Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of computer

Debugging API for semantic pipelines

Miron Grishchenko

Supervisor: Mgr. Miroslav Blaško, Ph.D.
Field of study: Software Engineering and Technology
May 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Grishchenko** Jméno: **Miron** Osobní číslo: **480526**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Rozhraní pro ladění sémantických datových proudů

Název bakalářské práce anglicky:

Debugging API for semantic pipelines

Pokyny pro vypracování:

SPipes (Semantic data pipelines) [1] is 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.
The goal of this work is to reimplement or extend the functionality of existing debugging REST API of SPipes engine [3, 4]. The API should be discoverable [5] and able to query execution history, analyze performance of modules and pipelines, and find candidate modules to cache.
Instructions:
1) become familiar with Semantic Web technologies (OWL, RDF, JSON-LD, SPARQL, RDF4J)
2) review existing debugging capabilities of SPipes and similar tools
3) analyze requirements for the debugging API and define scenarios to use it
4) design the API and implement its prototype
5) test implemented prototype and validate it on selected scenarios with at least three users

Seznam doporučené literatury:

[1] Blaško, Miroslav, SPipes (online at https://github.com/kbss-cvut/s-pipes)
[2] TopQuadrant, Inc. "SPARQL motion" (online at http://sparqlmotion.org)
[3] Petr Jordán, Debugging scripts in SPipes editor, 2021. (https://dspace.cvut.cz/handle/10467/97070)
[4] 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.
[5] Fielding, Roy T. "REST APIs must be hypertext-driven." Untangled musings of Roy T. Fielding (2008): 24.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Mgr. Miroslav Blaško, Ph.D. skupina znalostních softwarových systémů

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

Datum zadání bakalářské práce: **30.01.2023** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **22.09.2024**

Mgr. Miroslav Blaško, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

I would like to thank my supervisor, Mgr. Miroslav Blaško, Ph.D., for his patient guidance, encouragement, and advice during my studies and while writing my bachelor thesis. Completing this work would have been all the more difficult had it not been for the support provided by friends, who are also members of the CTU Prague student community, including Roman Stepa, Bc. Andrey Bortnikov, Daniil Simon and Yevhen Chaban. I must also express my gratitude to my family for their continued support and encouragement.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by Act No. 121/2000 Coll., the Copyright Act, as amended, in particular, that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague, May 15, 2023

Abstract

This thesis examines the need for debugging tools in SPipes, a language for managing semantic pipelines. Exploring the current debugging capabilities of SPipes and similar tools, the research designs possible debugging scenarios. By analyzing scenarios and implementing the SPipes debugging API, this work successfully adds multiple debugging tools to facilitate error identification and optimization of SPipes scripts. The thesis concludes with recommendations for further development of SPipes debugging API.

Keywords: SPipes, Debugging tools, Debugging API

Supervisor: Mgr. Miroslav Blaško, Ph.D.

Abstrakt

Tato práce zkoumá potřebu ladicích nástrojů v SPipes, jazyce pro správu sémantických datových proudů. Zkoumáním současných možností ladění SPipes a podobných nástrojů, výzkum navrhuje možné scénáře ladění. Analýzou scénářů a implementací SPipes debugging API tato práce úspěšně přidává několik nástrojů pro ladění, které usnadňují identifikaci chyb a optimalizaci skriptů SPipes. V závěru práce jsou uvedena doporučení pro další vývoj SPipes debugging API.

Klíčová slova: SPipes, Nástroje pro ladění, Rozhraní pro ladění

Překlad názvu: Rozhraní pro ladění sémantických datových proudů

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Foreword

Semantic Data, also known as Linked Data, is a powerful tool for organizing and integrating data from different sources. They are based on open standards and protocols, such as RDF and SPARQL, and allow the creation of a global network of linked data. Using Linked Data, it is possible to integrate data from different sources, improving the accessibility and understanding of information.

One language inspired by the Linked Data concept is SPipes, a language for managing semantic pipelines defined in RDF. SPipes provides the ability to create and execute scripts based on the SPARQLMotion programming language to process and analyze semantic data. Unfortunately, SPipes does not have enough tools for debugging. This can make it difficult to detect and fix errors in scripts written in SPipes.

1.2 Goals and motivation

The purpose of this thesis is to create tools for debugging scripts written in SPipes language. These tools will be designed to accelerate the work of developers who create and maintain SPipes-based pipelines. They will be designed to facilitate the process of debugging, identifying, and fixing bugs, and improving developer productivity when creating and maintaining SPipes-based scripts.

The main motivations and goals of this thesis include:

- Speeding up the debugging process: Developing tools to help developers efficiently find and fix errors in SPipes scripts. This will reduce the time spent on debugging and improve the quality of developed pipelines.
- Allow users to better optimize SPipes scripts: Creating tools that will allow you to get statistical information about the operations performed will allow user to see clearly which parts of the script should be optimized, for faster operation or for less memory consumption.

In addition, the work will not involve creating a user interface, but instead, a discoverable API using HATEOAS principles, by which the user can navigate between endpoints using the links provided directly in the response with a specific object.

Chapter 2

Background

This chapter describes technologies that are related to SPipes, a tool for processing of semantic pipelines written in RDF language. Here will be described such technologies and concepts as Semantic Web, RDF, RDFS, OWL, SPARQL, and SPARQL Motion.

2.1 Semantic Web

The Semantic Web, also known as WEB 3.0, is an extension of the existing "Web of documents" [1].

In the Semantic Web data are connected to each other not only on one current website, but it is linked through the entire web. However, to be able to use all the power of linked information, it's needed to have a huge amount of data on WEB available in a standard format so that Semantic Web tools can access and manage this data [2].

The main goal of the Web of Data is to create a web of machine-understandable information. With linked data, when a person or machine has some part of this data, it's possible to find other data, somehow related to the piece of information, that was given in the beginning [3].

2.2 Linked Data

Linked Data is a structured data graph, which allows information interlinking across independent servers. Linked Data helps machines and people to access data across servers and gives a better understanding of data meaning.

In 2006, Tim Berners-Lee formulated four basic principles of Linked Data [3]:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI (Uniform Resource Identifier), provide useful information, using the standards (RDF, SPARQL). RDF and SPARQL will be described in the following sections.

- Include links to other URIs. so that they can discover more things.

LOD - Linked Open Data is a project that differs from Linked Data with free access for everyone. The aim of the project is to identify datasets that are available under open licenses, re-publish these in RDF on the Web and interlink them with each other [4]. Nowadays there are thousands of datasets published among this project in different fields of knowledge, such as geography, government, media, publications, and others. And the most powerful thing about it is that all these data are connected and information leads to other information. In the picture 2.1 is a graph of datasets from lod-cloud.net. According to lod-cloud.net, on 20 May 2020, they had 1,255 registered datasets [5]. Each dataset contains from 1 and up to 1.5 billion triples (simple expressions about some object), as DBpedia has.

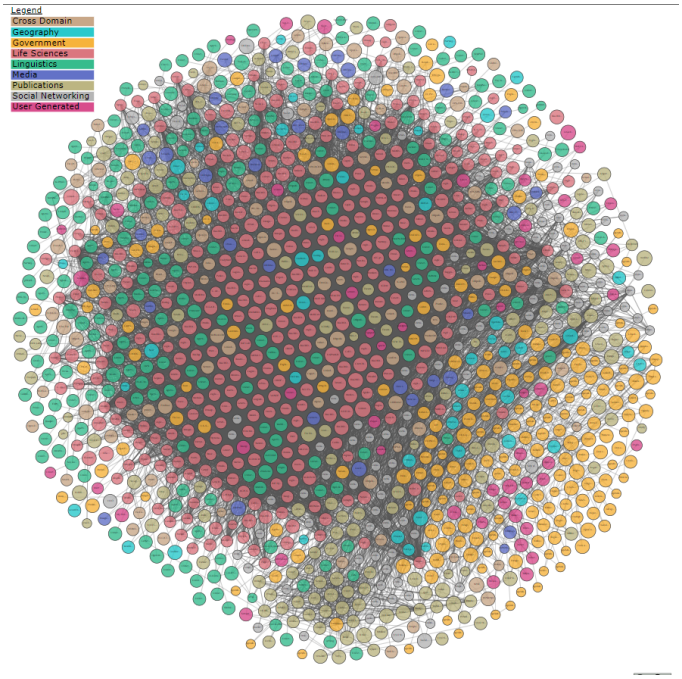

Figure 2.1: lod-cloud.net LOD graph [5]

2.3 Ontology

Ontology in computer science is an attempt at a comprehensive and detailed formalization of some domain of knowledge by means of a conceptual schema. Ontology is a collection of Entity Type and Entity Relationship Type definitions associated with a realm of discourse. Ontologies are used in programming as a form of representation of knowledge about the real world or part of it. Ontologies are built on much the same principle. Ontologies are usually represented in languages that allow them to move away from strict data structures and implementation strategies. In practice, ontology languages are closer in expressive power to first-order logic than languages used to model databases [6].

2.4 Ontology components

Amount of ontology components differ from source to source, but these are the most common: **Classes**, **Individuals**, **Properties**, **Logical expressions and rules** and **Annotations**.

Components of ontology usually include the following elements [7] [8]:

- Classes - Classes represent entities, objects, or concepts in the subject area covered by the ontology. Classes can have a hierarchical structure and include subclasses and superclasses.
- Individuals - Individuals represent specific instances of classes in ontology. Individuals can be individual objects, events, places, and other domain-specific entities.
- Properties - Properties define attributes, characteristics, or relationships between classes, individuals, or other entities in the ontology. Properties can be of two types: object properties (leading to other instances) (such as "has an owner") and attribute properties (such as "has age" or "has weight").
- Logical expressions and rules - Logical expressions and rules define logical relationships, constraints, and rules in the ontology. They may include constraints on property values, relations between classes, logical operators, and other logical expressions that define the rules for inference and reasoning in the ontology.
- Annotations - Annotations are metadata or additional information related to classes, individuals, properties, or relationships in the ontology. They may contain descriptions, comments, keywords, authorship, and other additional information that may be useful for understanding

2.5 RDF

The Resource Description Framework (RDF) is a framework for representing information on the Web. First, it is important to understand what resources are. Any IRI (Internationalized Resource Identifier) or literal (simple values, such as numbers, strings, dates) denotes something in the world (the "universe of discourse") [9]. These things are called resources. Resources are for example documents, students, numbers, any kind of animal, scientific phenomenon, basically anything.

There are multiple concepts that RDF is following [9]:

- Graph data model
- URI-based vocabulary
- Data Types

- Literals
- XML serialization syntax
- Expression of simple facts

The expression of simple facts is achieved by the data structure, which is called triple. One expression is one triple, which represents the relationship between resources. Triples consist of a subject, predicate, and object. Here are some examples :

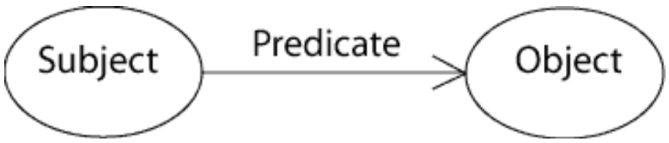

Figure 2.2: Triple construction [10]

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

Listing 2.1: Set of triples [11]

Set of triples from example 2.1 generates graph on figure 2.3:

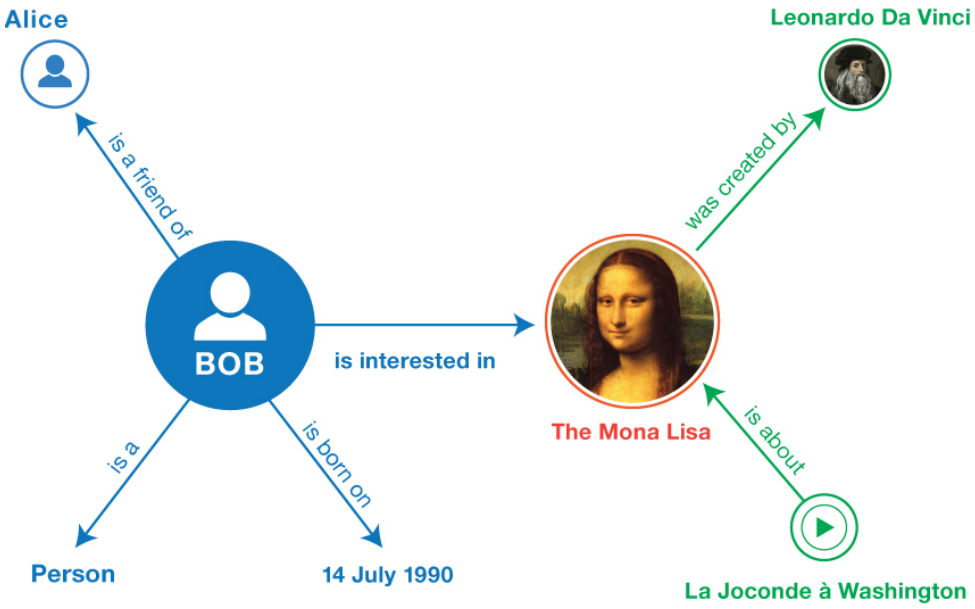

Figure 2.3: Graph example[11]

2.6 RDFS and OWL

RDFS is an extension of RDF that allows describing taxonomies and memberships in classes to represent domain semantics more [12]. However, RDFS is not sufficient for modeling complex semantic relations and constraints. Therefore, OWL, an ontology language that extends the RDFS vocabulary and provides a logical layer for reasoning about knowledge, was developed. OWL allows to describe elements such as non-overlapping and symmetric relations, cardinality, equality and enumerated classes. It also distinguishes between data and object properties. It is to be noted that OWL ontologies are primarily exchanged as RDF documents as each OWL document can be serialized as an RDF document.

2.7 RDF serialization formats

There are multiple ways of encoding RDF data. RDF document uses specific syntax for representing RDF graphs or RDF datasets. These syntaxes are Turtle, RDFa, JSON-LD, TriG and others. This part will describe only RDF syntaxes that are used in the SPipes project.

2.7.1 JSON-LD

JSON-LD (JavaScript Object Notation for Linked Data) is a lightweight Linked Data format. It is easy for humans to read and write. It is based on the already successful JSON format and provides a way to help JSON data interoperate at Web-scale [13]. One of the main benefits of JSON-LD is that it allows data to be embedded directly into web pages, making it easier for search engines and other software to understand and process the data. In addition, this syntax has the clearest and most readable structure for a person less trained in linked data. This is achieved thanks to JSON, the basis of this RDF syntax.

```
{
  "@context": "https://json-ld.org/contexts/person.jsonld
   ",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Listing 2.2: Example of JSON-LD linked data format [13]

The @context keyword is used to associate a context with a JSON-LD document, allowing the data to be interpreted and processed in a standardized and interoperable way. It helps to ensure that the meaning of the data is properly understood by machines, and facilitates data integration and exchange in a linked data environment.

2.7.2 Turtle

Turtle is a very human-friendly syntax because it greatly reduces the number of characters needed to write triples and reduces duplication of information. Turtle uses the following techniques [14]:

- Prefixes are defined at the beginning of each file, which then replaces the long IRIs.
- If several expressions are written for the same subject, they are separated by a semicolon and the next expression doesn't need to use the subject, it will be determined automatically from the first one. If you want to define several objects with the same predicate for the same subject, the objects are separated by a comma.
- Replacing rdf:Type with a simple a. This way we don't need to create an extra prefix and complicate reading the file.
- Each set of expressions that refer to the same subject ends with a dot.

```
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix schema: <https://json-ld.org/contexts/person.
   jsonld#> .

dbpedia:John_Lennon a schema:Person ;
    schema:name "John Lennon" ;
    schema:born "1940-10-09" ;
    schema:spouse dbpedia:Cynthia_Lennon .
```

Listing 2.3: Example of Turtle on same data as on listing 2.2

2.8 SPARQL

SPARQL is a query language for RDF. SPARQL allows users to specify patterns in the data they want to retrieve and then retrieve any data that matches those patterns. It has a syntax similar to SQL, but is more expressive and can be used to query a wide variety of data sources, including databases, files, and web services [15].

The following example retrieves from the database names and birthdays of authors that were born between 1900 and 2000 years. The SELECT clause specifies what variables should be in the output, the same as in SQL languages and the WHERE clause specifies the pattern that should be matched for the result:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?name ?birthYear
WHERE {
 ?person a dbo:Artist .
 ?person foaf:name ?name .
 ?person dbo:birthYear ?birthYear .
 FILTER(?birthYear >= 1900 && ?birthYear < 2000)
}
```

Figure 2.4: Sparql example query

Translating this SPARQL query into human language, the result should match these statements:

- Person should have <http://dbpedia.org/ontology/Artist> type.
- Name should be taken from <http://xmlns.com/foaf/0.1/name>.
- Birth Year should be taken from <http://dbpedia.org/ontology/birthYear>.
- Filtering results, by birth year so only results with year more than 1900 and less than 2000 are taken.

2.9 SPARQLMotion

SPARQLMotion is an RDF-based scripting language with a graphical notation to describe data processing pipelines [16]. SPARQLMotion is designed to make it easy to create complex data processing pipelines by chaining together a series of simple SPARQL queries. The main idea of SPARQLMorion is to make it possible to pass results from one process step to another, creating a chain. As the name of the language says, behavior in each module is driven mostly by SPARQL queries. These queries are used for iterating through result sets, constructing new RDF triples, performing updates to RDF data sources, and many more ways of interacting with linked data. The SPARQLMotion language itself is a fairly lightweight collection of classes and properties used to represent SPARQLMotion scripts in RDF.

For a better understanding of SPARQLMorion, it's needed to describe some key concepts. These concepts are [1]:

- Script: A SPARQLMotion script is a sequence of steps written in RDF that define the processing and manipulation of data. Each step in the

[1]The SPARQLMotion system vocabulary is found in the namespace http://topbraid.org/sparqlmotion, which is typically abbreviated with the prefix sm.

workflow is represented by a SPARQLMotion module, which performs a specific task.

- Module: Modules can be connected to each other with different relationships. Each processing step in SPARQLMotion is called a module. SPARQLMotion provides a range of predefined modules that can be used to build a workflow, such as modules for querying data, transforming data, and writing data to a file or database.
- Variable: SPARQLMotion allows users to store data in variables, which can be used to pass data between different modules in a workflow.

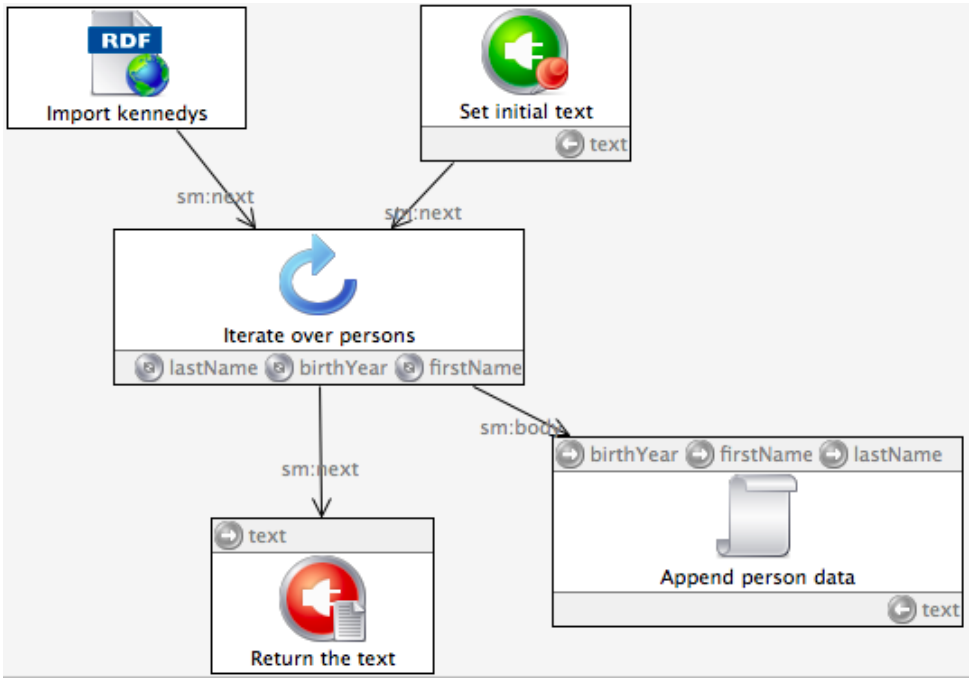

Figure 2.5: Sparql Motion flow example [16]

2.10 RDF4J and RDF4J repository

RDF4J is an open-source Java framework for working with RDF data. It provides a set of APIs and tools for creating, storing, querying and manipulating RDF data. The framework supports several RDF serialization formats, including RDF/XML, Turtle, N-Triples, and JSON-LD, and provides a number of query languages, including SPARQL, SeRQL, and RDF4J's own query language [17].

RDF4J includes an implementation of the RDF4J repository, which is a database system designed to store and query RDF data. The RDF4J repository can be used with a variety of data sources, including in-memory

databases, file-based databases, and remote databases accessible over a network. The repository supports transactional updates, versioning, and queries to large datasets, making it suitable for use in a variety of applications [18].

RDF4J is widely used in the Semantic Web community to develop and deploy applications that work with RDF data. Its flexible APIs and support for multiple query languages and serialization formats make it a powerful tool for working with RDF data.

2.11 HATEOAS

HATEOAS is one of the principles of the REST architecture, which provides a flexible way to build web services [19]. Rather than rigidly defining how clients can interact with resources, HATEOAS suggests using hyperlinks so that clients can access resources and manage the application state.

Specifically, HATEOAS defines that each resource must contain links to other resources that are associated with it. This allows clients to automatically discover new opportunities to interact with resources without having to know about them in advance.

The HAL (Hypertext Application Language) protocol is one example of the use of HATEOAS. It presents resources as JSON or XML documents that contain links to other resources and additional meta-information. HAL is a simple format that gives a consistent and easy way to hyperlink between resources in API [20].

```
{
  "id": 1,
  "name": "Product 1",
  "_links": {
    "self": { "href": "/products/1" },
    "category": { "href": "/categories/2" }
  }
}
```

Listing 2.4: Example of JSON-LD linked data format

The example 2.4 shows the use of HAL, where self is the link that was used to get this data. And the category link leads to the category under which this product is located.

2.12 SPipes

SPipes is a tool for managing semantic pipelines defined in RDF inspired by the SPARQLMotion language [21]. Each node or other word module in a pipeline represents some stateless transformation of data [21]. SPipes allow not only passing variables from module to module but also declaring global variables, which can be used in any data transformation step. Modules for

SPipes pipelines can be generated in two ways: either directly in script or in Java. The tool supports only scripts written in one RDF syntax Turtle.

In this work, you can often come across the concepts of module execution and pipeline execution. At the moment when the pipeline is launched, we can store the data about the pipeline's execution. This information is saved in the RDF4J repository and contains such data as when the pipeline was launched, which modules were executed in the pipeline, where inputs and outputs of modules are stored, and other useful information.

Semantic pipeline execution data can be stored in the RDF4J repository within AdvancedLoggingProgressListener, implemented in SPipes. This is a key aspect for accomplishing the goals of this thesis.

In the work "Debugging scripts in SPipes editor" by Bc. Petr Jordán [22] I found an excellent diagram describing the concepts used in SPipes. I used this diagram as a base and extended its contents a bit. The diagram is shown in figure 2.6

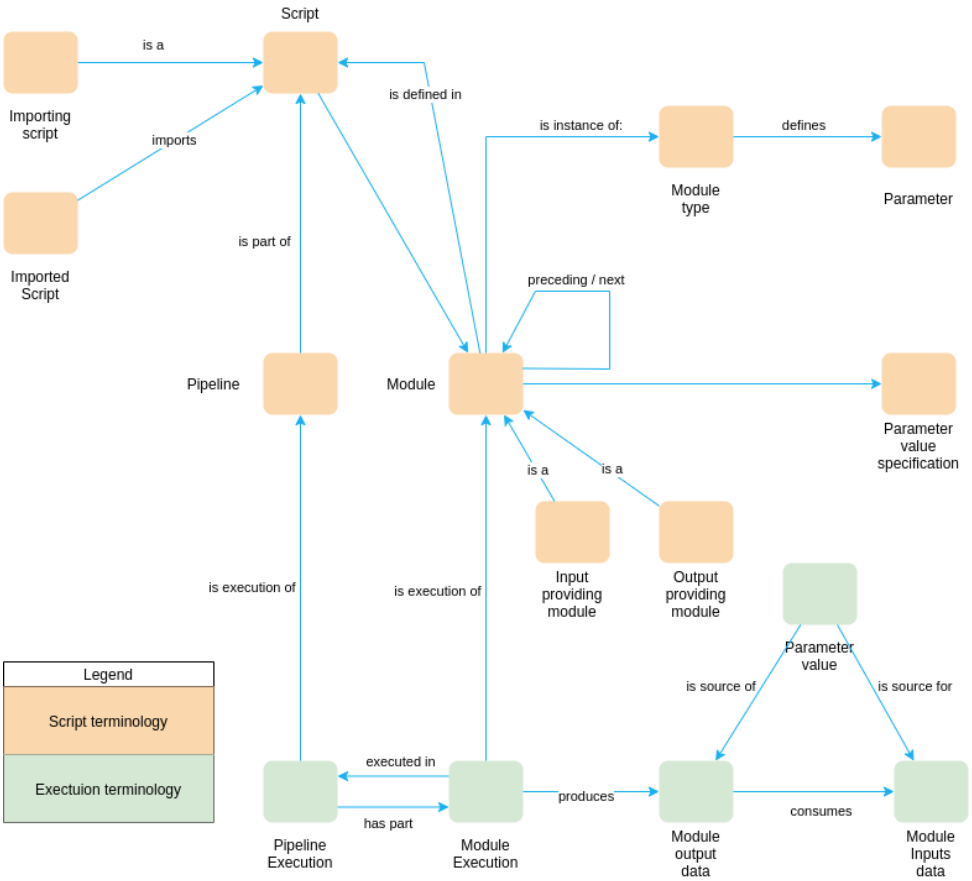

Figure 2.6: SPipes language terminology

Chapter 3

An overview of the debugging capabilities of SPipes and similar tools

In this section, I'll take a look at other tools similar to SPipes, as well as an overview of their debugging capabilities. At the end of the chapter, I will describe how to debug in SPipes.

3.1 ETL

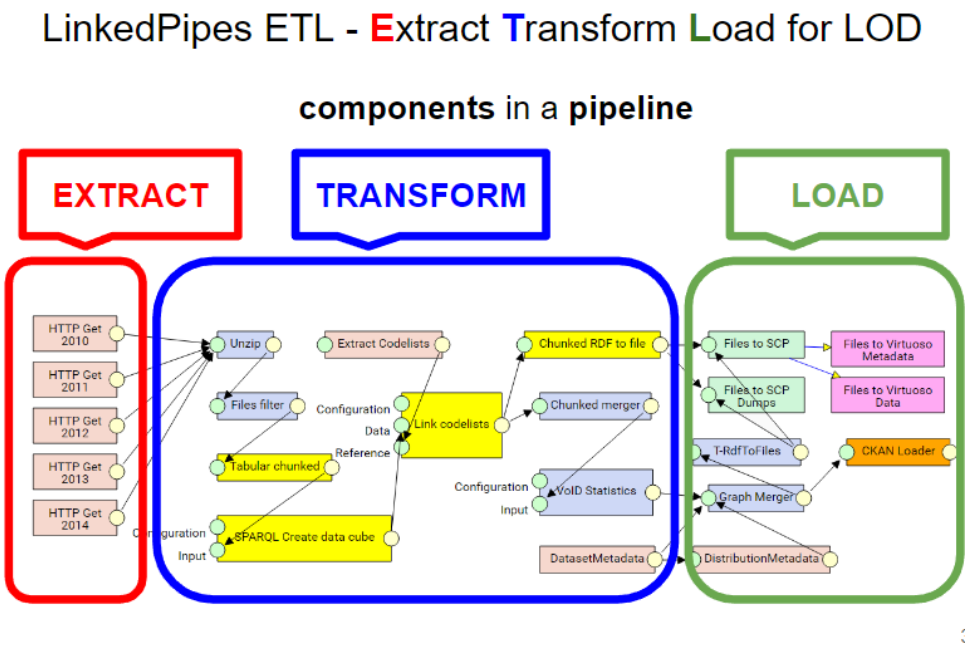

Figure 3.1: Screenshot of a pipeline in LinkedPipes ETL [23]

ETL is an acronym that stands for Extract, Transform, Load (see figure 3.2). In the context of LOD publishing tools, ETL refers to the process of properly lifting the source data to RDF as well as covering other operations required for publishing data [24]. Many of these tools have a narrow focus and only

support data transformation, but ETL involves much more than just data transformation. It also includes:

- Gathering data from its source.
- Creating metadata.
- Publishing data and metadata on the web.
- Creating appropriate catalog records.
- Loading the data into a triple store.

Therefore, ETL is an essential part of the LOD publishing process, and a tool that supports the full ETL process is necessary for successful LOD publication, particularly for public administrations.

3.2 LinkedPipes ETL

LinkedPipes ETL is a tool designed to support the process of publishing datasets as Linked Open Data [24]. It allows users to create data transformation pipelines consisting of configurable components that produce and/or consume files or RDF data. The tool has a library of more than 60 reusable components, and new components can be easily developed. LinkedPipes ETL is based on open APIs and is configured based on the Linked Open Data principles. Its typical workflow includes adding an extractor component to the source data, adding necessary transformation steps to the pipeline, transforming raw RDF representation to RDF modeled using proper vocabularies, uploading data to a web server or a triple store, and describing data with metadata.

3.2.1 Debug in LinkedPipes ETL

When a failure occurs, LinkedPipes ETL displays a graphical view that shows the point of failure along with an error report [24]. The report contains an execution list. Clicking on the execution in the list opens a graphical execution overview that shows what went wrong. You can also click on a data unit, represented by a circle on a component, to browse its contents and investigate the source of the error. Clicking on the component itself provides details about the failed execution. In order to fix the failed component, you must switch to edit mode and make the necessary changes.

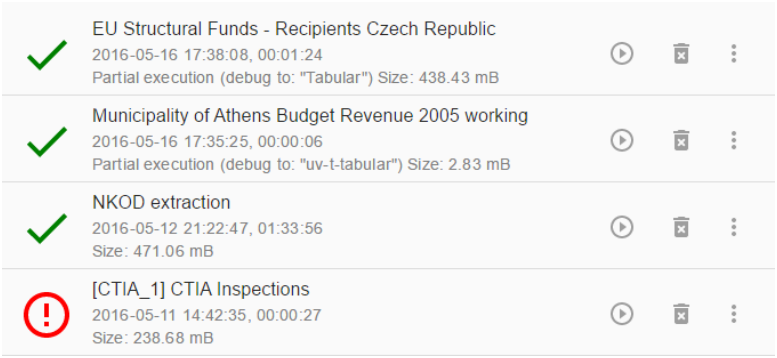

Figure 3.2: Report with execution list [25]

LinkedPipes ETL also offers "debug from" and "debug to" features:

- "Debug from" allows designers to run only the part of the pipeline that failed in the previous execution. This feature saves time and resources by reusing the data already gathered in the previous execution, thereby eliminating the need to re-run the entire pipeline.
- "Debug to" feature allows designers to run only the necessary parts of the pipeline to execute a specific component, such as the section leading up to the failed component. This selective execution ensures that resources are used efficiently, and the rest of the pipeline remains unexecuted.

3.3 OpenRefine

OpenRefine is an open-source desktop tool for cleaning and converting dirty data [26]. The application looks like a spreadsheet, but works like a database in a web browser, providing an easy-to-use interface for novice users, as well as offering advanced features and capabilities for more experienced users.

OpenRefine allows users to import data from a variety of sources, including CSV, TXT, JSON, XML, XLS, MARC, and RDF. After importing data, users can apply various data-cleaning strategies and algorithms using OpenRefine's built-in functions or by writing their own expressions in GREL (General Refine Expression Language), Jython (Python), or Clojure.

OpenRefine also offers a number of extensions, matching services, and client libraries. One of the extensions is RDF [27]. The RDF extension is a tool for creating linked data using OpenRefine. It allows users to convert data into RDF format and define relationships between different data elements.

3.3.1 Debug in OpenRefine

Since the work with OpenRefine is not by running some script, but in such a mode that gradually applies any changes to certain data, OpenRefine does not contain many debugging instruments.

OpenRefine has an undo/redo feature that allows users to revert changes made to their data [28]. The change history is saved with the project's data,

and users can view and undo changes after restarting OpenRefine. To revert data back to an earlier state, users click on the last action in the timeline they want to keep. Users can also reuse operations performed in OpenRefine by extracting and copying JSON-encoded operations. Not all operations can be extracted, such as edits to a single cell.

3.4 Debugging capabilities of SPipes

Unfortunately, I could not find any documentation describing the debugging features of SPipes. The debugging capabilities will be described based on my personal experience with SPipes and consultations with the supervisor of this thesis.

If the pipeline has not been completed, the script designer immediately sees in the logs which module, stopped the work of the pipeline. Further, SPipes has the ability to provide all necessary data to the module and start only one module, without continuing in the pipeline. Each executed module can be executed on the same inputs. This way user can change the configuration of the module and test it on previous input. It allows users to debug specific modules fast.

In addition, SPipes has the ability to save the input and output of modules to files and save execution metadata to the RDF4J repository. Using these features, the developer could use information about the progress of pipeline execution to debug the script.

For debugging SPipes scripts, there are several other tools that are enabled by enabling these tools in the configuration file:

- Property audit.enable. When true, then Turtle format files are generated for each executed module, which contains the inputs and outputs.
- Property execution.checkValidationConstraints. When true, then another mechanism available to SPipes comes into effect: Constraints. Constraints are used to specify rules for the data, thus verifying their correctness at the output.
- Property contextsLoader.data.keepUpdated. When true, all changes made to the script are instantly applied when the script is launched.

3.4.1 Debugging process in SPipes

1. Set the execution environment to development by setting execution.environment property to development.
2. Create a script and execute it.
3. If any validation constraints fail, take the following actions:
    - Check the output of the pipeline execution.

- If the output is not satisfactory, examine other inputs/outputs along the module's path.

4. If the problem you solved might occur later or you want to document how the module works, create a new validation constraint.
5. Check the output by either examining the Turtle file (generated by enabling of property audit.enable) or loading it into RDF4J and querying it.
6. Re-run the module.

Chapter 4

Requirements

The chapter presents the requirements analysis for the SPipes debug API project. It covers activity diagrams with debugging scenarios, requirement categorization using the MoSCoW method and lists the functional and non-functional requirements for the project.

4.1 Debugging scenarios

In this chapter, I will describe possible scenarios of malfunctioning pipelines with possible solutions to these problems. Then from these scenarios, we will derive the requirements for a future solution.

In general, these scenarios can be divided into two categories:

- Optimization scenario, which describes where you can find problems with the slow execution of the pipeline. Figure 4.1 represents the process of finding problems related to the slow performance of pipelines.

- Scenario for error detection, which allows the developer to find errors during the implementation of the SPipes script. Figures 4.2 and 4.3 describe the solution to such problems as the absence of the desired triple in the output, or, conversely, the presence of an unexpected triple in the output of the pipeline (please note, that these two figures represent one schema).

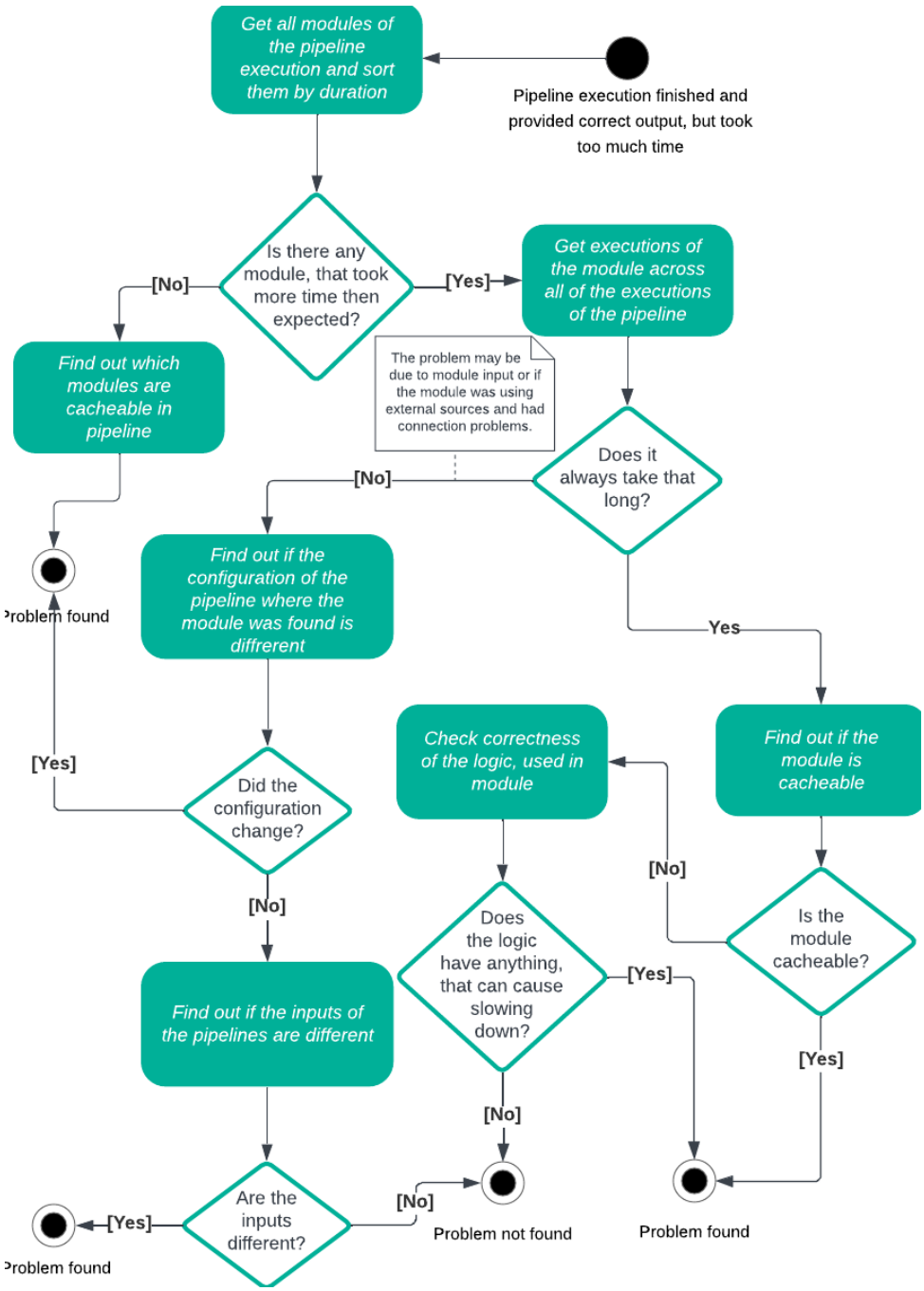

Figure 4.1: Activity diagram for time optimization of pipeline

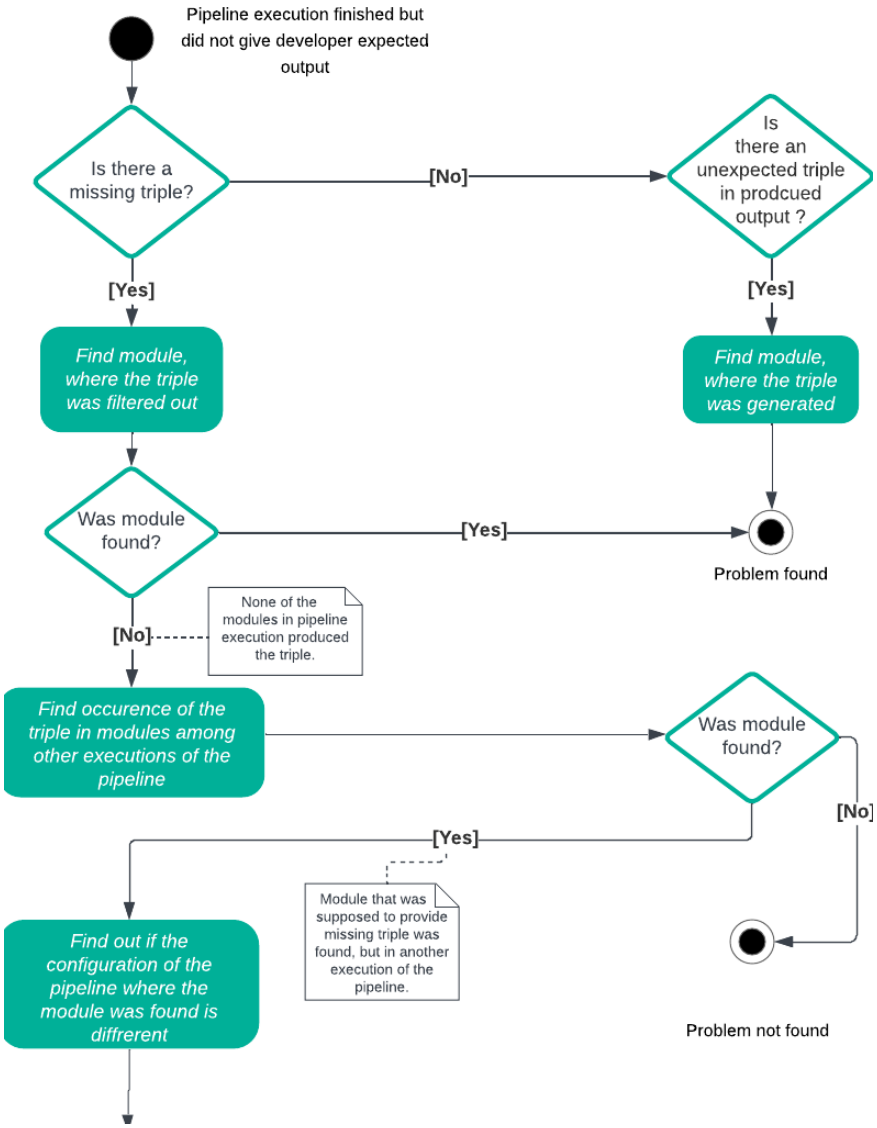

Figure 4.2: Activity diagram for error findings part 1

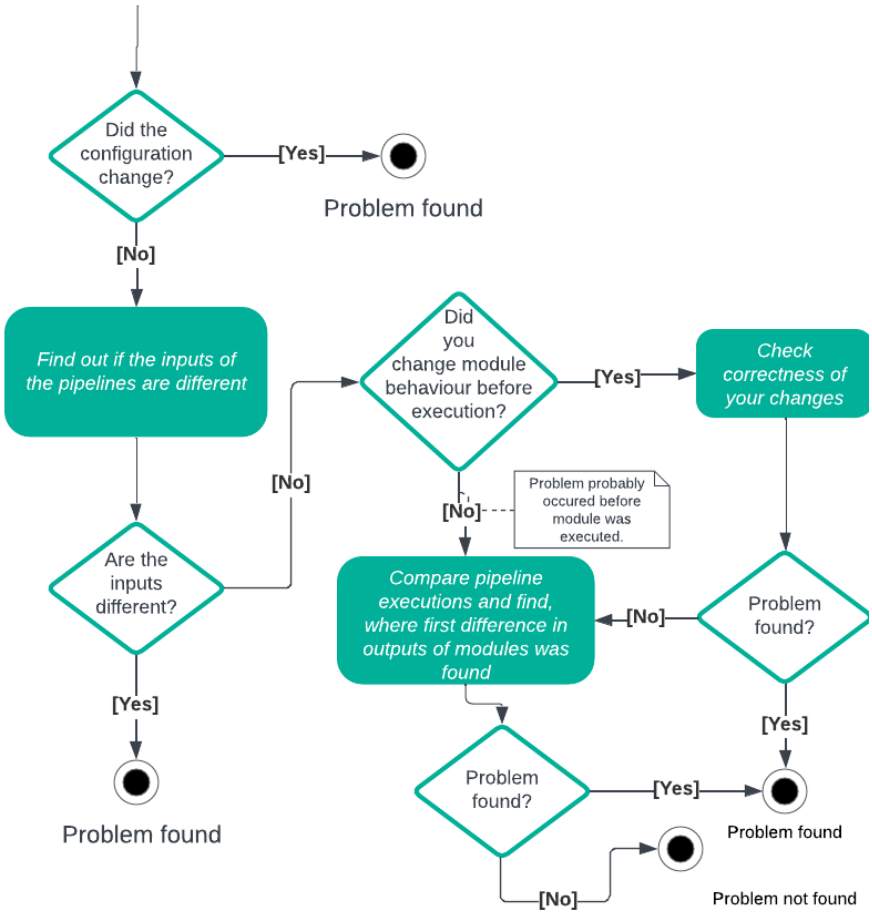

Figure 4.3: Activity diagram for error findings part 2

4.2 Analysis of scenarios

Based on the above diagrams, it is possible to draw up a certain set of tools that SPipes scripts developers could use in the future. This set of tools is presented on figure 4.4 as a Use Case diagram.

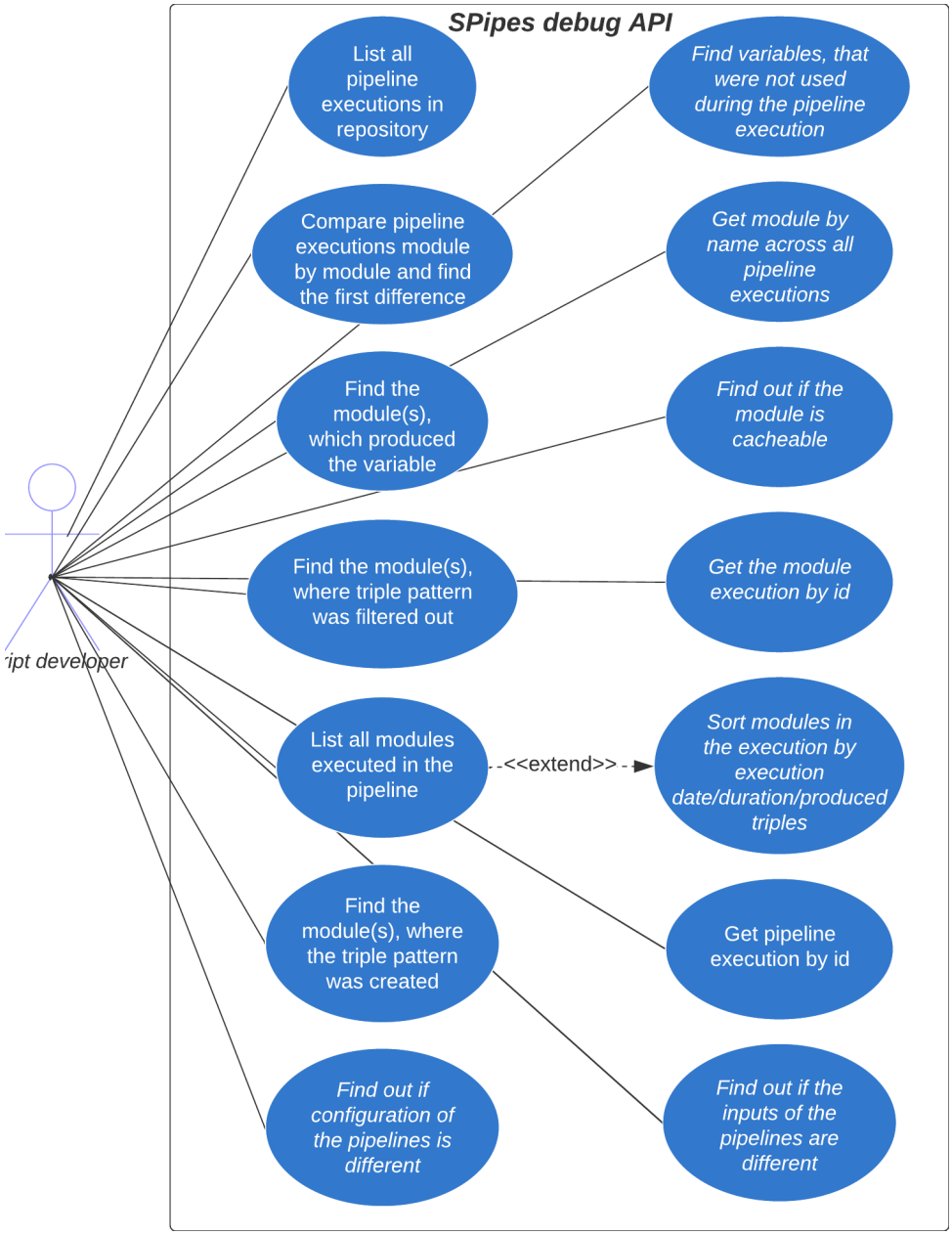

Figure 4.4: Use cases diagram of SPipes debug API

4.3 MoSCoW method

Requirements for this project are made using the MoSCoW method. All requirements are important, but they bring different values to the final product, some of them are critical and some of them are just nice to have. MoSCow method helps us to categorize them into 4 different groups [29]:

- Must have (M) - the most critical requirements for the final product. Even if at least one of them is failed, the whole work can be considered

as failed.

- Should have (S) - the second category of requirements, they are almost as important as the first category, but there can be other ways to satisfy the requirement, or they are not that time-dependent, so the requirement can be satisfied in another time box.
- Could have (C) - these requirements are not necessary, they usually improve user experience. Usually, they are done when there are resources available after completing the first two categories.
- Won't have (W) - Least-critical requirements, they are either dropped or can be planned in another timebox.

4.4 Functional requirements

Based on the Use Case created and in consultation with the thesis supervisor, functional and non-functional requirements for the future implementation of the project were created.

- FR1 (M) - The system allows the use of REST API.
- FR2 (C) - The system allows the use of simple UI for web API.
- FR3 (S) - The system allows displaying the count of output triples of each module.
- FR4 (S) - The system allows displaying the count of input triples of each module.
- FR5 (M) - The system allows displaying the execution time of each module.
- FR6 (S) - The system allows the finding of the module(s) that generated the given triple pattern.
- FR7 (S) - The system allows the finding of the module(s) that filtered out the given triple pattern.
- FR8 (S) - The system allows comparing pipeline executions and finding, where the first difference occurred.
- FR8.1 (S) - The system allows saving data about the comparison of two pipelines in the RDF4J database, such as if the pipelines are the same or if not, where the difference was found. So in subsequent requests, the results of the comparison are not calculated but simply returned to the user from the database.
- FR9 (S) - The system allows the finding of the module, which bounded variable with the provided name.

- FR10 (C) - The system allows finding cacheable modules.
- FR11 (M) - The system allows to list of possible debugging tools, related to returned entities after pipeline execution.
- FR12 (C) - The system allows getting module execution across all pipeline executions.
- FR13 (C) - The system allows finding variables that were not used during the pipeline execution.
- FR14 (C) - The system allows getting module execution by ID.
- FR15 (S) - The system allows detecting the configuration difference of the pipelines.
- FR16 (C) - The system allows the detection of the input difference of the pipelines.

4.5 Non functional requirements

- NFR1 (C) - Cover functionality with tests.
- NFR2 (M) - Dockerization of used technologies.
- NFR3 (S) - Test functionality with at least 3 users.
- NFR4 (M) - Create Swagger API documentation.
- NFR5 (M) - REST endpoints must return JSON-LD format.

Chapter 5

System Design

This chapter is intended to describe general system design decisions. It includes information about the selected architecture, system components, services, modules, and their interaction.

5.1 REST API

REST is an acronym for REpresentational State Transfer and an architectural style for distributed hypermedia systems. Roy Fielding first presented it in 2000 in his famous dissertation[30].

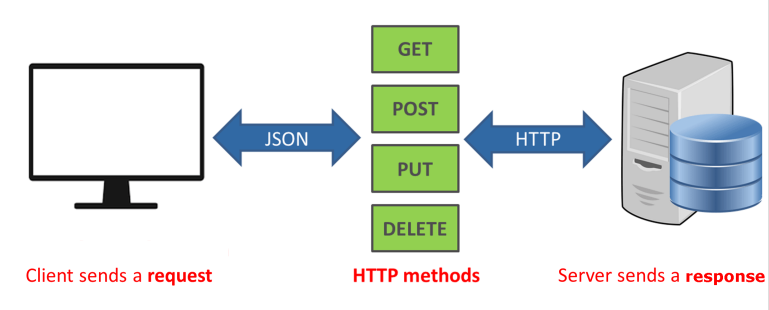

Figure 5.1: Rest API client-server architecture [31]

The basic REST principles described in the book "RESTful Web Services" by Leonard Richardson and Sam Ruby include [19]:

- Client-server architectural style shown on figure 5.1: The system is divided into clients, which initiate requests, and servers, which process requests and provide resources (e.g., data or functionality).
- Stateless: Each client request to a server must contain all the information necessary to process it, without storing state on the server between requests. The server should not remember previous requests from the client.
- Caching: Server responses can be cached on the client so that repeated requests for the same resources can be processed faster and reduce the

load on the server.

- Uniform Interface: The interface between client and server must be simple, uniform, and limited. It should contain a minimum set of operations (eg, CRUD - create, read, update, delete) and use standard HTTP methods (eg, GET, POST, PUT, DELETE) to access resources.

- Layered System: The architecture can be composed of multiple layers, where each layer performs specific functions. Clients do not need to be aware of the internal structure of the system and servers can be scaled horizontally by adding additional layers.

All of the above principles should be applied in the development of the SPipes debug API. These REST principles provide simplicity, scalability, reliability, and performance in developing web services, making them easily accessible and interoperable with different clients and platforms.

Figure 5.1 shows the data transfer in JSON format, as SPipes works primarily with Linked Data, instead of JSON format the data are transferred to the client side in JSON-LD format, which is described in the section 2.7.1.

5.2 Three-layer architecture

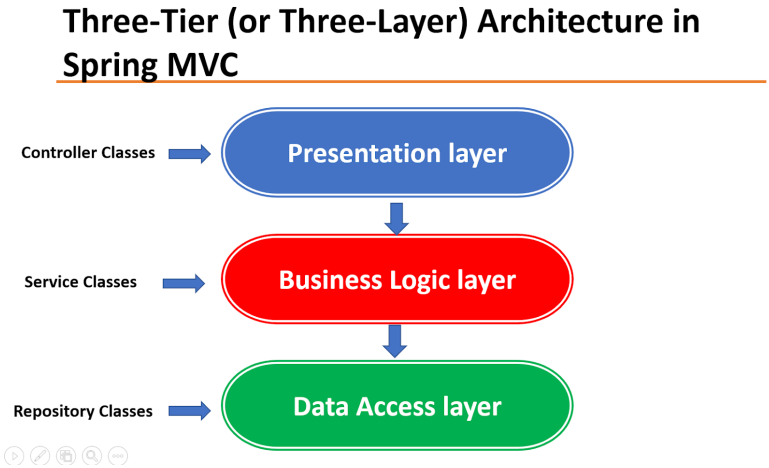

Figure 5.2: The three-layer architecture [32]

A three-layer architecture is one of the common approaches to organizing program code and application functionality. It consists of three main layers: the presentation layer, the business logic layer, and the data access layer [32].

The presentation layer is responsible for user interaction and data display. It includes the user interface, user input/output components, and the logic for handling user actions. This layer is separate from the business logic and data access layer, and its purpose is to provide data display and user interaction.

The business logic layer contains the core application logic. It handles business rules, data validation, query processing, and business process decisions. The business logic layer is the heart of the application and its job is to ensure that the business logic functions properly and maintains data integrity.

The data access layer is responsible for interacting with databases, external services, or other data sources. It provides functions for reading, writing, and updating data, as well as for handling transactions and managing database connections. Separating the data access layer from the other layers allows for the separation of responsibility and simplification of system support and modification.

5.3 Modules and components

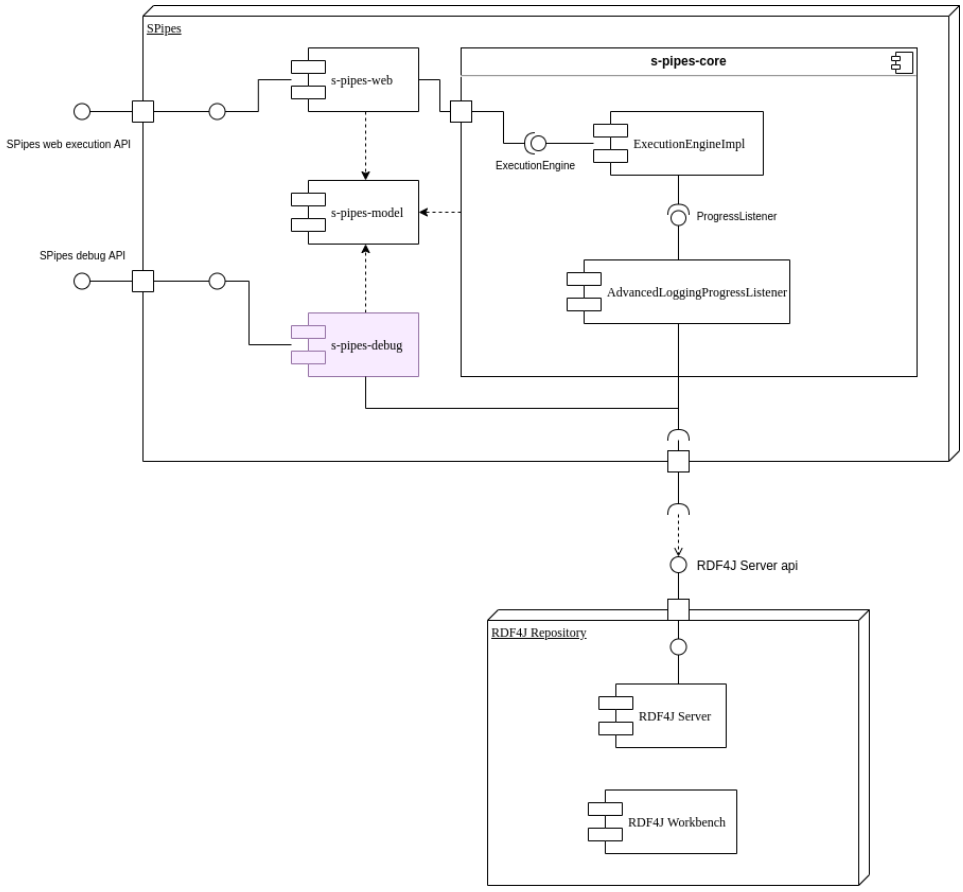

Figure 5.3: Component diagram, showing interaction of spipes-debug-module with other modules and components

The component diagram 5.3 shows part of the SPipes project and the RDF4J repository. The diagram shows all the important components that the s-pipes-debug module communicates with, either directly or indirectly.

It starts with the flow, which is used to run the semantic pipelines. One

way to run a pipeline is to use the s-pipes-web module, which provides a WEB API to run a pipeline or some parts of a pipeline. The module then delegates the work on the execution of the pipelines themselves to the s-pipes-core, where all the important calculations take place. The s-pipes-core contains ProgressListeners, which are executed when the pipelines are started before each module is started, after each module is completed, and after the whole pipeline is completed. One class that implements the ProgressListener is the AdvancedLoggingProgressListener, which writes some data about the execution of a module or a pipeline to the RDF4J repository.

The s-pipes-debug module, which will be implemented as part of the thesis, should use these data by pulling it from the RDF4J repository, processing it, and returning it to the user in the desired format.

5.4 Entity model

Figure 5.4 shows the Entity Model used in the SPipes debugging API. Entities from this model are saved to the RDF4J repository when using JOPA (see 6.1.3).

There are not many entities in the diagram as all the work in the project is centered around ModuleExecution and PipelineExecution. Both entities are inherited from the Thing entity. Thing refers to a fundamental concept that represents any entity or resource that can be identified, described, or referenced on the web. Also on the diagram, you can see the PipelineComparison entity, which is a comparison of two PipelineExecution entities.

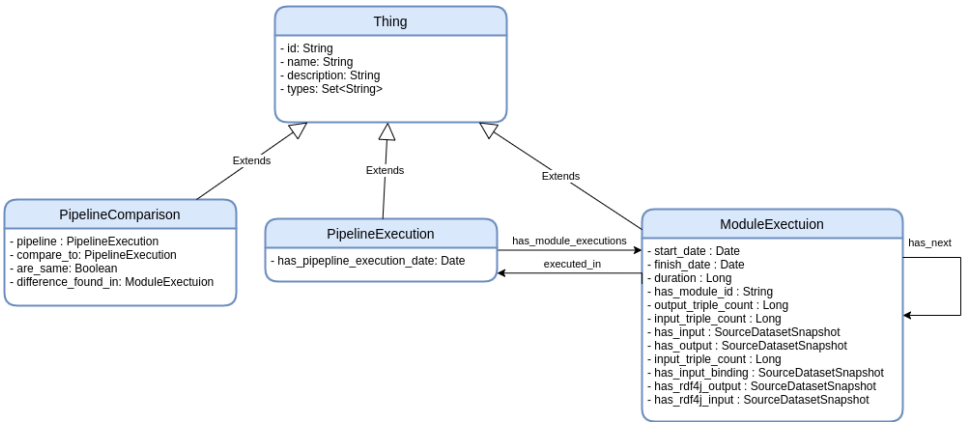

Figure 5.4: Model diagram of entities, used in SPipes debug API

Chapter 6

Implementation

This chapter will describe the technologies I worked with to implement the solution. Later in the chapter, some details of the solution are described, and at the end the results of the implementation.

6.1 Technology stack

This section will describe the main technologies used in the implementation of the solution

6.1.1 Java

SPipes is written in the Java programming language, and since a lot of functionality will be used from the existing project, it was decided to create a new Maven module in the SPipes project [1].

Java is an object-oriented programming language developed by Sun Systems [33]. Its most important features are the unopened language and the so-called Java Virtual Machine (JVM). The JVM allows to write code once and then run the program on any operating system as the operations are not performed directly on the hardware but through JVM. Java compiles the code into what is called byte code, then the JVM interpreter translates this byte code into operations that the specific operating system understands.

Java is also known for its extensive class library, which contains many tools and functions that simplify the development process. Java class library contains an extensive set of methods and tools that allow developers to reduce the time and effort required to write applications [34].

6.1.2 Spring framework

Spring is a popular open-source framework for creating Java-based applications. Thanks to its adaptability and extensive feature set, it has become very popular among developers since its first release in 2002.

The Inversion of Control (IoC) container, which controls the lifecycle of objects and their dependencies, is one of the key components of Spring. As

[1] https://github.com/kbss-cvut/s-pipes

a result, components can communicate freely, making it simple to modify and test specific application components. In addition, Spring provides a wide range of modules and tools for various aspects of development. [35]

Spring is the standard of modern Java application development, the framework supports a huge number of technologies. [36] It is also important to note the development-friendly architecture of spring applications, thanks to which the application is easy to test and extend.

6.1.3 JOPA

JOPA is a Java OWL persistence framework aimed at efficient programmatic access to OWL2 ontologies and RDF graphs in Java. The system is based on integrity constraints in OWL that JOPA uses to establish the contract between a JOPA-enabled Java application and an OWL ontology. The system architecture and API are similar to JPA 2.1. [37]

Code snippet of a model java class using JOPA for accessing RDF graph from JOPA examples repository [38]:

```
@OWLClass(iri = Vocabulary.Superhero)
public class Superhero implements Serializable {

    @Id
    private URI uri;

    @OWLDataProperty(iri = Vocabulary.p_firstName)
    private String firstName;
}
```

Iri in the code snippet is String. For example:

```
public static final String p_firstName = "http://krizik.felk.
    cvut.cz/ontologies/jopa/example05#firstName";
```

6.1.4 Docker

Docker is a platform for automating the deployment and management of applications in a containerized environment. Docker uses containerization technology that isolates an application and its dependencies from the operating system on which it runs. This allows developers to package applications and their dependencies into a single container that can run on any platform that supports Docker [39].

One of the main benefits of Docker is the ability to create containers based on different images. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [40]. Image-based container creation provides uniformity in application deployment and simplifies the scaling process.

Thus, Docker provides developers with a convenient and easy way to package applications and their dependencies into a single container, which greatly simplifies the process of deploying and managing applications. In

addition, Docker provides security and stability to the system as a whole because applications and their dependencies are isolated from the rest of the processes on the host system.

6.2 Execution tree

Often at the end of the execution, the developer is confronted with the results that he would like to find the source. For example, we have written a simple script, and there is a variable in the output. In order to find the module, that created this variable, we need to know in which order the pipeline modules were executed. In the RDF4J repository, each module execution has the field "has_next" (see section 5.4), leading to the next execution of the module. Using this data, we can build a tree where the modules that ran first are at the very bottom of the tree and the root is the last module, called the return module.

To build the tree ExecutionTree class is used. By passing all the module executions of pipelines into its constructor, the algorithm will wrap each module into a ModuleExecutionNode which has additional parameters such as inputExecutions, execution, and the depth at which the node is located. We need the depth to find the earliest modules. Keeping the depth is necessary in order to find multiple modules. This can happen if the condition for finding has been fulfilled for several modules at once at the same depth.

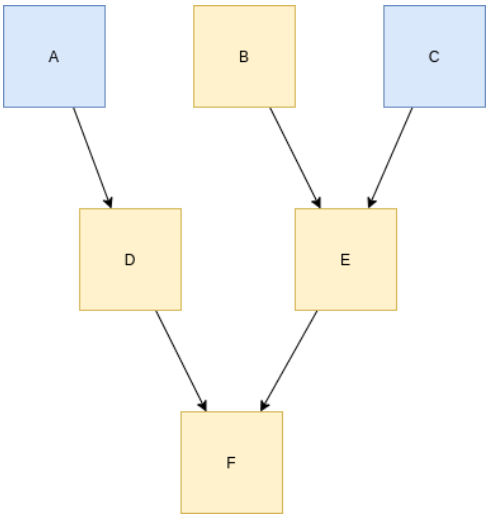

Figure 6.1: Execution tree

In the following example, refer to figure 6.1

Imagine that we are looking for which module first generated the firstName variable. By passing all pipeline execution modules to the Execution Tree constructor, we will build a tree. Next, using other s-pipes-debug-module mechanisms, we will find in which modules the firstName variable appeared. Suppose these were modules A, C, D, E, and F. Using the findEarliest method

and passing the ModuleExecutions identifiers to it, the method will return modules A and C to us since they were executed before the others.

6.2.1 HATEOAS and Linked Data

Unfortunately, I could not find ways to apply existing libraries such as Spring-HATEOAS. Spring required me to inherit from a class that had a field responsible for resources. Unfortunately, I couldn't mark this field with the @OWLObjectProperty annotation from the JOPA (see section 6.1.3) library. It made it impossible for me to offer JSON-LD format to the user correctly when the result arrives. I decided to write my implementation using only a method from Spring-HATEOAS that could access the controller methods and make the URL.

Related resources are done with the help of ResponseBodyAdvice, according to official spring documentation, ResponseBodyAdvice allows customizing the response after the execution of an @ResponseBody or a ResponseEntity controller method but before the body is written with an HttpMessageConverter [41]. So, when the controller produces PipelineExecution or ModuleExecution I can capture the object and modify its content e.g. add related resources based on the returned entity type.

Figure 6.2: Related resources example

6.3 Three-layer architecture

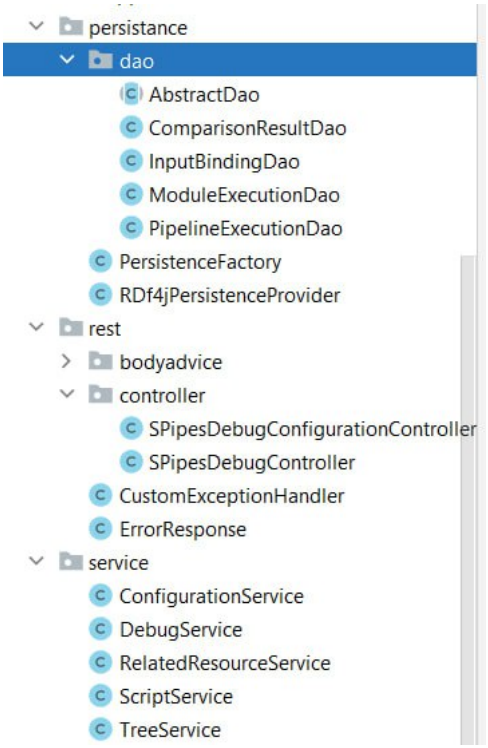

Figure 6.3: The three-layer architecture in the s-pipes-debug module

The screenshot on figure 6.3 shows that the s-pipes-debug module, written as part of the thesis, is written according to a three-layer architecture. There is no logic in the controller package, only controller methods, which delegate all the work to service methods. The service methods in their turn use the DAO layer to retrieve data from the RDF4J repository and exchange data with the database.

6.4 The result of implementation

As a result of the implementation, the s-pipes-debug module was created in the SPipes project. During the development, 3,899 lines were added.

Changes were also made outside the s-pipes-debug module since some logic was added or changed in such modules as s-pipes-core, s-pipes-model, and s-pipes-web. The changes did not critically change the original SPipes processes in any way. The main change outside the s-pipes-debug module was the removal of the Transformation class, which represented pipeline executions and module executions, and replacement it with PipelineExecution and ModuleExecution, which would allow to better define areas of responsibility and give an understanding of which object is handled, not based on the object name alone.

In s-pipes-debug itself, several of the tools shown in diagram 6.4 are implemented.

No domain objects are returned from controllers, instead, DTOs (Data Transfer Object) are used. To transform domain objects to DTOs, mapstruct mappers are used that interface perfectly with Spring and are defined as a component by adding @Mapper annotations.

To get data from the RDF4J repository in a DAO layer, I used JOPA (see section 6.1.3), an example of DAO class is shown on code snippet 6.1.

```
@Repository
public class ModuleExecutionDao extends AbstractDao<
    ModuleExecution> {

    protected ModuleExecutionDao(EntityManager em) {
        super(em);
    }

    public Boolean askContainOutput(String context, String
    graphPattern) {
        try {
            return (Boolean) em.createNativeQuery("ASK {"
                    + "  GRAPH <" + context + "> {"
                    + graphPattern +
                    "  }}").getSingleResult();
        } catch (Exception e) {
            return false;
        }
    }

    public Boolean askContainInputAndNotContainOutput(String
    inputContext, String outputContext, String graphPattern) {
        try {
            return (Boolean) em.createNativeQuery(String.format
    ("ASK {"
                            + "  GRAPH <%s> {"
                            + "    FILTER NOT EXISTS {%s}"
                            + "  }"
                            + "  GRAPH <%s> {%s}"
                            + "}", outputContext, graphPattern,
    inputContext, graphPattern))
                    .getSingleResult();
        } catch (Exception e) {
            return false;
        }
    }
}
```

Listing 6.1: ModuleExecutionDao

The module is built in such a way that adding other tools shouldn't cause much of a problem.

6.5 Implemented functionality

This section describes all the features you get with the SPipes debug API. In the diagram 6.4 implemented features are marked with green color in the case diagram.

Figure 6.4: Use case with implemented cases

In figure 6.5, you can see endpoints for the implemented features from diagram 6.4, if you want to see more details about the endpoints, then you can find them in appendix C. Tables 6.1 and 6.2 show the fulfillment of requirements.

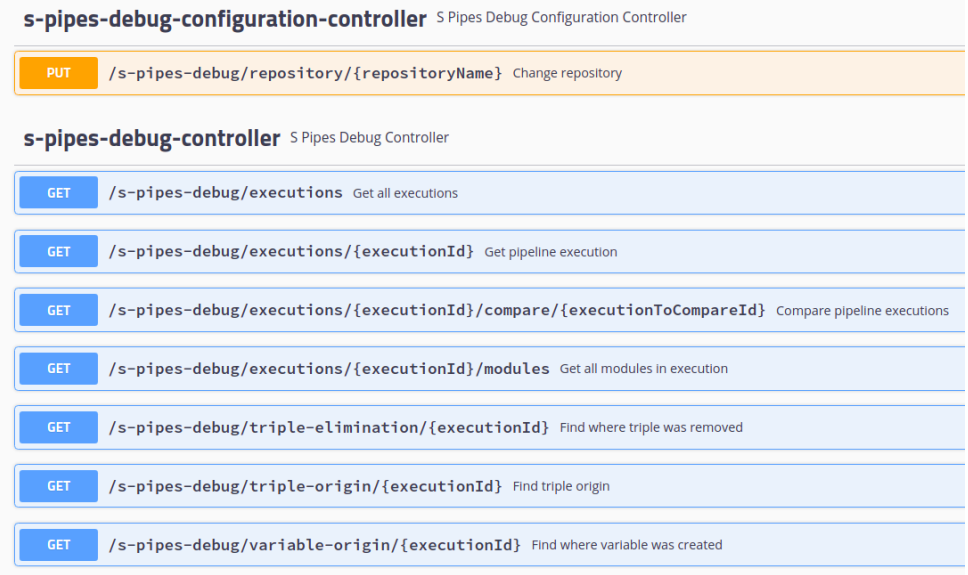

Figure 6.5: Swager API

Requirement Name	Status
FR1 (M) - The system allows the use of web API.	Implemented
FR2 (C) - The system allows the use of simple UI for web API.	Not implemented
FR3 (S) - The system allows displaying the count of output triples of each module.	Implemented
FR4 (S) - The system allows displaying the count of input triples of each module.	Implemented
FR5 (M) - The system allows displaying the execution time of each module.	Implemented
FR6 (S) - The system allows the finding of the module that generated the given triple pattern.	Implemented
FR7 (S) - The system allows the finding of the module that filtered out the given triple pattern.	Implemented
FR8 (S) - The system allows comparing pipeline executions and finding where the first difference occurred.	Implemented
FR8.1 (S) - The system allows you to save data about the comparison of two pipelines in the RDF4J database so that in subsequent requests, the results of the comparison are not calculated but simply returned to the user from the database.	Implemented
FR9 (S) - The system allows the finding of the module which bounded a variable with the provided name.	Implemented
FR10 (C) - The system allows finding cacheable modules.	Not implemented
FR11 (M) - The system allows listing possible debugging tools related to returned entities after pipeline execution.	Implemented
FR12 (C) - The system allows getting module execution across all pipeline executions.	Not implemented
FR13 (C) - The system allows finding variables that were not used during the pipeline execution.	Not implemented
FR14 (C) - The system allows getting module execution by ID.	Not implemented
FR15 (S) - The system allows the detection of the configuration difference of the pipelines.	Not implemented
FR16 (C) - The system allows the detection of the input difference of the pipelines.	Not implemented

Table 6.1: Table of functional requirements fulfillment

Requirement Name	Status
NFR1 (C) - Cover functionality with tests.	Not done
NFR2 (M) - Dockerization of used technologies.	Done
NFR3 (S) - Test functionality with at least 3 users.	Done
NFR4 (M) - Create Swagger API documentation.	Done
NFR5 (M) - REST endpoints must return JSON-LD format.	Done

Table 6.2: Table of Non-functional requirement fulfillment

6.6 Dockerization

In order to dockerize the SPipes debug API, I extended the existing docker file so that in addition to deploying s-pipes-web into the docker container, s-pipes-debug is also deployed. Because of problems with permissions in the RDF4J container, it was decided to extend the existing RDF4J image to provide the necessary rights. As a result, two docker containers are created, which can be run using the following docker-compose file. With the help of the supervisor, we have set up an automatic docker image publishing after each commit to the repository using GitHub actions. Thus, the latest image can be retrieved at ghcr.io/mircheqtm/s-pipes/s-pipes-engine. You can find code snippets for docker in appendix A.

6.7 Difficulties in implementation

- An unfamiliar topic.

  The topic of Linked Data, for me, was almost new and unexplored, I had heard about Linked Data but never encountered it in practice. Thus, I did not know many of the principles and technologies that are used in the semantic web. Before I started to write the paper or to collect the requirements I read and looked through a lot of sources and even with the acquired knowledge it was sometimes difficult to understand what the supervisor of my thesis had in mind. But the more I wrote Debug API, the more I understood and corrected mistakes made when writing the first lines of code in Debug API.

- Using a mix of DTO and domain objects.

  In the original version of SPipes, the RDF4J database stored a Transformation object, which represented both pipeline execution and module execution. When I started writing the s-pipes-debug module, I was afraid to change anything in the original implementation and decided to map Transformation to pipeline execution and module execution after getting it from the database, so I also returned this object to the client, which ended up containing not only dto objects but also domain. With this approach in the code appeared quite a lot of non general solutions,

which I had to fight with when implementing the new functionality. As a result, I decided to change the code in AdvancedProgressListener and change the Model so that the separate entities PipelineExecution and ModuleExecution were stored in the database. Thanks to this the code became much clearer and readable, there was a solid division into DTO and Domain objects. The potential scalability of the project was also improved.

Chapter 7

Evaluation

SPipes debug API testing will be based on the scenarios from Appendix B. The test scenarios will cover the use of tools such as:

- Finding the module where the variable was created.
- Finding the module where the triple was created.
- Showing all modules that have been executed in pipeline execution and sorting them by duration and number of output triples.
- Comparing pipeline executions and finding the module where the first output difference was found.
- Using of HATEOAS.

The testers will answer four questions for each scenario, one additional question for scenario number 4, and one question global for all scenarios.

Description of respondents

1. Tester 1: Backend Java developer. Has very little idea about Linked Data, and does not know anything about SPipes at all.
2. Tester 2: Backend Java developer. Has very little idea about Linked Data, and does not know anything about SPipes at all.
3. Tester 3: Developer. Knows semantic web technologies, and had a little knowledge of SPipes
4. Tester 4: Supervisor of the thesis. Knows semantic web technologies and knows SPipes very well.
5. Tester 5: SPipes Developer. Knows semantic web technologies and knows SPipes very well.

Questions

1. Was the scenario completed successfully?
2. How much did it take to complete the whole scenario?
3. Did you have any problems during scenario completion? If yes, what went wrong?
4. Is there anything you want to improve?

7.1 User testing

In this section, the answers of the 5 respondents to each of the scenarios will be presented.

7.1.1 Tester 1

Scenario 1

1. Yes.
2. 2 min.
3. Testing information does not contain information about JSON-LD structure, so it was a little confusing at the beginning.
4. -

Scenario 2

1. Yes.
2. 11 min.
3. Swagger UI did not offer any examples for comparing pipeline executions, so it was confusing and I had to use HATEOAS.
4. Add examples to swagger UI.

Scenario 3

1. Yes.
2. 8 min.
3. Swagger UI has no comments and examples, so the only option to execute the scenario is to look at HATEOAS links (in this case orderBy parameter names).
4. Add examples to swagger UI.

Scenario 4

1. Yes.
2. 10 min.
3. First time I've chosen the wrong execution and got 404, I needed to use the first execution.
4. -

7.1.2 Tester 2

Scenario 1

1. Yes.
2. 5 min.
3. -
4. -

Scenario 2

1. Yes.
2. 10 min.
3. I did not get any information, that differences in the pipeline executions should be found through some endpoint.
4. Add information in the testing scenario, that task should be done through debug API.

Scenario 3

1. Yes.
2. 6 min.
3. -
4. Rename "Module execution" related resources to something that makes more sense, for example, "Get module executions executed in the pipeline execution".

Scenario 4

1. Yes.
2. 8 min.
3. -
4. -

7.1.3 Tester 3

Scenario 1

1. Yes
2. 1 min
3. No
4. -

Scenario 2

1. No.
2. 5 min.
3. Yes, I was unable to complete the scenario.
4. I don't know. has_related_resources seems to return an unordered array, in one execution, the first object name is "Find triple origin", and in the other, it is "Get all module executions in the pipeline execution". Also, as I couldn't finish the scenario, improve the scenario description.

Scenario 3

1. Yes.
2. 2min 35s.
3. No.
4. The SwaggerDoc does not contain info about which parameters can be used as orderBy values. It also does not say that executionId is just the last part of the IRI of the execution.

Scenario 4

1. Yes.
2. 4 min.
3. I used the wrong execution ID originally.
4. -

7.1.4 Tester 4

Scenario 1

1. Yes.
2. 45 sec.
3. No.
4. Order of elements in JSON-LD, see my later comments.

Scenario 2

1. Yes.
2. 1 min 15 sec
3. Yes, it was a little misleading that the output was the actual module execution in which the difference occurred. The documentation does not say if the module execution is from the first pipeline or the second. I believe that it returns the module from the first pipeline based on executionId, but that should be noted in the Swagger documentation.
4. Yes, the output of the service to compare pipelines should be more informative

Scenario 3

1. Yes.
2. 1 min 10 sec.
3. No.
4. No.

Scenario 4

1. Yes.
2. 1 min.
3. No.
4. No.

Other notes

- It would be nice if comparing pipelines would lead to the actual diff in the module outputs. I mean I do not only want to see that the same module returns different output for two pipelines, but I also want to know what is the diff. There should be at least some sample of different triples returned in the output. Moreover, I would like to see the same comparison with respect to triple count.

- Ordering of the JSON-LD output of services should be done more carefully. It seems like it was done in a random way, but I would prefer to have a direct output of the service at the beginning (i.e. for endpoint /executions I would like to have returned executions at the beginning) and e.g. related-resources at the end of the JSON-LD output
- In related resources there is a missing id of a link, why there is null?

7.1.5 Tester 5

Scenario 1

1. Yes.
2. 10 min (I had to change the docker-compose volumes- because I'm on MacOs)
3. No.
4. -

Scenario 2

1. Yes.
2. 2 min
3. I was confused about what to do, but I guess I found it correctly.
4. I would like to have examples for execution comparison.

Scenario 3

1. Yes.
2. 2 min.
3. No.
4. No, seems good.

Scenario 4

1. Yes.
2. 10 minutes.
3. I put in the wrong pipeline ID.
4. No.

7.2 Test results

- Scenario Completion: the majority of scenarios were completed successfully, proving that functionality of SPipes debug API works as expected.
- Scenario Duration: the completion time varied among the testers, ranging from a few seconds to around 11 minutes. The difference is mainly due to the fact that some testers knew nothing at all about Linked Data and SPipes or encountered unforeseen problems, such as entering the wrong ID or setting up a MacOS startup.
- Suggestions for improvement:
  - Enhancing the testing documentation by providing more detailed examples and explanations, particularly related to JSON-LD structure and usage of HATEOAS.
  - Improving the Swagger UI by adding relevant examples, comments, and clarifications for parameters and endpoints, making it easier for testers to navigate in API.
  - Providing more informative output when comparing pipeline executions, including details about the differences in module outputs and triple counts.
  - Ensuring consistent and logical ordering of related resources in the JSON-LD output.
  - Resolving null ID's in the related resources field.

Chapter 8
Conclusion

The main goal of the thesis was to create debugging API for scripts written in SPipes. To achieve this goal, I studied Semantic Web related technologies such as RDF, SPARQL, SPARQL Motion, and others and described them in the thesis. I also learned how SPipes and similar tools work and explored their debugging capabilities. Based on this knowledge I was able to create scenarios of using the debugging tools and start designing and implementing SPipes debug API.

The result is a new module, s-pipes-debug, created in the SPipes project. It is an independent web module that can be run independently of the SPipes engine. The module provides a REST API that the user can work with without using the UI because the module uses the principles of HATEOAS and allows easy navigation between the endpoints.

The module handles data stored in the RDF4J repository. Based on this data, useful functions have been implemented, such as finding the module where the triple was created or filtered out, comparing pipeline exectuions, finding the module where the variable was created, getting information about how long specific modules were executed and how many input and output triples they have. Implemented tools will help SPipes script developers find bugs faster and help them optimize their scripts.

I hope that in the future my work will be useful for the Faculty of Electrical Engineering at CTU Prague and that it will be a good source of information or software basis for future developments in the area of debugging scripts written in SPipes. I am very grateful to CTU and the Faculty of Electrical Engineering for all the experience and knowledge that I gained during my studies and was able to apply to my bachelor's thesis.

8.1 Recommendations for future work

In my work, I think I have provided a good base for further work with the Debug API, where developers can add more and more new functionality based on the already written logic and architecture. In further work, I would recommend the following:

- Debug API contains endpoint PUT /repository/repositoryName, add

logic, so that it creates a new repository if the repository does not exist.

- Improve logging :
  - The module generates a lot of DEBUG-level logs and does not react on adding of logback.xml file.
  - There is no logging of errors or program progress, which may make it difficult to find errors.
- Cover functionality with tests.
- Implement other debugging tools from activity diagrams from section 4.1.
- Improve the Swagger UI by adding relevant examples, comments, and clarifications for parameters and endpoints, making it easier for testers to navigate in API.
- Providing more informative output when comparing pipeline executions, including details about the differences in module outputs and triple counts.
- Ensuring consistent and logical ordering of related resources in the JSON-LD output.
- Resolving null ID's in the related resources field.

References

1. AGARWAL, Parth R. Semantic Web in Comparison to Web 2.0. In: *2012 Third International Conference on Intelligent Systems Modelling and Simulation*. 2012, pp. 558–563. Available from DOI: 10.1109/ISMS.2012.49.
2. W3C. *What is Linked Data?* [W3C]. 2023-02-22. Available also from: https://www.w3.org/standards/semanticweb/data.
3. TIM BERNERS-LEE. *Linked Data*. 2023-02-22. Available also from: https://www.w3.org/DesignIssues/LinkedData.html.
4. BIZER, Christian; HEATH, Tom; IDEHEN, Kingsley; BERNERS-LEE, Tim. Linked Data on the Web (LDOW2008). In: *Proceedings of the 17th International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 1265–1266. WWW '08. ISBN 978-1-60558-085-2. Available from DOI: 10.1145/1367497.1367760. event-place: Beijing, China.
5. *Linked Open Data Cloud*. [N.d.]. Available also from: https://lod-cloud.net/. Accessed on February 2, 2023.
6. TOM GRUBER. Ontology. 2008. Editors: Ling Liu and M. Tamer Özsu Publisher: Springer-Verlag.
7. NOY, Natalya F.; MCGUINNESS, Deborah L. *Ontology Development 101: A Guide to Creating Your First Ontology*. 2001.
8. GRUBER, Thomas R. A translation approach to portable ontology specifications. *Knowledge Acquisition* [online]. 1993, vol. 5, no. 2, pp. 199–220 [visited on 2023-05-06]. ISSN 1042-8143. Available from DOI: 10.1006/knac.1993.1008.
9. W3C. *RDF 1.1 Concepts*. 2014. Available also from: https://www.w3.org/TR/rdf11-concepts/.
10. W3C. *RDF 1.1 Concepts and Abstract Syntax*. 2014. Available also from: https://www.w3.org/TR/rdf-concepts/.
11. W3C. *RDF 1.1 Primer*. 2014. Available also from: https://www.w3.org/TR/rdf11-primer/.

12. EL IDRISSI, Bouchra; BAÏNA, Salah; MAMOUNY, Anass; ELMAALLAM, Mina. RDF/OWL storage and management in relational database management systems: A comparative study. *Journal of King Saud University - Computer and Information Sciences*. 2022, vol. 34, no. 9, pp. 7604–7620. ISSN 1319-1578. Available from DOI: https://doi.org/10.1016/j.jksuci.2021.08.018.

13. THE JSON-LD COMMUNITY GROUP. *JSON-LD* [online] [visited on 2023-03-16]. Available from: https://json-ld.org/.

14. WIKITRAVEL. *Wikitravel: Turtle RDF* [online] [visited on 2023-03-06]. Available from: https://wikitravel.org/en/Wikitravel:Turtle_RDF.

15. PRUD'HOMMEAUX, Eric; SEABORNE, Andy. SPARQL query language for RDF. 2007.

16. SPARQLMOTION TEAM. *SPARQLMotion* [online] [visited on 2023-03-09]. Available from: https://sparqlmotion.org/.

17. ECLIPSE FOUNDATION, INC. *RDF4J* [online] [visited on 2023-02-15]. Available from: https://rdf4j.org/.

18. ECLIPSE FOUNDATION, INC. *RDF4J Documentation* [online] [visited on 2023-02-15]. Available from: https://rdf4j.org/documentation/.

19. FIELDING, Roy T.; TAYLOR, Richard. *RESTful Web Services*. O'Reilly Media, 2008. ISBN 0596529260, ISBN 9780596529260. Available also from: https://books.google.cz/books?id=RQVu5YN59loC.

20. KELLY, Mike. *HAL - Hypertext Application Language* [online] [visited on 2023-04-30]. Available from: https://stateless.group/hal_specification.html.

21. KNOWLEDGE-BASED SOFTWARE SYSTEMS GROUP. *S-Pipes* [online] [visited on 2023-03-09]. Available from: https://github.com/kbss-cvut/s-pipes.

22. JORDAN, Petr. *Title of the Thesis*. 2021. Available also from: https://dspace.cvut.cz/handle/10467/97070. MA thesis. Czech Technical University in Prague.

23. JAKUB KLÍMEK, Petr Škoda. *LinkedPipes ETL*. Available also from: https://goo.gl/zfcbhg.

24. KLÍMEK, Jakub; ŠKODA, Petr. LinkedPipes ETL in use: practical publication and consumption of linked data. In: 2017, pp. 441–445. Available from DOI: 10.1145/3151759.3151809.

25. TEAM, LinkedPipes. *LinkedPipes documentation* [online] [visited on 2023-05-13]. Available from: https://etl.linkedpipes.com/documentation/.

26. MILLER, Meg; VIELFAURE, Natalie. OpenRefine: An Approachable Open Tool to Clean Research Data. *Geospatial Data and Software Reviews*. 2021. ISSN 2561-2263.

27. WALLSCOPE. *Creating Linked Data* [online]. 2018-05 [visited on 2023-05-14]. Available from: https://medium.com/wallscope/creating-linked-data-31c7dd479a9e.

28. *OpenRefine documentation* [online]. 2023 [visited on 2023-05-14]. Available from: https://openrefine.org/docs/manual/running.

29. BUSINESS ANALYSIS, International Institute of. *A Guide to the Business Analysis Body of Knowledge*. 2nd ed. 2009. ISBN 978-0-9811292-1-1.

30. GUPTA, Lokesh. *What is REST* [online] [visited on 2023-04-15]. Available from: https://restfulapi.net/.

31. *What is REST, API and REST API?* [Online]. 2023 [visited on 2023-04-14]. Available from: https://phpenthusiast.com/blog/what-is-rest-api.

32. FADATARE, Ramesh. *Rest API client-server architecture* [online]. JavaGuides, 2020 [visited on 2023-04-13]. Available from: https://www.javaguides.net/2020/07/three-tier-three-layer-architecture-in-spring-mvc-web-application.html.

33. FARRELL, Joyce. *Java Programming*. 9th. Cengage Learning, 2022. ISBN 9780357673428.

34. SCHILDT, Herbert. *Java: A Beginner's Guide*. 8th. McGraw-Hill Education, 2020. ISBN 9781260440217.

35. MANE, Dashrath; CHITNIS, Ketaki; OJHA, Namrata. The Spring Framework: An Open Source Java Platform for Developing Robust Java Applications. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*. 2013, vol. 3, no. 2, pp. 128–131. ISSN 2278-3075. Available also from: https://www.ijitee.org/wp-content/uploads/papers/v3i2/B2123083213.pdf.

36. *Why Spring?* [Online] [visited on 2023-02-10]. Available from: https://spring.io/why-spring.

37. GROUP, KBS Software Solutions. *JOPA: Java Object Persistence API* [online] [visited on 2023-02-15]. Available from: https://github.com/kbss-cvut/jopa.

38. GROUP, KBS Software Solutions. *JOPA example* [online] [visited on 2023-02-15]. Available from: https://github.com/kbss-cvut/jopa.

39. NICKOLOFF, Jeffrey; KUENZLI, Stephen. *Docker in Action*. Manning Publications, 2016.

40. *What is a container?* [Online] [visited on 2023-04-28]. Available from: https://www.docker.com/resources/what-container/.

41. SPRING FRAMEWORK DOCUMENTATION. *Spring Framework - ResponseBodyAdvice Interface* [online] [visited on 2023-05-17]. Available from: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/ResponseBodyAdvice.html.

Appendix A
Code snippets

A.1 Dockerfile

```
# STAGE MAVEN_BUILD
FROM maven:3.8.6-openjdk-11 AS MAVEN_BUILD

COPY ./ ./

RUN mvn clean package -T 2C -DskipTests -q


# STAGE MODULES_CHECKOUT
FROM alpine/git:v2.32.0 AS MODULES_CHECKOUT

WORKDIR /
RUN git clone --depth 1 https://kbss.felk.cvut.cz/gitblit
    /r/s-pipes-modules.git


# FINAL STAGE
FROM tomcat:9.0-jdk11-corretto

EXPOSE 8080

RUN rm -rf /usr/local/tomcat/webapps/*

COPY --from=MODULES_CHECKOUT ./s-pipes-modules /scripts/s
    -pipes-modules
COPY --from=MAVEN_BUILD /s-pipes-web/target/s-pipes-web
    -*[0-9] /usr/local/tomcat/webapps/s-pipes
COPY --from=MAVEN_BUILD /s-pipes-debug/target/s-pipes-
    debug-*[0-9] /usr/local/tomcat/webapps/s-pipes-debug

CMD ["catalina.sh","run"]
```

Listing A.1: Docker file

A.2 Docker RDF4J

```
FROM eclipse/rdf4j-workbench:3.7.7
USER root
RUN chown -R _apt:root /var/rdf4j/
```

Listing A.2: Docker rdfj4 file

A.3 Docker-compose

```
version: '3.7'

services:
  s-pipes-engine:
    image: ghcr.io/mircheqtm/s-pipes/s-pipes-engine-debug
   :latest
    container_name: s-pipes-engine
    ports:
      - "8081:8080"
    expose:
      - "8080"
    networks:
      - overlay
    depends_on:
      - rdf4j
    environment:
      - CONTEXTS_SCRIPTPATHS=
      - STORAGE_URL=http://rdf4j:8080/rdf4j-server/
   repositories
    volumes:
      - /tmp:/tmp
      - /home:/home
      - /usr/local/tomcat/temp/:/usr/local/tomcat/temp/

  rdf4j:
    build:
      context: .
      dockerfile: Dockerfile_rdfj4
    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 A.3: Docker compose file

Appendix B

Evaluation of framework for debugging SPipes

The goal of this evaluation is to test the user experience with the new SPipes debug API which allows you to use new debugging tools in the SPipes semantic pipelines.

This document contains the basic information the tester should know before starting the testing process and 4 simple testing scenarios. Please, study the information for testing, go through the scenarios, and answer the questions for each scenario. Leave your answers at the end of the document, use the answer template that is located under the last testing scenario.

B.1 Run SPipes with debug module

- Clone the repository from https://github.com/mircheqtm/s-pipes. This fork contains changes with SPipes debugging tools.

- Go to s-pipes/s-pipes-debug/doc/hands-on-tutorial/ and run command docker-compose up –build.

- Strictly recommend to download some json formatter for your browser, as it will be hard to orient with lots of data. (Personal recommendation is JsonDiscovery [1])

[1] JsonDiscovery extension URL - https://chrome.google.com/webstore/detail/jsondiscovery/pamhglogfolfbmlpnenhpeholpnlcclo

B.2 URL's

Name of service	Address
Spipes core engine	localhost:8081/s-pipes
Sipes debug engine	localhost:8081/s-pipes-debug
RDF4J server	localhost:8080/rdf4j-server
RDF4J workbench	localhost:8080/rdf4j-workbench

Table B.1: Table of services

For list of SPipes Debug API endpoints, check out swagger API, available on `http://localhost:8081/s-pipes-debug/swagger-ui.html#`

B.3 Related resources

The SPipes debug API implements HATEOAS technology. HATEOAS allows you to find other endpoints associated with the entity that was returned to the user.

When using the SPipes debug API, pay attention to the has_related_resources field, which describes all possible endpoints related to the returned entity, their purpose and sometimes possible parameters that can be used in the request.

B.4 Questions

For every scenario please answer the following questions:

- Did you finish the scenario?
- How much did it take to complete the whole scenario?
- Did you have any problems during scenario completion? If yes, what went wrong?
- Is there anything you want to improve?

B.5 Scenarios

B.5.1 Precondition

- Scenario will be based on the hello world example, directly from the SPipes project. You can find all information about the script here[2].

[2] `https://github.com/kbss-cvut/s-pipes/blob/main/doc/examples/hello-world/hello-world.md`

Please make sure that you have a basic understanding of how the hello world example works.

- SPipes debug API interacts mostly with two main entities ModuleExecution and PipelineExecution. In main README.md [3] there is a description of what are Pipelines and Modules. So basically ModuleExecution and PipelineExecution are entities representing data about executed Module or Pipeline. It can contain such data as, where are stored output of modules, time, when execution happened, and a lot of different useful information.
- Read Swagger API [4].

B.5.2 Scenario 1

- Run the hello world example script with the following URL: `http://localhost:8081/s-pipes/service?_pId=execute-greeting&firstName=TestName&lastName=TestSurname` You should be able to see the greeting message "Hello TestName TestSurname." as part of the output JSON-LD.
- Check following link to RDF4J workbench: `http://localhost:8080/rdf4j-workbench/repositories/s-pipes-hello-world/summary` Number of statements should not be 0. If it's so, we can continue. Otherwise, you messed something up at step 1.
- Using the debug API [5], try to look through all pipeline executions that were executed.
- If you see one execution, then the scenario is successfully passed.

B.5.3 Scenario 2

- Let's change a bit our script: open /s-pipes/doc/examples/hello-world/hello-world.sms.ttl and change

  BIND(concat("Hello ", ?personName, ".") as ?greetingMessage)

  to

  BIND(concat("Hello ", ?personName, "!") as ?greetingMessage)
- Run the script one more time in the same way as scenario 1 (Step 1) B.5.2.
- Using the debug API, try to look through all pipeline executions that were executed.

[3] `https://github.com/mircheqtm/s-pipes/blob/main/README.md`
[4] `http://localhost:8081/s-pipes-debug/swagger-ui.html#`
[5] `http://localhost:8081/s-pipes-debug/swagger-ui.html#`

- Now from the response you should get 2 pipeline executions.
- We changed our script, let's see, what is the difference between two pipeline executions. Try to find, in which module, the first difference between pipeline executions was found.

B.5.4 Scenario 3

- We already have some pipeline executions, so we don't need to run anything else.
- Using the debug API, try to display all executed modules in any of your pipeline executions.
- Sort modules by duration and find which module was the slowest.
- Sort modules by a count of output triple and find which module produced the biggest amount of tiples.

B.5.5 Scenario 4

- When we executed the first execution, the response looked similar to this B.1:

```
{
@id: "http://onto.fel.cvut.cz/ontologies/s-pipes/hello
    -world-example-0.1/testname-testsurname",
is-greeted-by-message: "Hello TestName TestSurname.",
@context: {
is-greeted-by-message: { @id:"http://onto.fel.cvut.cz/
    ontologies/s-pipes/hello-world-example-0.1/is-
    greeted-by-message"
```

Listing B.1: Response from pipeline execution.

  Its RDF representation looks like this: <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world-example-0.1/testname-testsurname> <http://onto.fel.cvut.cz/ontologies/s-pipes/hello-world-example-0.1/is-greeted-by-message> "Hello TestName TestSurname.

- Using the debug API, try to find which module produced the tiple from the previous step.
- Using the debug API, find out which module first produced the variable lastName.

Appendix C

Endpoints

- **Change repository**
  PUT /repository/{repositoryName}
  Changes the repository of RDF4J database.
  - **Examples**
    - /repository/s-pipes-hello-world
    - /repository/s-pipes-skosify
- **Get all pipeline executions in the repository**
  GET /executions
  Returns brief info about the most recent executions in the repository. Latest on top.
- **Get pipeline execution by ID**
  GET /executions/{executionId}
  Returns info about execution and short information about modules, executed in this pipeline execution.
- **Get module executions, executed in the pipeline execution**
  GET /executions/{executionId}/modules
  Optional parameters
  - Parameter orderBy:
    - duration
    - start-time
    - output-triples
    - input-triples
  - Parameter orderType:
    - ASC
    - DESC

  Returns modules, for given pipeline execution and complete information about the modules. Possible to sort them by duration, count of output triples, count of input triples or start time in given order ASC or DESC.

- **Compare pipeline executions and find first difference**
  GET /executions/{executionId}/compare/{executionToCompareId}
  Compare the result of two pipeline executions. If they are not the same field, are_same will return false, and the module, where the first difference was found will be returned. The result is saved to RDF4J db, so it will not recompute everything and just return the result from DB.

- **Find module execution, where triple was created**
  GET /triple-origin/{executionId}
  Mandatory parameters:

  - graphPattern=<http://some.subject> <http://some.predicate> <http://some.object>

  The browser will transfer all symbols like " or whitespaces to the encoded version and translate them back on the back end. Returns module, where the triple was first produced as the output of the module. In case more modules have it as the output on the same depth of the execution tree, multiple modules will be returned.

- **Find module execution, where triple was filtered out**
  GET /triple-elimination/{executionId}
  Mandatory parameters:

  - graphPattern=<http://some.object> <http://some.predicate> <http://some.object>

  The browser will transfer all symbols like " or whitespaces to the encoded version and translate them back on the back end. Returns module, where the triple was first passed as input, but was eliminated on output. In case more modules have it as input, but don't have an output on the same depth of the execution tree, multiple modules will be returned.

- **Find module execution, where the variable was created**
  GET /variable-origin/{executionId}
  Mandatory parameters:

  - variable=variableName

  The browser will transfer all symbols like " or whitespaces to the encoded version and translate them back on the back end. Returns module, where the variable was created. In case more modules created the same variable on the same depth of the execution tree, multiple modules will be returned.