low
- **Found by participant** - P0, P3
- **Found in test scenario** - All
- **Description** - If the user wants to execute a function, the submit keyword is Debug instead of Submit or Execute.

F6 Order of the list of executions.

- **Severity** - Low
- **Found by participant** - P0, P2, P3
- **Found in test scenario** - All
- **Description** - The last executed execution is at the end of the list. The list of the execution should start with the newest one.

²<https://fontawesome.com/>

F7 Only the part of the execution input/output is downloaded.

- **Severity** - Medium
- **Found by participant** - P0
- **Found in test scenario** - TC1
- **Description** - When the user downloaded, the execution input or output of the prefixes was not downloaded. It was fixed immediately after PO testing for more friendly usage of the editor.

F8 Cytoscape BirdEye overlap modal window.

- **Severity** - Low
- **Found by participant** - P0, P2
- **Found in test scenario** - All
- **Description** - The CSS *z-index*³ property of BirdEye component is not correctly setup.

F9 Adding edge from module to file group is possible.

- **Severity** - Low
- **Found by participant** - P0, P2
- **Found in test scenario** - TC3
- **Description** - Users can connect the module with the module group(the file name), and the edge is rendered. This action has only a visualization effect, and it does not affect the script; however, it should be forbidden.

F10 Bad execution name is rendered.

- **Severity** - Medium
- **Found by participant** - P2, P3
- **Found in test scenario** - TC3
- **Description** - Bad execution name if more than five execution is in the list of executions.

F11 Execution of corrupted script.

- **Severity** - Medium
- **Found by participant** - P1
- **Found in test scenario** - TC2
- **Description** - SPipes engine returns a Server error(500), when the user executes the corrupted script. The participant created a script, which did not make sense, and ran it. The SPipes engine returned a Server error, and it does write enough data about the execution.

³https://www.w3schools.com/cssref/pr_pos_z-index.asp

F12 Prefix of new added module

- **Severity** - Low
- **Found by participant** - P0, P2, P3
- **Found in test scenario** - TC2
- **Description** - The editor suggests the URI of the new module with prefix `http://example.com/change_`; however, a more convenient prefix would be the script prefix.

■ Summary

All participants were able to complete the test scenario; however, P1 needed quite a lot of assistance. It should be noted that none of the participants were in the role of script developer (defined in subsection 5.2.4), which requires knowledge of SPipes module types and SPARQL language. From the responses to the test scenario, it is evident that the participants had the most difficulty with the **TC3** scenario. In addition, participants had difficulties with the SPARQL language. The High severity problems were removed immediately after P0 and the Medium and Low severity findings will be removed in the future.

■ 8.2 Comparison with original SPipes Editor

The new editor re-implements the original editor, trying to keep the original functionality and add a new one. The main goal of the re-implementation was to simplify development for developers, add tests, and fix bugs. The new functionality mainly focuses on advanced features to manage SPipes scripts. This section compares the implementation and the characteristics of new and original editors.

■ 8.2.1 Implementation Comparison

Based on the analysis in chapter 3, we decided that the project will no longer be developed in the Scala language, as it did not bring any benefits. Another important change was replacing the Spring framework with Spring Boot, which will allow easier development. Furthermore, for the sake of sustainability, it is necessary to implement tests in the application. The analysis shows that a significant part of the declared functionality does not work and it is highly challenging to fix it without the tests. Another problem presents the used graphics library, which is no longer developed and does not support the new functionality. Based on the chapter 4 analysis, the Cytoscape.js library is selected, which has the necessary features.

Application development is also truly problematic. Both editors depend on other services (SPipes engine and RDF4J) that they need to be configured. This topic is not handled in the original editor, but the new editor introduces

dockerization of the project. Dockerization will help with the launch and proper configuration of all services and will simplify future deployment.

The last significant change is the separation of the front-end part of the application into a separate project to decrease the coupling of the application. The table with the implementation comparison is shown in Table 8.4.

	Original Editor	New Editor
Programming language	Scala	Java
Platform	Web	Web
Application framework	Spring	Spring boot
Visualization library	The Graph Editor	Cytoscape.js
Front-end framework	React	React
Tests	No	Yes
Dockerized	No	Yes
Front-end as standalone application	No	Yes

Table 8.4: Implementation comparison between original and new SPipes Editor

8.2.2 Features Comparison

The new editor first tries to fix bugs in the original application and focuses on the new functionality. It re-implements all the features of the original editor, except support for display on a mobile device, but it has been marked as non-essential. The main problem with the original editor was not working, adding and editing modules, which are vital to the operation of the application. This functionality has been fixed and re-implemented.

The next step was to add a script function call, which would enable debugging. Due to this requirement, Docker was introduced; thus, the integration of all services would be facilitated. Execution can then be traversed and debugged with the help of repeated start-ups or mocking the module input. Unfortunately, due to lack of time, the run part of the pipeline for indirect execution of the module was not implemented, which would further facilitate the debugging.

In addition to the original editor, SHACL validation has been added to help prevent errors and introduce the best practices for writing the scripts.

Thanks to Cytoscape, better visualization adjustments could be made, such as collapsing, which supports better orientation in larger graphs. The framework also allows for better navigation and scripting and will enable you to label corrupt modules.

The table with the full comparison is shown in Table 8.5

Functionality	Original Editor	New Editor
Script creation	No	Yes
Script visualization	Yes	Yes
Modules grouping and collapsing	No	Yes
Multiple layout algorithms	Yes	Yes
External change notification	Yes	Yes
Add new module	Partly	Yes
Module parameter value adjustment	Partly	Yes
Script execution	No	Yes
Script debugging	No	Yes
Show the graph overview	Yes	Yes
Collapsing based on module membership	No	Yes
Collapsing respects hierarchy of files	No	Designed
List execution history	No	Yes
Query execution history	No	Yes
Multiple user editing	Yes	Yes
Mobile visualization	Yes	No
Tablet visualization	Yes	Yes
Manage script imports	No	Yes
Transfer module to related script	No	Yes
General transfer of module	No	Designed
Module execution	No	Yes
Module debug	No	Yes
Indirect module debug	No	Designed
Script and module validation	No	Yes
Execution report	No	Yes
Module execution parameters visualizaiton	No	Yes

Table 8.5: Original and new SPipes editor functions comparison

Chapter 9

Conclusion

9.1 Summary

This thesis aimed to provide a new implementation of the editor for SPipes scripts or extend the original editor. Based on the detailed analysis, we decided for re-implementation. The benefits of the re-implementation are manifold. Let us name the major ones, that can be considered as an original contribution to the topic by the author. It needs to be highlighted that all the objectives that were set out in the thesis guideline have been fulfilled completely.

First, the backend part was rewritten from Scala to Java based on a detailed analysis of related problems. Rewrite eliminated issues stemming from intricate compatibility issues between Scala and Spring framework in the original editor.

Great deal of effort was put into testing of the application and providing tests for individual components of the application. This aspect was largely neglected previously, which was intertwined with hidden bugs in the original implementation. Apart from the revelation of such errors, it also guards against unintentional changes in the behavior of the application during the future development. Further, the deployment process and rather complex configuration of all the parts of the editor was simplified significantly with introduction of Docker and definition of components and their communication via a docker-compose configuration file.

In terms of new provided functionality, the possibility to validate and debug scripts and modules were added as well allow to create a comprehensive view on large and complicated scripts. Validation of the scripts and modules, as implemented in this work, brings a striking improvement of the work with semantic data in the SPipes language.

For future development of the editor, the focus could be put on aspects that would further make adoption of its use easier, not only for the end users but also for developers. Thus, the focus could be put on extension of the list of best-practices for module/scripts validations, proper documentation of every module type and its constraints. From the developers view, it could be beneficial to implement tests for the frontend part that would give the developer larger confidence that changes made due to the addition of new

9. Conclusion

functionality would not break existing behavior of the editor.

Appendix A

Code Attachments

```
version: '3.7'

services:
  s-pipes-editor-ui:
    image: 'chlupnoha/s-pipes-editor-ui:latest'
    ports:
      - '3000:80'
    networks:
      - overlay
    depends_on:
      - s-pipes-editor-rest
    environment:
      SERVICE_URL: "s-pipes-editor-rest:18115"

  s-pipes-editor-rest:
    image: 'chlupnoha/s-pipes-editor-rest:latest'
    container_name: s-pipes-editor-rest
    ports:
      - '18115:18115'
    expose:
      - "18115"
    networks:
      - overlay
    depends_on:
      - s-pipes-engine
      - rdf4j
    environment:
      - SCRIPTPATHS=####
      - ENGINEURL=http://s-pipes-engine:8080/s-pipes/
      - SCRIPTRULES=####
      - RDF4J_REPOSITORYURL=http://rdf4j:8080/rdf4j-server/rep
      - RDF4J_REPOSITORYNAME=s-pipes-hello-world
      - RDF4J_PCONFIGURL=####
    volumes:
```

```
- /tmp:/tmp
- /home:/home
- /usr/local/tomcat/temp/:/usr/local/tomcat/temp/

s-pipes-engine:
  image: 'chlupnoha/spipes-engine:latest'
  container_name: s-pipes-engine
  ports:
    - "8081:8080"
  expose:
    - "8081"
  networks:
    - overlay
  depends_on:
    - rdf4j
  environment:
    - CONTEXTS_SCRIPTPATHS=####
  volumes:
    - /tmp:/tmp
    - /home:/home
    - /usr/local/tomcat/temp/:/usr/local/tomcat/temp/

rdf4j:
  image: 'eclipse/rdf4j-workbench:amd64-3.5.0'
  container_name: rdf4j
  ports:
    - "8080:8080"
  expose:
    - "8080"
  networks:
    - overlay
  environment:
    - JAVA_OPTS=-Xms1g -Xmx4g
  volumes:
    - data:/var/rdf4j
    - logs:/usr/local/tomcat/logs

volumes:
  data:
  logs:

networks:
  overlay:
```

Listing 19: SPipes editor Docker compose

Appendix B

Installation guide

The recommended installation of the SPipes editor is via the Docker.

B.1 Installation via Docker

1. Install Docker <https://docs.docker.com/get-docker/>.
2. Clone project `git clone https://github.com/chlupnoha/s-pipes-editor-ui` for newest version or `s-pipes-editor-ui` from the attachment. To obtain the testing data it is recommended to clone the `git clone https://github.com/chlupnoha/s-pipes-editor` or use `s-pipes-editor` from the attachment.
3. Open `docker-compose.yml` and adjust the parameters to respect your configuration. The explanation of the parameters is described in `README.md`. The sample data are part of the project `s-pipes-editor/src/test/resources/scripts_test/sample` so you can use this data. The important variables are the following ones:
 - a. `SCRIPTPATHS` - Location of the scripts for *s-pipes-editor-rest*.
 - b. `SCRIPTRULES` - Location of the rules for *s-pipes-editor-rest*.
 - c. `RDF4J_PCONFIGURL` - Configuration of RDF4J for *spipes-engine*.
 - d. `CONTEXTS_SCRIPTPATHS` - Location of the scripts for *spipes-engine*.
4. Execute `docker-compose up`.
5. Unfortunately, JOPA requires a repository at the application startup; thus, you have to open `http://localhost:8080/rd4j-workbench` and create a new repository, which is the same as `RDF4J_REPOSITORYNAME`. The default one is `s-pipes-hello-world`.
6. Kill `docker-compose` process and execute `docker-compose up` again.
7. Open `http://localhost:3000/`.

Appendix C

Attachment

```
/
├── thesis.pdf ..... this thesis
├── thesis ..... the source files of this thesis in LATEX
│   ├── img ..... figures
│   ├── tex ..... content files
│   ├── code ..... code attachments
│   └── zav_prace.pdf ..... thesis assignment
├── app ..... Application files
│   ├── s-pipes-editor ..... SPipes editor
│   └── s-pipes-editor-ui ..... SPipes editor UI
```


Appendix D

Bibliography

- [1] Angus Addlesee. Understanding Linked Data Formats. <https://medium.com/wallscope/understanding-linked-data-formats>. (Accessed on 2021/06/08).
- [2] Alvin Alexander. *Scala cookbook*. O'Reilly, Sebastopol, CA, 2013.
- [3] Francesco Antoniazzi and Fabio Viola. Rdf graph visualization tools: A survey. In *Proceedings of the 23rd Conference of Open Innovations Association FRUCT*, FRUCT'23, Helsinki, Uusimaa, FIN, 2018. FRUCT Oy.
- [4] Apache Jena - Apache Jena SHACL. <https://jena.apache.org/documentation/shacl/index.html>. (Accessed on 2021/06/20).
- [5] ArangoDB, the multi-model database for graph and beyond. <https://www.arangodb.com/>. (Accessed on 2021/06/11).
- [6] Ken Arnold, James Gosling, and David Holmes. *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.
- [7] Howard Austerlitz. Computer programming languages. In *Data Acquisition Techniques Using PCs*, pages 326–360. Elsevier, 2003.
- [8] Baeldung. A comparison between spring and spring boot. <https://www.baeldung.com/spring-vs-spring-boot>, journal=Baeldung. (Accessed on 2021/07/15).
- [9] Baeldung. Custom Error Message Handling for REST API | Baeldung. <https://www.baeldung.com/global-error-handler-in-a-spring-rest-api>. (Accessed on 2021/06/09).
- [10] Baeldung. Integration testing in spring. <https://www.baeldung.com/integration-testing-in-spring>. (Accessed on 2021/07/15).
- [11] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. 2009.

- [12] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Commun. ACM*, 37(8):76–82, August 1994.
- [13] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5:1–22, 01 2009.
- [14] Nikolaos Bourbakis. *Artificial intelligence and automation*. World Scientific, Singapore River Edge, NJ, 1998.
- [15] Richard Brath. *Graph analysis and visualization : discovering business opportunity in linked data*. John Wiley & Sons, Indianapolis, IN, 2015.
- [16] Kevin Brennan. *A guide to the Business analysis body of knowledge (BABOK guide)*. International Institute of Business Analysis, Toronto, 2009.
- [17] A Brief History of Scala. <https://www.artima.com/weblogs/viewpost.jsp?thread=163733>. (Accessed on 2021/06/09).
- [18] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall Press, USA, 3rd edition, 2009.
- [19] Diego Valerio Camarda, Silvia Mazzini, and Alessandro Antonuccio. Lodlive, exploring the web of data. In *Proceedings of the 8th International Conference on Semantic Systems, I-SEMANTICS '12*, page 197–200, New York, NY, USA, 2012. Association for Computing Machinery.
- [20] Charalampos C. Charalampidis and Euclid A. Keramopoulos. Semantic web user interfaces – a model and a review. *Data & Knowledge Engineering*, 115:214–227, May 2018.
- [21] Helen Cook, Nadezhda Doncheva, Damian Szklarczyk, Christian von Mering, and Lars Jensen. Viruses.string: A virus-host protein-protein interaction database. *Viruses*, 10:519, 09 2018.
- [22] Mahboubeh Dadkhah, Saeed Araban, and Samad Paydar. A systematic literature review on semantic web enabled software testing. *Journal of Systems and Software*, 162:110485, April 2020.
- [23] Eclipse RDF4J developers. Welcome · Eclipse RDF4J™ | The Eclipse Foundation. <https://rdf4j.org/>. (Accessed on 2021/06/11).
- [24] Best Graph Database - Native GraphQL Database. <https://www.dgraph.io/>. (Accessed on 2021/06/11).
- [25] dotnetrdf/dotnetrdf. <https://github.com/dotnetrdf/dotnetrdf>, June 2021. (Accessed on 2021/06/20).

- [26] Martin Fowler. *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley, Boston, 2004.
- [27] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, page btv557, September 2015.
- [28] Jeff Friesen. Are checked exceptions good or bad? <https://www.infoworld.com/article/3142626/are-checked-exceptions-good-or-bad.html>, November 2016. (Accessed on 2021/06/09).
- [29] Ian Gorton. *Essential software architecture*. Springer, Berlin Heidelberg New York, 2011.
- [30] Graphlytic - Graph Analytics And Visualization Software. <https://graphlytic.biz/>. (Accessed on 2021/06/13).
- [31] GraphQL | A query language for your API. <https://graphql.org/>. (Accessed on 2021/06/11).
- [32] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
- [33] Tuukka Hastrup, Richard Cyganiak, and Uldis Boj. Browsing linked data with fenfire. 369, 01 2008.
- [34] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. Relfinder: Revealing relationships in rdf knowledge bases. In *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies (SAMT 2009)*, pages 182–187, Berlin/Heidelberg, 2009. Springer.
- [35] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156, 2015.
- [36] Matthew Horridge. OWL Syntaxes. <http://ontogenesis.knowledgeblog.org/88>, January 2010. (Accessed on 2021/06/07).
- [37] Dorota Huizinga. *Automated defect prevention : best practices in software management*. Wiley-Interscience IEEE Computer Society, Hoboken, N.J, 2007.
- [38] Chapter 11. Interoperability between Scala and Java · Scala in Action. <https://livebook.manning.com/scala-in-action/chapter-11>. (Accessed on 2021/06/09).
- [39] Introduction scala. <https://docs.scala-lang.org/tour/tour-of-scala.html>. (Accessed on 2021/06/09).

- Eero Hyvönen, Eva Blomqvist, Valentina Presutti, Guilin Qi, Uli Sattler, Ying Ding, and Chiara Ghidini, editors, *Knowledge Engineering and Knowledge Management*, pages 154–158, Cham, 2015. Springer International Publishing.
- [55] Christos Loverdos and Apostolos Syropoulos. *Steps in Scala: An Introduction to Object-Functional Programming*. Cambridge University Press, USA, 2010.
- [56] Eva a Zdeněk Macháčkovi. Metoda MoSCoW a model KANO. <https://www.systemonline.cz/rizeni-projektu/metoda-moscow-a-model-kano.htm>. (Accessed on 2021/07/02).
- [57] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008.
- [58] Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, 2015.
- [59] THE MVC-WEB DESIGN PATTERN. In *Proceedings of the 7th International Conference on Web Information Systems and Technologies*. SciTePress - Science and and Technology Publications, 2011.
- [60] 17.Web MVC framework. <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html>. (Accessed on 2021/06/11).
- [61] nishanil. Co je Docker? <https://docs.microsoft.com/cs-cz/dotnet/architecture/containerized-lifecycle/what-is-docker>. (Accessed on 2021/07/15).
- [62] OWL Web Ontology Language Guide. <https://www.w3.org/TR/owl-guide/>. (Accessed on 2021/06/08).
- [63] James Powell and Matthew Hopkins. Ontologies. In *A Librarian's Guide to Graphs, Data and the Semantic Web*, pages 31–43. Elsevier, 2015.
- [64] 4IZ440 - Propojená data na webu (Vojtěch Svátek). <https://nb.vse.cz/~svatek/rzzw.html>. (Accessed on 2021/06/07).
- [65] RDF 1.1 Concepts and Abstract Syntax. <https://www.w3.org/TR/rdf11-concepts/>. (Accessed on 2021/06/07).
- [66] RDF 1.1 Primer. <https://www.w3.org/TR/rdf11-primer/>. (Accessed on 2021/06/07).
- [67] RDF Primer. <https://www.w3.org/TR/rdf-primer/#rdfschema>. (Accessed on 2021/06/07).
- [68] RDF Schema 1.1. <https://www.w3.org/TR/rdf-schema/>. (Accessed on 2021/06/07).

- [69] React – A JavaScript library for building user interfaces. <https://reactjs.org/>. (Accessed on 2021/06/11).
- [70] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008(173), September 2008.
- [71] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O’Reilly Media, Inc., 2013.
- [72] Jan Rylich. Technologie sémantického webu. <https://ikaros.cz/technologie-semantickeho-webu>, October 2011. (Accessed on 2021/06/07).
- [73] s-pipes-modules.git - Gitblit. <https://kbss.felk.cvut.cz/gitblit/summary/s-pipes-modules.git>. (Accessed on 2021/06/11).
- [74] Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3). <https://aws.amazon.com/s3/>. (Accessed on 2021/07/13).
- [75] SHACL. <https://book.validatingrdf.com/bookHtml011.html>. (Accessed on 2021/06/20).
- [76] weso/shaclex. <https://github.com/weso/shaclex>, June 2021. (Accessed on 2021/06/20).
- [77] SHACL Test Suite and Implementation Report. <https://w3c.github.io/data-shapes/data-shapes-test-suite/>. (Accessed on 2021/06/20).
- [78] P. Shannon. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13(11):2498–2504, November 2003.
- [79] Shapes Constraint Language (SHACL). <https://www.w3.org/TR/shacl/>. (Accessed on 2021/06/20).
- [80] Sharp, Austin et al. Janusgraph. <https://janusgraph.org/>. (Accessed on 2021/06/16).
- [81] Mgr. Linda Skolková. Sémantický web – jak dál? <https://ikaros.cz/semanticky-web-\T1\textendash-jak-dal>, Jun 2009. (Accessed on 2021/06/07).
- [82] SPARQL Query Language for RDF. <https://www.w3.org/TR/rdf-sparql-query/>. (Accessed on 2021/06/08).
- [83] SPARQLMotion. <https://sparqlmotion.org/>. (Accessed on 2021/06/08).

- [84] Basic Spring web application in Java, Kotlin and Scala - comparison . <https://rskupnik.github.io/basic-spring-webapp-java-kotlin-scala>. (Accessed on 2021/06/09).
- [85] Spring Boot Architecture - javatpoint. <https://www.javatpoint.com/spring-boot-architecture>. (Accessed on 2021/07/20).
- [86] Vojtěch Svátek. Ontologie a www. In *Sborník konference Datakon*, pages 27–55, 2002.
- [87] *Systems engineering fundamentals*. US Army Department of Defense, Washington D.C, 2001.
- [88] TopBraid SPARQLMotion Library. <https://www.topquadrant.com/sparqlmotion/lib.html#sml:ApplyConstruct>. (Accessed on 2021/06/11).
- [89] TopQuadrant. Sparqlmotion. <https://sparqlmotion.org/images/Example-SM-Flow.png>. (Accessed on 2021/07/20).
- [90] Topbraid composer - maestro edition. <https://www.topquadrant.com/products/topbraid-composer/>. (Accessed on 2021/07/20).
- [91] TopQuadrant/shacl. <https://github.com/TopQuadrant/shacl>, June 2021. (Accessed on 2021/06/20).
- [92] SPARQLMotion | TopQuadrant, Inc. <https://www.topquadrant.com/technology/sparqlmotion/>. (Accessed on 2021/06/11).
- [93] RDF 1.1 Turtle. <https://www.w3.org/TR/turtle/>. (Accessed on 2021/06/08).
- [94] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., USA, 1996.
- [95] Akash Verma. Journey from React 15 to React 16. <https://akash-mihul.medium.com/journey-from-react-15-to-react-16-56e336092379>, March 2018. (Accessed on 2021/07/13).
- [96] W3C. Layer cake. <https://www.w3.org/2007/03/layerCake.png>. (Accessed on 2021/07/20).
- [97] W3C. Rdf example graph. <https://www.w3.org/TR/rdf11-primer/example-graph.jpg>. (Accessed on 2021/07/20).
- [98] Craig Walls. *Spring Boot in Action*. Manning Publications Co., USA, 1st edition, 2016.

- [99] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA, 2012. Association for Computing Machinery.
- [100] Websocket. <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>. (Accessed on 2021/07/20).
- [101] We Love Graphs: JavaScript Graph Drawing Libraries. <https://anvaka.github.io/graph-drawing-libraries/#!/all>. (Accessed on 2021/06/14).
- [102] Why Spring? <https://spring.io/why-spring>. (Accessed on 2021/07/15).
- [103] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles — visualization and automatic layout of graphs. In *Graph Drawing Software*, pages 173–191. Springer Berlin Heidelberg, 2004.
- [104] Doroshenko Yan. Editor sémantických datových proudů. June 2018.