Chemnitz University of Technology
Faculty XXX
Chair XXX

faculty

chair

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Diploma Thesis

zur Erlangung des akademischen Grades

Web Engineering

vorgelegt von

Wesley Giovanni Obi

Development of IDE extensions for artificial, schema-driven generation and visualization of RDF A-Box Resources

Reviewer: XXX
XXX
Advisor: XXX

advisor

Chemnitz, date

date

Summary

The Resource Description Framework (RDF) is a standard for describing resources on the web. RDF extends the Web's linking structure by using URIs to name the relationships between resources and the two ends of the link. It is designed for machine readability rather than human readability.

Developers often struggle with the time-consuming and complex task of manually generating example instances for new RDF schemas, which can lower their productivity and the seamless integration of these schemas into software projects.

This thesis aims to improve the user-friendliness visualization of RDF schema, by using a code editor extension that allows not just to better visualize the schema and the relations within but also provides automatically generated example instances for a better understanding.

The main challenge for this project is the User Interface Design. RDF data is structured as a graph of triples (subject-predicate-object), which can be difficult to represent visually. Unlike traditional tabular data, RDF's graph-based nature requires a UI that can effectively display nodes and their relationships. RDF schemas can vary widely in structure and content. Designing a UI that can dynamically adapt to different schemas and data types while remaining clear is a significant challenge. The UI should handle many types of vocabularies and ontologies.

Addressing this challenge will require regular software/web development knowledge but also a deep understanding of the Semantic web and its technologies, like RDFs and SPARQL.

Task

Titel

Replace or remove title

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

to adjust

Contents

Chapter 1

Introduction

The World Wide Web (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents [16].
The Web works thanks to a set of standards and protocols that guarantee interoperability at various levels. It is designed for human interaction, but the next generation web aims to make the Web understandable by machines: The Semantic Web [14].

As described in *Scientific American*, *"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation"* [1].

The Hyper Text Markup Language (HTML) is the standard markup language used for structuring and displaying documents on the Web. While this language is great for human users, it is not optimized for structure data processing [3].
Consider the following Airline Scraper example:

```
<div class="flight-result">
    <h2>Berlin to New York</h2>
    <p>Airline: Lufthansa</p>
    <p>Price: $450</p>
    <p>Departure: 10:00 AM</p>
</div>

```

Listing 1.1: Example of HTML flight data from an Airline company

In this scenario, the labels of fields like "Price", "Departure", etc., are not standardized. If

the Airline changes any of those labels, the data extractor algorithm breaks. Developers would need to monitor every single Airline website and update the data extractor algorithm as soon as possible in order to guarantee the availability of their scraping service [3].

It is now clear why the Semantic Web needs its own standard to describe web resources that need to be process by machines.

The Resource Description Framework (RDF) is a standard model for data exchange on the Web. It has features that facilitate the merging of data with different schemas, and it supports the evolution of schemas over time without requiring all the data consumers to be changed [10].
RDF enhances the linking structure of the Web to use URIs to label both the relationships between entities and the entities themselves (aka "triple"). This model allows for structured and semi-structured data to be mixed, exposed, and shared among different applications. This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view provides an accessible mental model for understanding RDF and it's often used with its visual explanations

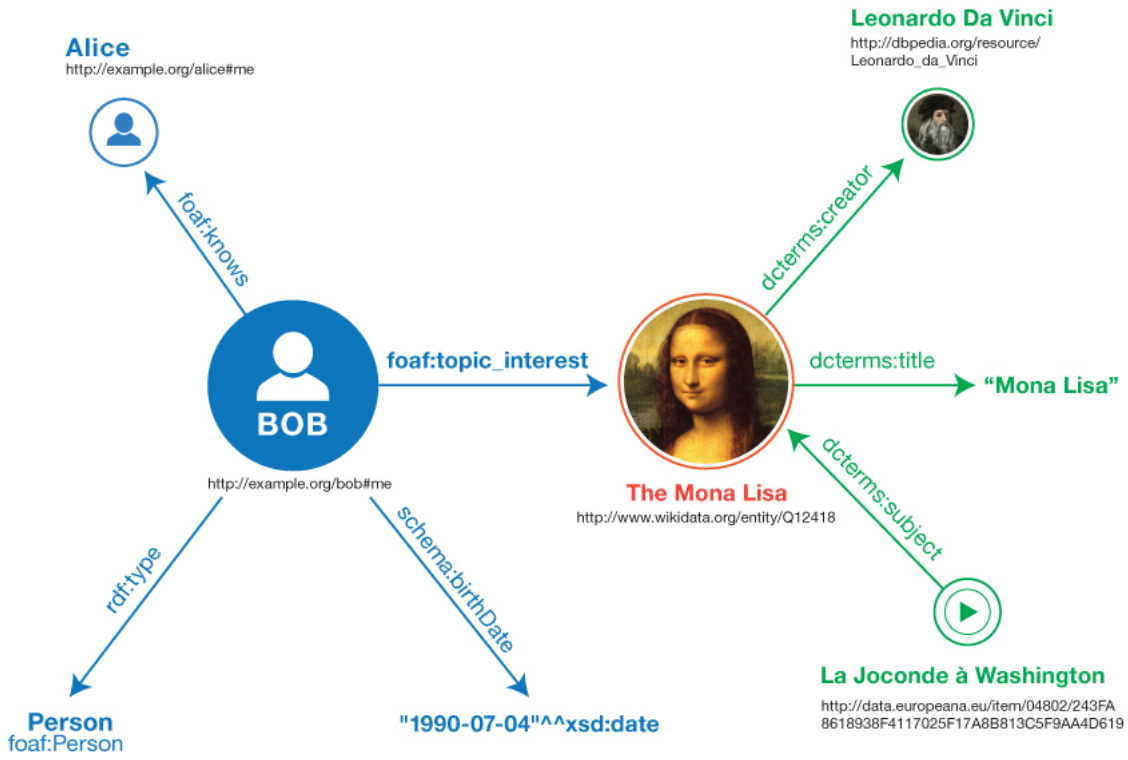


Figure 1.1: Example of RDF Graph

Consider the following Airline Scraper RDF example:

```
@prefix schema: <https://schema.org/> .

<https://example.com/flights/12345> a schema:Flight;
    schema:departureAirport "Berlin" ;
    schema:arrivalAirport "New York" ;
    schema:airline "Lufthansa" ;
    schema:price "450"^^xsd:decimal ;
    schema:departureTime "10:00 AM"^^xsd:time .

```

Listing 1.2: RDF representation of the flight data

In this scenario scenario, the Airline company has decided to adopt the RDF standard to describe their data. Developers can now query the Airline web service without worrying about breaks due to changes in the Airline company [3].

The RDF standard is not exclusively used by web developers. It is also widely utilized by scientists, engineers, and researchers across various disciplines:

- **Data Scientists**:
  - RDF aligns with their work on structured data, semantic queries, and knowledge graphs.
  - They use RDF to model relationships, extract insights from interconnected datasets, and support machine learning on graph data.
- **Computer Scientists and Engineers**:
  - They focus on RDF's theoretical foundations, optimization of storage and querying, and application development for the Semantic Web.
  - They are often involved in creating RDF tools like RDF stores and query languages.
- **Knowledge Engineers**:
  - Specialize in designing ontologies and frameworks using RDF to represent knowledge in areas like AI, natural language processing, and expert systems.

To assist Scientists in their work, the Semantic Web supports a set of tools and technologies that enable the creation, sharing, and analysis of structured data. Platforms like Protégé, Apollo, and NeOn are well-known in the ontology engineering community.

"An Ontology is a knowledge structure used to formally represent and share domain

knowledge through the modeling and creation of a framework of relevant concepts and the semantic relationships between the concepts" [11].

These ontology editors offer comprehensive support for creating and editing ontologies and RDF datasets through their graphical user interfaces (GUIs), which enhance visualization and make RDF development easier.

1.1 Motivation Scenario

The following hypothetical scenario illustrates the current situation.
John Doe is a Junior Software Developer and just started working in his first company as Web Developer. He quickly immersed himself in his daily tasks, working with his favorite IDE, Visual Studio Code, to craft the backbone of their innovative web application—a platform with social media-like features.

While John primarily worked with VSCode due to its lightweight and customizable nature, he soon realized that his workflow differed significantly from that of his colleagues. Some teammates preferred IntelliJ IDEA for its extensive refactoring tools, while others relied on Protégé because its visual representation of ontologies helped them better understand and manage complex data relationships.

A significant part of the project involved working with RDF files, mainly RDF/XML and Turtle formats, to define the application's semantic data structure and its RDF schema. Although the project repository already contained some instances examples, John found himself losing a lot of time creating a valuable amount of example instances to test the limitation of his RDF schema. This extra step, required to ensure that every modification produced the intended results, often disrupted his coding flow.

Although he is writing in Turtle language for the majority of his time and Turtle is the most readable among all the RDF formats, John also struggles to read and understand many lines of code and often mistake like every other human being. Just like his colleagues, he started using an RDF schema visualizer like "isSemantic.net" or RDF Grapher, to see a graphical representation of his RDF schema [4, 6]. The representation helps him a lot to identify relations errors, but importing his schema in another software every time contributes to the interruption of his workflow.

1.2 Theses and Scientific Contribution

The previous scenario highlights two big challenges. The first is the lack of a seamless integration between specialized development environments and RDF visualization tools.

Developers should be free to choose the IDE that best suits them, but the disjointed process of visualizing semantic data remains a critical bottleneck.

The second challenge lies in the inefficiency of testing and validating RDF schemas. Manually generating new instances to test schema's limitations remains a major time-consuming task. Developers must invest considerable effort in creating and maintaining test data, moving their focus from core development activities. This repetitive manual work not only delays progress but also increases the risk of inconsistencies and undetected schema flaws.

This Master's Thesis aims to address these challenges and tries to incorporate all developer tools into a single environment. By developing an IDE extension, developers will be able to handle RDF writing, data visualization, and instance generation in one place. This IDE extension could support the direct editing of the generated instances, allowing for a more streamlined and efficient workflow.

1.3 Positioning within the Scientific Context

This Thesis is positioned in the domain of Semantic Web and RDFs. It tries to address in the integration of development environments, RDF visualization and RDF instances generation, to enhance both theoretical understanding and practical application in the field.

As stated in the previous sections, the Semantic Web and RDFs, are very important for the scientific community. Their goal is to simplify machines' and computer scientists' lives when extracting and processing a vast amount of information from the W3. Despite all its advantages, the Semantic Web struggles to move beyond academic use and find widespread commercial applications. There is indeed a lack of tools that can help engineers to work with RDFs and the Semantic Web to facilitate the creation and management of RDF Data. Among the most popular Integrated Development Environments, like Visual Studio Code, IntelliJ IDEA, and others, we can mainly find extensions that support syntax highlighting, with grammar checker and SHACL validators, and only one VSCode extension for the visualization of RDFs limited to the N3 and Turtle formats [15, 5, 13] (see Figure 1.2).

The Semantic Web community needs a solution that could bridge this gap, something that is not limited to a specific software, architecture, or environment. A solution that could potentially be compatible with every IDE and Browser. An open-source software with and MIT license can be enhanced not just by the community, but also by companies and institutions. A program with a core that allows an easy integration of modules and plugins.

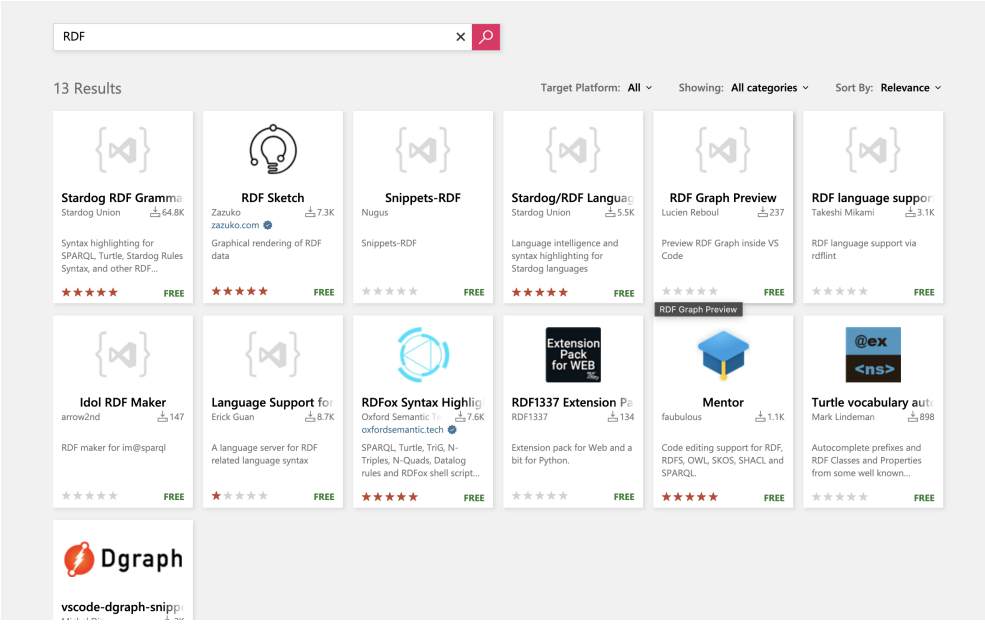


Figure 1.2: Limited Support: A Search for 'RDF' in the Plugin Marketplace Yields Only 13 Results, Highlighting the Scarcity of RDF-Specific Extensions for Developers.

For this Master's Thesis, I will develop a Web Service using Python 3.11 and the Fastapi library, able to process the most common RDF formats, generate n amount of instances for a given RDF Schema, and return the desired results. The outcome will be provided in different formats, but mainly served as HTML pages, so that can be processed by a Visual Studio Code extension in a Web View. This approach will limit the service to be used only by the VSCode IDE, but by all the ones that support rendering of HTML content in their environment.

To prove the validity of this approach, I will develop not just the Visual Studio Code extension using TypeScript, but also a dedicated browser frontend view, and an IntelliJ IDEA plug-in, in Kotlin, with a limited set of features. These Client Side applications will communicate with the Web Service, and via REST calls. The user will be able to:

- Send an RDF Schema to the Web Service
- Request an n amount of instances for the given schema
- Edit the RDF Schema with the instances generated by the server
- Visualize the RDF
- Download the Graph as an image.

Due to the vastness of this project, some features will only be deepened in a theoretical

manner, like the AI Instance Generation, and only some because features for the IntelliJ IDEA plugin will be covered.

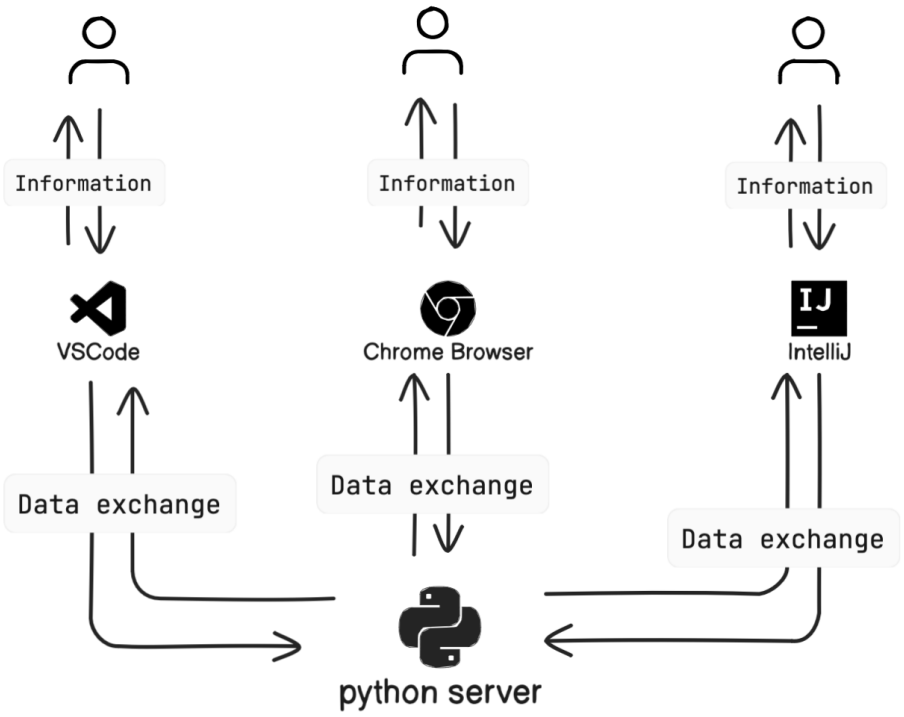


Figure 1.3: System Architecture: Information Flow and Data Exchange between Users, Development Environments (VSCode, IntelliJ), Chrome Browser, and a Python Server.

1.4 Structure of the Work

This thesis is separated into 6 chapters.

Chapter 2: State of the Art covers the related work in the RDF world, from the instance generation to the graph visualization.
A short view of existing tools and projects in the same domain, like GAIA and Ontodia.
A comparison of the two different approaches in terms of instance creation (Manual vs Automatic). A brief look at the current standards and technologies compatible with the semantic Web and RDFs.

Chapter 3: Concept includes the requirements for the project, a view of the technologies and high-level architecture, and the constraints and goals of this work.

Chapter 4: Implementation features the implementation of the project with all its components. Describe in very detail the design choices, building blocks, and algorithms of this system.

Chapter 5: Evaluation explains how the implementation is validated. Portrays benchmarks and measurements regarding the performance of the solution. Showcases some examples of RDF generation and graph visualization. Cite the experience of a group of Computer Scientists.

Chapter 6: Conclusion summarizes the thesis, describes the problems that occurred and the work that can be done in the future.

Chapter 2

State of the Art

This chapter is intended to give an introduction about relevant terms, technologies and standards in the field of Semantic Web. It also delves into similar and related implementations (i.e. GAIA) to provide a comprehensive understanding of the current state of the art.

2.1 Technologies

This section covers the fundamental technologies, specifically RDF and SPARQL, which serve as the foundational building blocks for structuring and querying data.

2.1.1 RDF (Resource Description Framework)

The RDF is the foundation of the Semantic Web. It's a framework for describing resources on the Web thanks to URIs. A resource could be anything: a human being, a document, an object a concept [10].

Specifically, RDF can be used to share and interlinked data. For example, if the following URI **http://www.example.org/bob#me** is retrieved, it can provide information about Bob, such as his name, his age, and his friend. If his friends' International Resource Identifiers (IRIs) are retrieved, like Alice's, more information regarding them can be accessed (i.e. more friends, interests, etc.). This process of "link navigation" is called Linked Data [10].

This RDF property can be useful for many use cases like:

- Creating distributed social networks by interlinking RDF descriptions of users.

- Integrating API feeds to ensure seamless discovery of additional data and resources by clients.
- Embedding machine-readable data into web pages.
- Standardizing data exchange across databases.

RDF interlinks resource with "statements". A statement is a triple containing a subject, a predicate, and an object. The subject and the object stand for the resources that need to be represented. The predicate is the type of relation between those resources, and it is always phrased in one directional way (from the subject to the object).
This relation between the two is called a Property [10].
It is possible to visualize these Triples as Graphs (see Figure 1.1) and query them using SPARQL.

In an RDF file, three type of data can occur :IRIs, literals and blank nodes.

- IRIs are the identifiers of the resources. They are similar to the Uniform Resource Locators (URLs), but they don't provide information about where the resource is located or how to access it. They can only be used as mere identifiers. IRIs can appear in all three positions of a triple.
- Literals are all the values that are not IRIs. They can be strings, numbers, dates, etc. They can only appear in the object position of a triple.
- Blank Nodes are all the nodes of a graph that are not identified by an IRI. They are like simple variables in algebra that represent something without saying what their value is. Blank nodes appear in the subject and object position of a triple.

RDF allows grouping multiple RDF statements into multiple graphs and associate them with a single URI.

```
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>.
```

Listing 2.1: RDF Grouped Data

One of the related problems with RDF is that the data model doesn't make assumptions about what resource URIs stand for. For example the statement **ex:Apple ex:isLocated ex:California**, without any additional information about Apple, could be misleading. Without additional context Apple could refer to a fruit in California or Apple incorporated in Cupertino.
One solution to this problem is to use IRIs in combinations with Vocabularies and other

conventions that add semantic information about the resources.

In order to include Vocabularies in an RDF graph, RDF provides a Schema language. The RDF Schema allows the description of groups of related resources and the relations between them. The class and property system is close to an object-oriented programming language. The difference with this model is that the RDF schema defines the properties in terms of Class and not vice versa as in object-oriented programming.

Construct	**Syntactic form**
Class (a class)	C `rdf:type rdfs:Class`
Property (a class)	P `rdf:type rdf:Property`
type (a property)	I `rdf:type C`
subClassOf (a property)	C1 `rdfs:subClassOf C2`
subPropertyOf (a property)	P1 `rdfs:subPropertyOf P2`
domain (a property)	P `rdfs:domain C`
range (a property)	P `rdfs:range C`

Table 2.1: RDF Schema Constructs

An RDF Class is a group of resource with common characteristics. The resources in a class are referred to as instances of that class.

As discussed in previously, properties are used to describe the relations between subject and object resources. The main properties for the construct of RDF schema are:

- **subClassOf** is used to state that all the instances of one class are instance of another.
- **subPropertyOf** is used to define that all resources related by one property are also related by another.
- **domain** is used to declare to which domain / class that property belong to.
- **range** is used to indicate the type of the property value.

RDF supports 4 main types of formats: The "Turtle family of RDFs", JSON-LD, RDF/XML and RDFa [10].

2.1.2 SPARQL

In the RDF world, there are 2 main ways to retrieve data from an RDF store. The first one is to simple is to use a REST Endpoint and perform CRUD operations on the DB. This is in many cases the quickest solution, because it doesn't required new knowledge or new knowhow. The main problem is when the RDF store doesn't allow HTTP CRUD or doesn't provide a REST Endpoint. In these cases the second solution is to use SPARQL.

SPARQL is a set of specifications that provide tools to retrieve and manipulate RDF graph content on the W3 or in an RDF store. One of this tool is the SPARQL Query Language [12].

The SPARQL Query Language it's a query language, not far from the most well known SQL, is used for query formulation and retrieval of on the web.

```
1    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2    SELECT ?name (COUNT(?friend) AS ?count)
3    WHERE {
4        ?person foaf:name ?name .
5        ?person foaf:knows ?friend .
6    } GROUP BY ?person ?name
```

Listing 2.2: Example of SPARQL Query

In order to make the exchange of query results among machines, SPARQL supports the following exchange formats: XML, JSON, CSV and TSV [12].

2.2 Related Work

This section provides a detailed overview and analysis of the related work.

2.2.1 GAIA (Automatic Instance Generator for Abox)

The GAIA project, is a RDF triple generation tool developed by CEDAR. This research introduces an OWL-based generic RDF triple generator capable of loading any OWL ontology schema and generating compliant RDF instances, unlike the LUBM instance generator, which only supports its predefined LUBM ontology. It strictly relies on OWL because the majority of ontologies that are currently available are written using this language [7].

GAIA is designed for performance and scalability, thanks to the multithreading implementation that allows to generate a large volume of triples, making it perfect for ontology benchmarking or synthetic dataset generation [7].

Despite its performance and accuracy, GAIA has several limitations. The software is missing a schema visualization system or interactive graph explorer. This makes it hard for a human to visualize at glance the newly generated instances. The system is also not limited to machines with JAVA installed on it, and having it implemented as IDE extension

is a very challenging task due to its architecture.
Furthermore, GAIA does not natively support RDF Schema (RDFS) or lightweight RDF graphs in Turtle or JSON-LD format (it only supports $.RDF$ or $.XML$).

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:ex="http://example.org/"
         xml:base="http://example.org/ontology"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema#">

    <owl:Ontology rdf:about="http://example.org/ontology">
        <owl:versionInfo>1.0</owl:versionInfo>
    </owl:Ontology>

    <rdf:Description rdf:about="http://example.org/Class/Person">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
        <rdfs:label>Person</rdfs:label>
    </rdf:Description>

    <rdf:Description rdf:about="http://example.org/Class/Organization">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
        <rdfs:label>Organization</rdfs:label>
    </rdf:Description>

    <rdf:Description rdf:about="http://example.org/Individual/JohnDoe">
        <rdf:type rdf:resource="http://example.org/Class/Person"/>
        <ex:name>John Doe</ex:name>
    </rdf:Description>

    <rdf:Description rdf:about="http://example.org/Individual/AcmeCorp">
        <rdf:type rdf:resource="http://example.org/Class/Organization"/>
        <ex:name>Acme Corporation</ex:name>
    </rdf:Description>

</rdf:RDF>
```

Listing 2.3: Example of RDF/XML Data

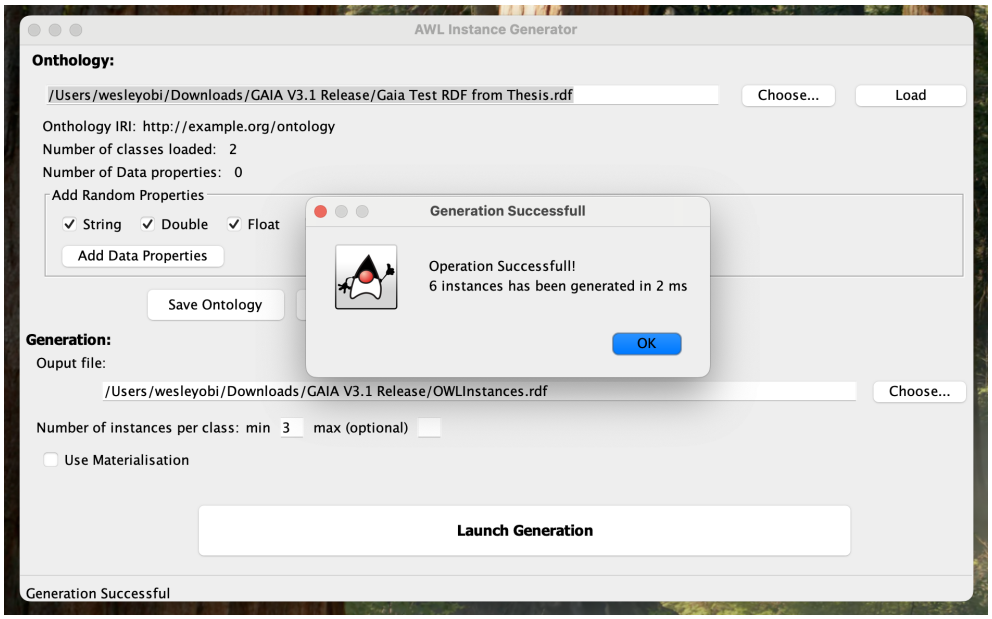


Figure 2.1: GAIA Instance Generator used via GUI

```
...
<!-- Instances Generated by OWL_GENERATOR -->

<owl:NamedIndividual rdf:about="http://example.org/ontology#
Organization_instance0">
    <rdf:type rdf:resource="http://example.org/ontology#Organization"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="http://example.org/ontology#
Organization_instance1">
    <rdf:type rdf:resource="http://example.org/ontology#Organization"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="http://example.org/ontology#
Organization_instance2">
    <rdf:type rdf:resource="http://example.org/ontology#Organization"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="http://example.org/ontology#
Person_instance0">
    <rdf:type rdf:resource="http://example.org/ontology#Person"/>
</owl:NamedIndividual>

<owl:NamedIndividual rdf:about="http://example.org/ontology#
Person_instance1">
    <rdf:type rdf:resource="http://example.org/ontology#Person"/>
</owl:NamedIndividual>
```

```
23
24      <owl:NamedIndividual rdf:about="http://example.org/ontology#
        Person_instance2">
25          <rdf:type rdf:resource="http://example.org/ontology#Person"/>
26      </owl:NamedIndividual>
27
28      </rdf:RDF>
```

Listing 2.4: Results of RDF/XML generated instances

2.2.2 The instance Generator project suggested by the professor

...

2.2.3 RDF Graph Preview

The Graph RDF Graph Preview it's an open source project developed by Lucien Reboul [9]. It's a VSCode extension that allows to load RDF files and visualize them as graphs. It uses the D3.js library as core graph rendering system.

One advantage of this extension is that it allows to quickly visualize RDF graph without leaving the current workspace.
But unfortunately, this plugin is only able to render RDF files written Turtle and N-Triples/N3 formats. The tool provides no support for automatic instance generation, forcing users to do all the heavy work manually.

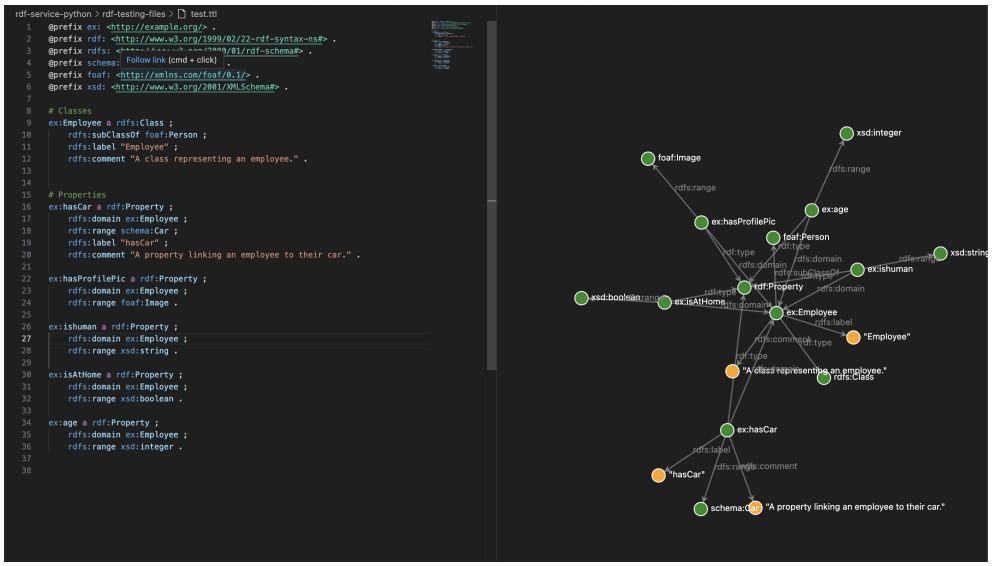


Figure 2.2: RDF Graph Preview VSCode Extension

Chapter 3

Concept

This chapter outlines the requirements and design choices necessary for developing the RDF Instance Generator Schema Visualizer and the RDF web service.

3.1 Overview

This project aims to simplify RDF schema exploration, validation, and instance generation. To achieve this, the following categories of requirements have been defined:

- Functional requirements: Core features such as instance generation, schema validation, and visualization.
- Non-functional requirements: Considerations related to performance, usability, and cross-platform compatibility.
- Technical requirements: Specifications for the underlying technologies and system design.
- Data privacy and security requirements: Measures to ensure the protection and confidentiality of user data.

The design prioritizes a client-server architecture and cross-platform compatibility to enhance usability and efficiency.
The web service is designed to be compatible with various browsers and IDEs.
The IDE extension will support users and seamlessly integrate into their workflow.

3.2 Functional requirements

This section covers the functional requirements for the entire system.

3.2.1 Instance Generation

The first requirement for this project is the generation of synthetic RDF instances.
When a user is writing an RDF file—in Turtle, N3, XML, or other RDF formats—the system will generate synthetic RDF instances to save time by avoiding manual insertion of test data.
The application will not only generate compatible RDF triples but will also allow users to modify the automatically generated instances.

3.2.2 Schema Validation

The program should be designed to validate a user-defined RDF schema by checking the structural integrity and semantic consistency of its triples. Each triple should conform to syntactic constraints (e.g., proper use of IRIs, literals, and blank nodes), and the asserted relationships should not lead to logical contradictions or unintended inferences.
If errors are found in the schema description, feedback should be provided to the user via the UI or terminal.

3.2.3 Multi-Vocabulary Support

Interoperability and semantic richness are crucial for ensuring seamless data integration, enhancing knowledge representation, and enabling efficient querying and reasoning across diverse RDF datasets. By supporting multiple vocabularies and ontologies—such as RDFS, FOAF, Schema.org, and others—the system should be able to handle a wide range of RDF data sources and visualize them as graphs.

3.2.4 Import and Export

The software should allow users to import their custom RDF files and export the files with automatically generated instances in the same format.

3.2.5 Schema Visualization

To simplify the exploration and analysis of RDF data from a human perspective, the program should display the RDF schema as a graphical representation. The displayed graph should support interactive features such as zooming, panning, and collapsing nodes. The graphical representation should adapt based on schema complexity and hierarchical structure.
In the IDE extension, the user should be able to visualize the graph in a side panel window next to the RDF file they are working on.

3.3 Non-functional requirements

This section defines the non-functional requirements of the system.

3.3.1 Performance

The system should ensure high-performance rendering of large RDF graphs to support smooth visualization and interaction. Additionally, actions such as node selection and zooming should occur with minimal latency to maintain a responsive user experience. The system should optimize its rendering pipeline to enable real-time exploration of complex RDF schemas without sacrificing performance.

3.3.2 Usability

Both the IDE extension and the web application should feature an intuitive, user-centric interface that enhances accessibility and minimizes the learning curve. To achieve this, the design should incorporate familiar UI elements to ensure a seamless and user-friendly experience.

3.3.3 Compatibility

To ensure smooth integration across different development environments, the web service should be designed for cross-platform compatibility, allowing easy adoption in various IDEs with minimal adjustments. This enhances interoperability, enabling the service to operate consistently across software ecosystems.
To validate this, minor features should also be implemented in a secondary IDE.

3.4 Technical Requirements

3.4.1 Web Architecture

The system should follow a client-server architecture. While there are no constraints on the specific communication technologies (e.g., SOAP, REST, gRPC), the architecture must follow web principles and use web technologies—such as HTML, CSS, JavaScript, or WebAssembly—and protocols such as HTTP and WebSockets.

3.4.2 IDE extension

The application should be accessible via a web-based client to ensure wide availability. However, the main emphasis should be placed on integration as an extension within widely used IDEs.

3.5 Data privacy and Security

3.5.1 Security

To ensure the confidentiality and integrity of RDF data, the system should use secure data transmission protocols, such as HTTPS, to protect sensitive information during transfer.

3.5.2 Confidentiality

According to the current design, no data should be stored from the business logic layer. All data processing will occur in-memory, either on the client-side or server-side, ensuring that user data is never persistently stored. This ephemeral data handling strategy significantly reduces the risk of data retention and unauthorized access. Consequently, it minimizes compliance obligations related to long-term data storage, such as those outlined in the General Data Protection Regulation (GDPR).

3.6 Solution strategy

As stated in the requirements section, the system uses a client-server architecture where the client and server serve distinct roles in the overall functionality.

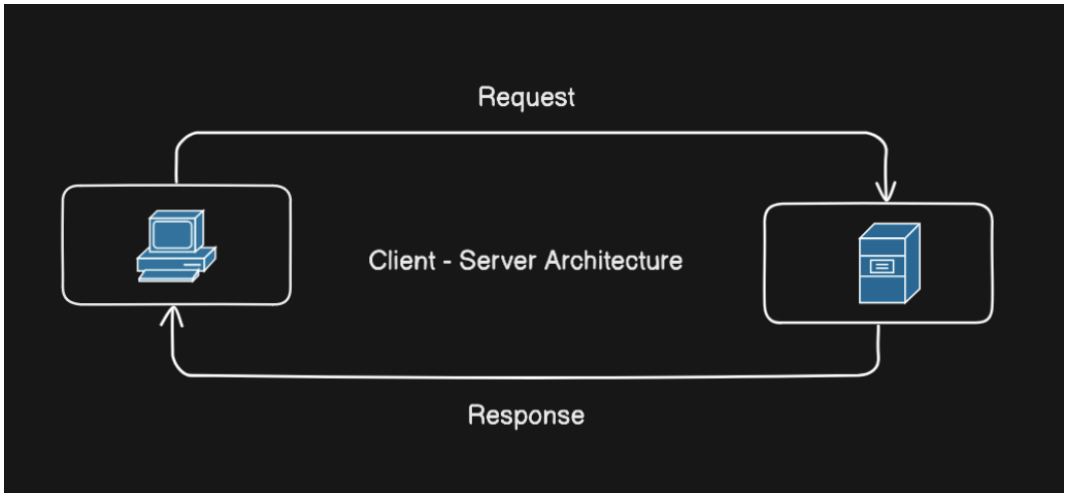


Figure 3.1: Client / Server

3.6.1 Client

To ensure broad system availability, the system is designed to support two types of user agents:

- Browsers

- Integrated Development Environments

The browser selected by the user should access the web application via the HTTP protocol. The user will then upload their RDF file and send it to the server. The client will receive the processed information and display the resulting graph (see Figure 3.2).

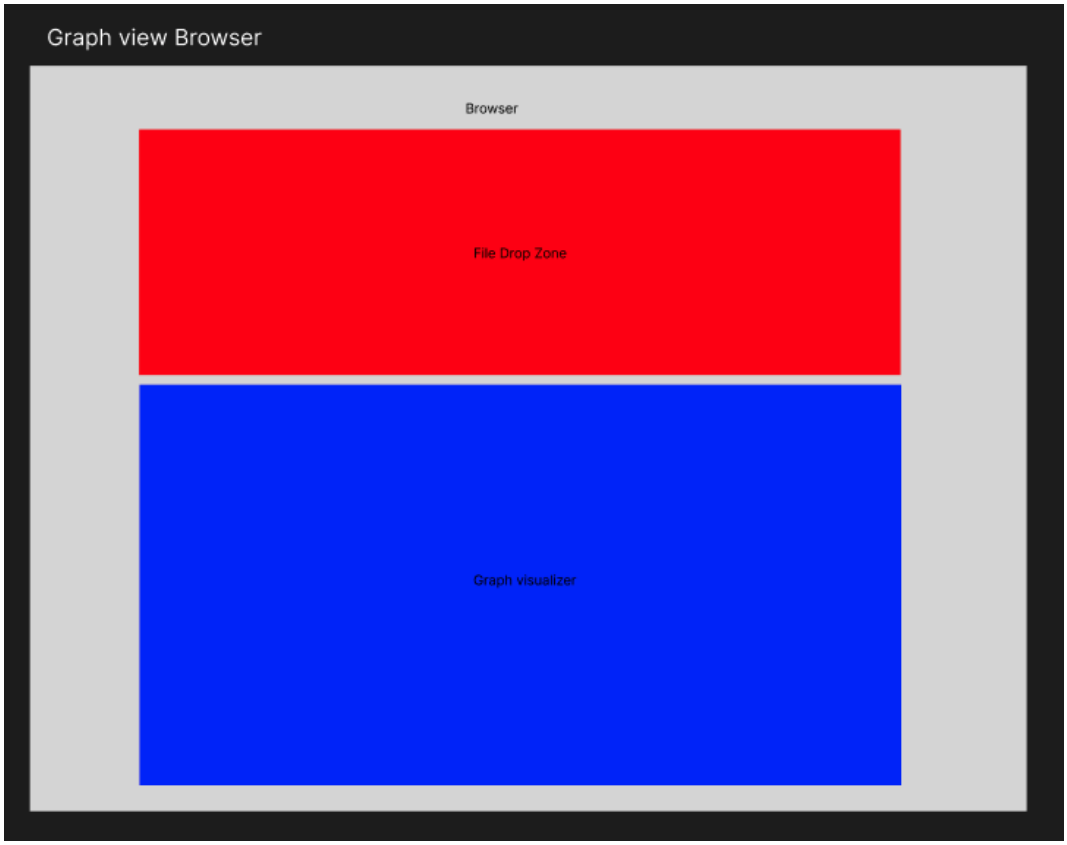


Figure 3.2: Browser file drop zone and graph view

Additionally, the user should be able to view the RDF file alongside the generated instances and its graphical representation (see Figure 3.3).

Within the IDE, the user experience will be slightly different from the browser. Instead of uploading an RDF file, users will interact with the system directly in the IDE. While working on an RDF file and executing a predefined command, a web-based visualization panel will dynamically open beside the file, rendering an interactive graphical view of the RDF schema in real time. This allows users to examine relationships, structures, and dependencies without leaving their coding environment (see Figure 3.4).

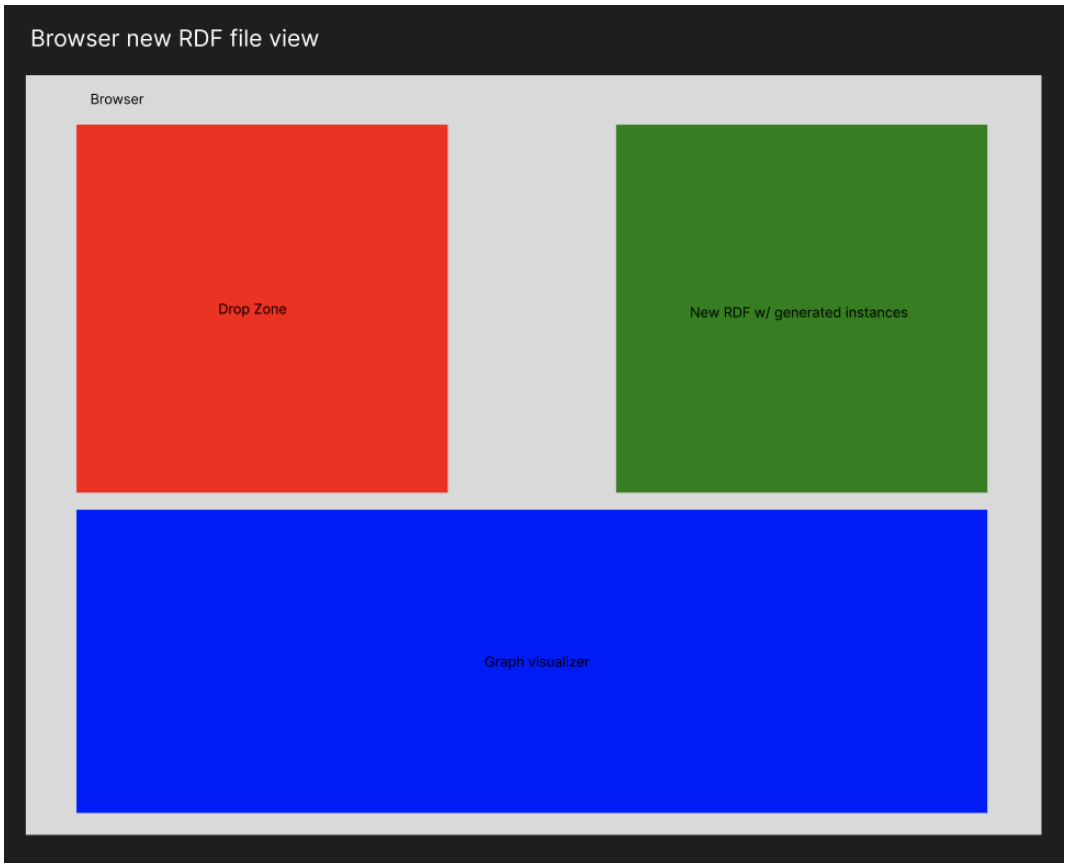


Figure 3.3: Browser new RDF side panel view

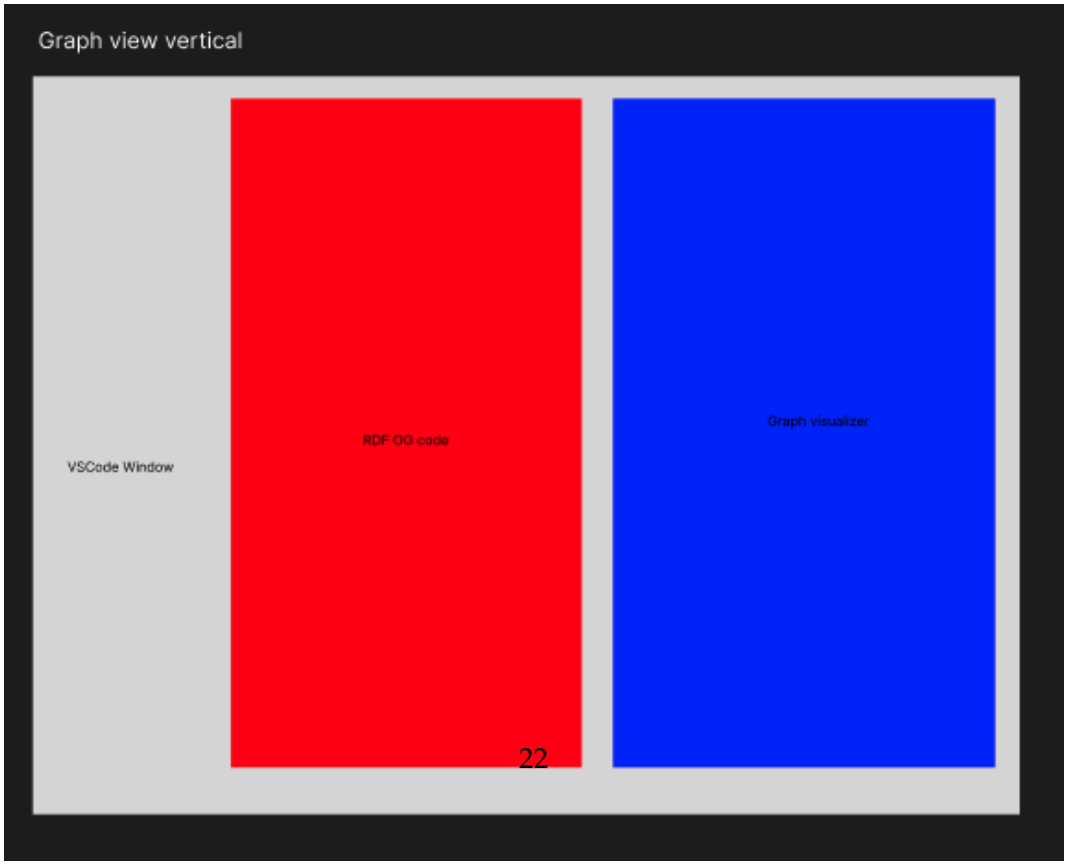


Figure 3.4: IDE Graph side panel view

The extension will not only support the visualization of RDF files in user-specified formats—like its browser counterpart—but will also allow real-time editing of the file (see Figure 3.5).

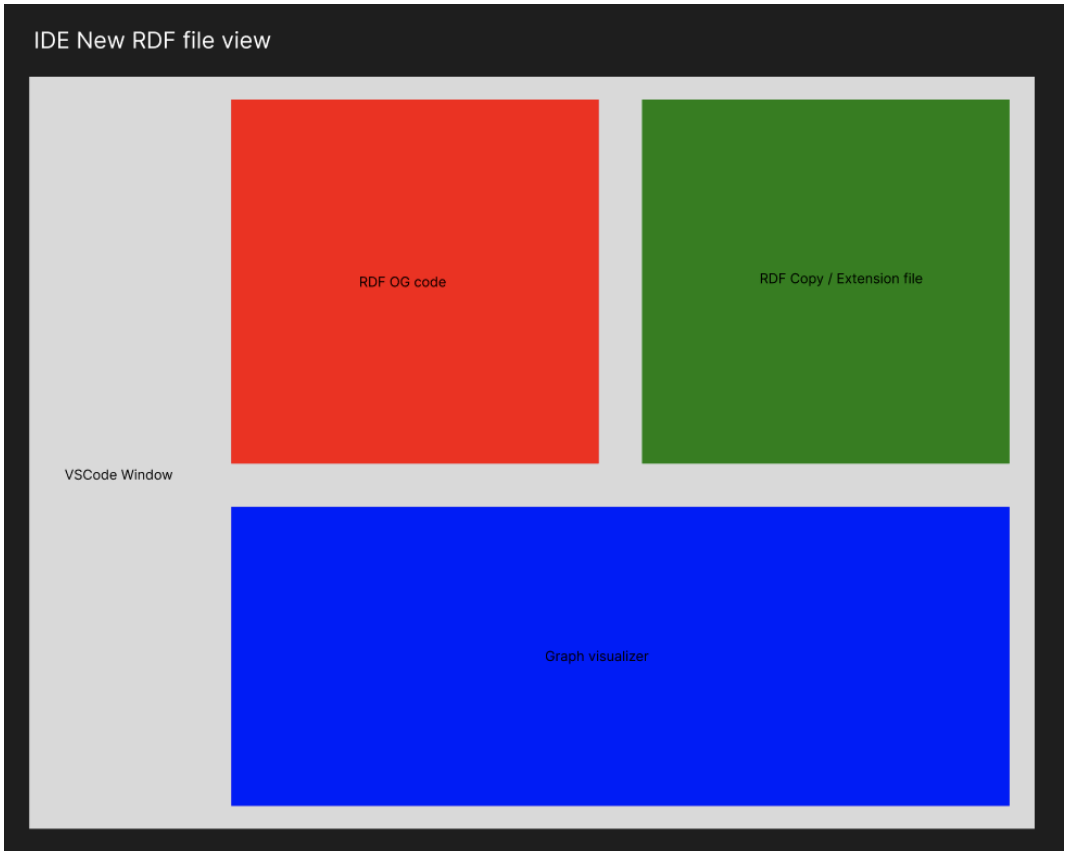


Figure 3.5: IDE new RDF side panel view

3.6.2 Server

To ensure system compatibility across various devices, operating systems, and user agents, the server is responsible for all computational tasks and data processing. By centralizing calculations and RDF graph handling on the server side, the system reduces client-side resource use and improves performance consistency.

The server plays a critical role in managing RDF data—ensuring schema validation, instance generation, SPARQL query processing, and graph construction.
Upon receiving an RDF file, the server scans it for potential flaws, such as syntactic or semantic errors. It should be capable of handling multiple RDF serialization formats, including Turtle, N3, XML, and JSON-LD.

The service generates synthetic RDF instances based on the given schema, maintaining logical consistency with the original data.
It identifies RDF classes and properties and creates additional instances where necessary to preserve semantic correctness.
This is particularly important when a property references an instance of a class that hasn't been explicitly defined. For example, if a schema includes a 'hasCar' property that expects an instance of the 'Car' class, the system will automatically generate one if it does not exist. This maintains logical coherence and satisfies all property constraints. By dynamically generating required instances, the system preserves data integrity, prevents incomplete assertions, and supports reliable reasoning processes in RDF applications.
The additional generated instances should have realistic properties and values that meet graph constraints.

Finally, the service returns the newly generated RDF file in the requested format, along with its graphical representation.

Chapter 4

Implementation

This chapter describes the implementation of the RDF instance generator and visualizer. Three systems were chosen as reference implementations: a VSCode version, an IntelliJ IDEA version, and a browser version.

4.1 Environment

The following software and operating systems were used for the implementation:

4.1.1 Software and development tools

- macOS Sequoia: used throughout the development process as the main operating system.
- Windows 11: used to test the extension for VSCode and IntelliJ in a Windows environment.
- Visual Studio Code: used to develop the web service and the VSCode extension.
- IntelliJ IDEA: used to develop the IntelliJ IDEA extension.
- Docker: used to create a virtual environment for the web service.

4.1.2 Programming languages, SDKs, and libraries

- Yeoman and VSCode Extension Generator: used to scaffold a TypeScript project ready for development.
- TypeScript and Node.js: used as the primary programming language and package manager for the VSCode extension.

- Kotlin and Gradle: used for developing and packaging the IntelliJ IDEA extension.
- Python and pip: used to implement and manage packages for the web service.
- FastAPI: the main Python library for creating the REST web service.
- rdflib: a Python library for handling and processing RDF data.
- sparqlwrapper: a Python library that simplifies the use of SPARQL.
- vis.js: a JavaScript library for creating interactive RDF graphs.

4.2 Project Structure

The implementation is divided into three distinct projects, as shown in Figure 4.1.

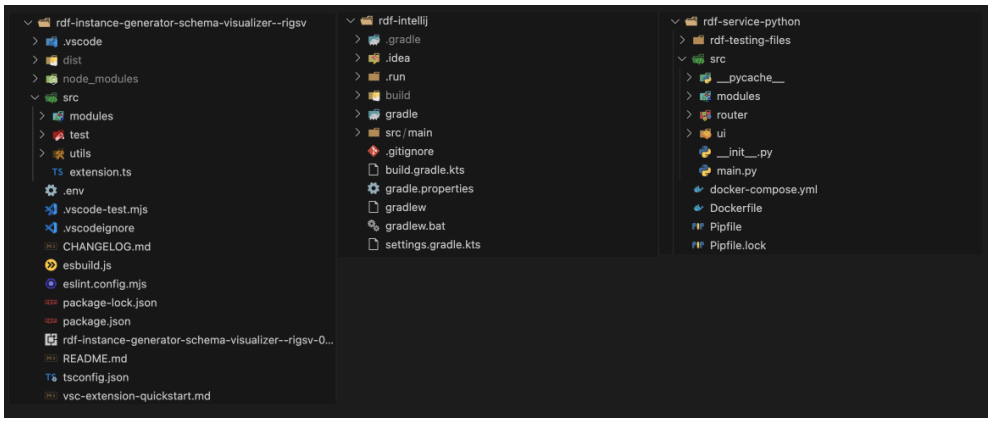


Figure 4.1: Project Structure

The first project is the web service (`rdf-service-python`), the second is the VSCode extension (`rdf-instance-generator-schema-visualizer-rigvs`), and the third is the IntelliJ plugin (`rdf-IntelliJ`).

4.3 Implementation of the Web Service

The following section details the implementation of the web service and its key components.

4.3.1 Server

Among the many Python libraries available for creating a REST web service, FastAPI was selected. It is a modern, high-performance web framework for building APIs in Python,

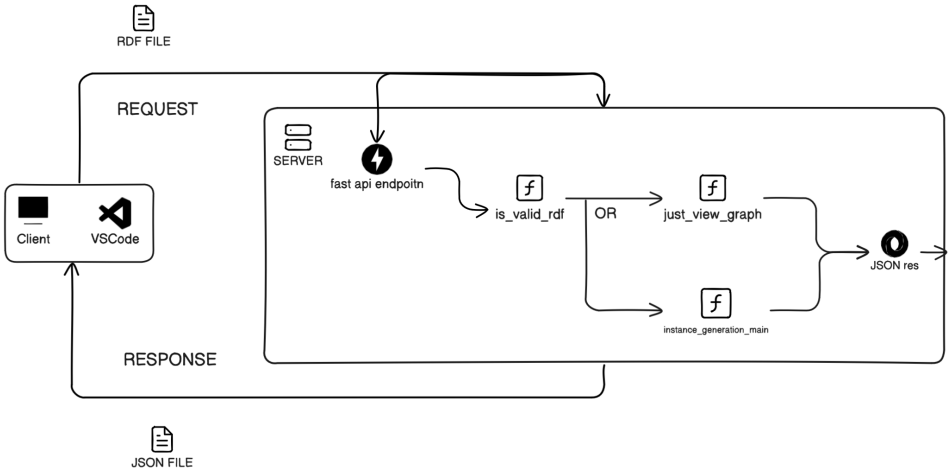


Figure 4.2: Server Schema

based on standard Python type hints [2].
The first step in setting up the server involved installing FastAPI and its dependencies. After that, initializing the app object (`app = FastAPI()`) and running the command `fastapi dev src/main.py` started the server.

The next step was to create the endpoints. In `src/router/router.py`, the router was initialized with `router = APIRouter()`, and three endpoints were created: two for the browser client and one for the extensions.

The first endpoint, `@router.get("/", response_class=HTMLResponse)`, is used to serve the `index.html` file to the web browser. This page allows users to upload RDF files (via a POST to the next endpoint), visualize the generated graph, and inspect the RDF file with the newly generated instances.

The second endpoint, `@router.post("/generate", tags=["users"])`, accepts POST requests from the browser with the RDF file. It returns the necessary data in JSON format.

The final endpoint, `@router.post("/", response_class=JSONResponse)`, handles POST requests from the VSCode and IntelliJ extensions. It also returns data in JSON format.

4.3.2 RDF Validation

When an endpoint receives an RDF file, the first function called is `is_valid_rdf(file)`. This asynchronous function is responsible for validating the incoming RDF file. First, it extracts the file extension and checks whether it exists in a predefined dictionary. If not,

an error is returned; otherwise, the process continues.
To verify both syntactic and semantic correctness, the function uses the `rdflib` library. This library allows parsing and serialization of RDF data, storing RDF triples, accessing SPARQL endpoints, and working with RDF graphs [8].
The core of this function uses `Graph.parse()`. If the file is valid, a new RDF graph is generated and returned; otherwise, the function returns `False`.

4.3.3 Graph Generator

After validating the RDF file, the next step is graph generation. The `getGraph()` function uses the parsing logic mentioned above to return the graph, the file extension, and the format to the `just_view_graph()` function.
The graph is then serialized into the requested format and into JSON-LD.

4.3.4 Instance and Graph Generator

This is the most important and complex functionality of the entire project. Just like in the previous step, the graph, file format, and extension are returned from `getGraph()`. A new graph is then initialized. To maintain consistency and readability, all namespace prefixes from the original graph (`or_graph`) are copied to the new one.

The RDF graph is then scanned using the `scan()` function, which extracts classes and properties. First, `detect_classes()` retrieves class definitions, then `detect_properties()` links properties to their domain and range. This ensures that all indirectly referenced classes are included.

Two generation paths are implemented: instance generation without property search, and with property search.

In RDF vocabularies, classes may be explicitly declared (e.g., `rdf:type rdfs:Class`) or implicitly referenced via property definitions (e.g., `rdfs:range schema:Car`).
In the case of generation without property search, the system generates instances for all identified classes—explicit or implicit.
The `generate_instance()` function receives the class URI, the new graph, the number of instances, and any defined properties.

If the class URI starts with `http://www.w3.org/2001/XMLSchema`, it is skipped. Then, two dictionaries are initialized: `processed_instances` (to avoid duplicate generation) and `initialized_instances` (to track assigned properties).

The function iterates `N` times to create instances. For each, a new URI is created (e.g., `ex:Employee_Instance1`), linked to the class URI, and added to the graph. If an instance has properties, each is processed as follows:

- If the value is a URI and maps to a primitive datatype, a literal is generated based on a predefined XSD hashmap.
- If the URI refers to another class, the function is called recursively to generate a sub-instance.
- If the value is an XSD datatype URI, a default literal is created.
- If the value is already a literal, it is directly added.
- If the value type is unsupported, an error is raised.

Each initialized property is recorded in `initialized_instances`. Once all instances are generated, they are returned.

4.3.5 Property Search

If the property search option is enabled, additional steps are required. After class detection, the `find_properties()` function retrieves implicitly referenced classes, then uses the `query()` function to fetch properties from the Linked Open Vocabularies (LOV) endpoint.

The `query()` function accepts a class term, the number of desired properties, and the ontology to search. It constructs a SPARQL query using `fetchQuery()` and runs it via `sparqlwrapper`. The results are filtered, randomized, and mapped to the target class.

The `find_properties()` output is passed to `update_props()` to enrich the property definitions used in `generate_instance()`.

Finally, the new graph is merged with the original RDF graph and returned as a JSON response, along with the file name and its JSON-LD serialization.

```
async def instance_generation_main(file, n=2, property_search=False):
  or_graph, format, fileFormatName = await getGraph(file)

  if or_graph is None:
    return None

  new_instances_graph = Graph()
  for prefix, namespace in or_graph.namespace_manager.namespaces():
```

```
    new_instances_graph.namespace_manager.bind(prefix, namespace)

  # Detect classes and their properties
  classes = scan(or_graph)
  property_definitions = {
    class_uri: details["properties"]
    for class_uri, details in classes.items()
  }

  if property_search == True:
    undeclared_classes_props = find_properties(classes, 1)
    new_property_definitions = update_props(property_definitions,
undeclared_classes_props)

    for class_uri in classes:
      generate_instance(
        class_uri,
        new_instances_graph,
        num_instances=n,
        property_definitions=new_property_definitions
      )
  else:
    for class_uri in classes:
      generate_instance(
        class_uri,
        new_instances_graph,
        num_instances=n,
        property_definitions=property_definitions
      )

  rdf_data = save_to_new_response(or_graph, new_instances_graph,
fileFormatName)
  json_dl = rdf_format_json(rdf_data, fileFormatName)

  return {
    "data": rdf_data,
    "fileName": f"new_rdf{format}",
    "json_dl": json_dl
  }

```

Listing 4.1: Main Function for RDF Instance Generation

4.4 Implementation of the Browser Application

The following section describes the implementation of the browser application, focusing on the client-side code.

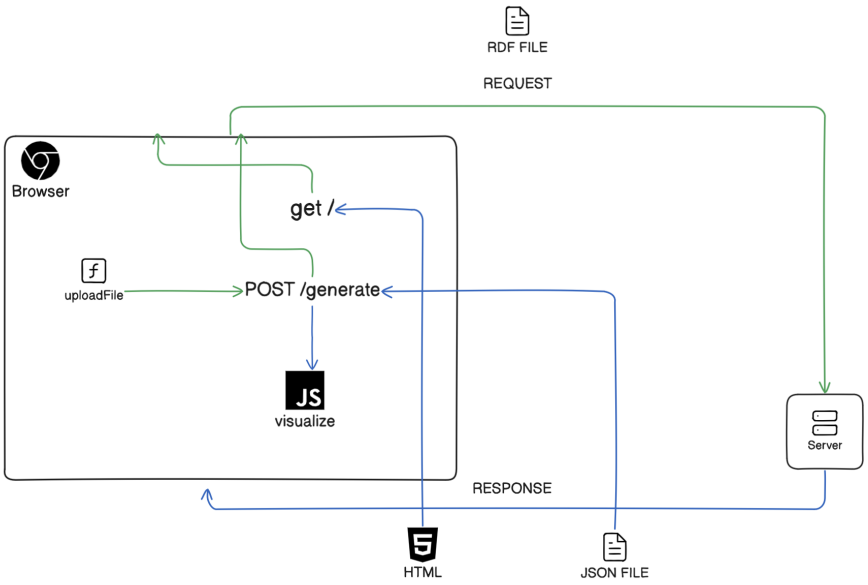


Figure 4.3: Web extension

4.4.1 The Fetch

When users want to use the RDF Instance Generator and Visualizer, the quickest way is through a web browser.
By accessing the server's default route (/), an HTML page is returned. This page allows users to upload an RDF file to the server (`/generate` route). Upon a successful upload, users can inspect the new RDF file and view the generated JSON-LD graph using the `vis.js` library.
The following describes the pipeline for converting JSON-LD into an interactive RDF graph.

4.4.2 Graph Rendering

When the `index.html` file is rendered, it loads two main JavaScript files: `uploader.js` and `visualizer.js`. The first script creates a web component called `TurtleFileUploader`, which enables users to upload files and send POST requests. Upon receiving a successful response, it injects the `json_ld` data into the custom `rdf-visualizer` as a `jsondata`

attribute. Additionally, the new RDF file (`res.data`) is attached to the DOM so the user can inspect and download it.

```
class TurtleFileUploader extends HTMLElement {
  constructor() { ... }

  connectedCallback() { ... }

  async uploadFile(edit = false, search = false) {
    const fileInput = this.shadowRoot.getElementById("fileInput");
    const responseDiv = this.shadowRoot.getElementById("response");
    const rdfVisualizer = document.getElementById("rdfVisualizer");
    const n = this.shadowRoot.getElementById("n").value;

    if (fileInput.files.length === 0) { ... }

    if (search && n > 3) { ... }

    const formData = new FormData();
    formData.append("file", fileInput.files[0]);

    try {
      const response = await fetch(`/generate?n=${n}&edit=${edit}&
    property_search=${search}`, {
        method: "POST",
        body: formData
      });

      if (!response.ok) throw new Error(`HTTP error! Status: ${response
    .status}`);

      const res = await response.json();
      responseDiv.innerText = res.data;

      const blob = new Blob([res.data]);
      const url = URL.createObjectURL(blob);
      const a = document.createElement("a");
      a.href = url;
      a.download = res.fileName;
      a.innerText = `Download ${res.fileName}`;
      a.style.display = "block";
      responseDiv.appendChild(a);

```

```
      rdfVisualizer.setAttribute("jsondata", res.json_dl);
    } catch (error) {
      responseDiv.innerHTML = "Error: " + error.message;
    }
  }
}

customElements.define('turtle-file-uploader', TurtleFileUploader);
```

Listing 4.2: `TurtleFileUploader` component

The second script defines the `rdf-visualizer` component. This component listens for the `jsondata` attribute and triggers a parser function to safely deserialize the data. To ensure consistent rendering, the data is normalized into an array of RDF subject objects, each containing an ID, type, and property-value pairs.

The core conversion from RDF semantics to network-compatible format is handled by the `processJsonLd` function. Each RDF subject is transformed into a graph node and classified as:

- **Class** (`rdfs:Class`) – shown as orange boxes
- **Property** (`rdf:Property`) – shown as green-yellow diamonds
- **Instance** (URI-based) – shown as blue dots
- **Literal** – shown as yellow boxes

```
function processJsonLd(data, dynamicPrefixes) {
  const nodesMap = {};
  const edges = [];
  let literalCounter = 0;

  data.forEach(item => {
    const subjectId = item['@id'];
    let nodeType = 'instance';

    if (item['@type']) {
      const types = Array.isArray(item['@type']) ? item['@type'] : [
    item['@type']];
      if (types.includes("...#Class")) nodeType = 'class';
      else if (types.includes("...#Property")) nodeType = 'property';
    }

    if (!nodesMap[subjectId]) {
```

```
17        let label = item["...#label"] ? item["...#label"][0]["@value"] :
    shorten(subjectId, dynamicPrefixes);
18        let shape = nodeType === 'class' ? 'box' : nodeType === 'property
    ' ? 'diamond' : 'dot';
19        let color = nodeType === 'class' ? '#FFA500' : nodeType === '
    property' ? '#ADFF2F' : '#97C2FC';
20        nodesMap[subjectId] = { id: subjectId, label, shape, color,
    baseColor: color, nodeType, font: { color: "#000000" } };
21      }
22
23      if (item['@type']) {
24        types.forEach(t => {
25          let typeId = typeof t === 'string' ? t : t['@id'];
26          if (typeId && !nodesMap[typeId]) {
27            nodesMap[typeId] = { id: typeId, label: shorten(typeId,
    dynamicPrefixes), shape: 'box', color: '#FFA500', ... };
28          }
29          edges.push({ from: subjectId, to: typeId, label: shorten("...#
    type", dynamicPrefixes), dashes: true, color: { color: '#000' } });
30        });
31      }
32
33      Object.keys(item).forEach(prop => {
34        if (prop === '@id' || prop === '@type') return;
35        item[prop].forEach(val => {
36          if (val['@id']) {
37            if (!nodesMap[val['@id']]) nodesMap[val['@id']] = { id: val['
    @id'], label: shorten(val['@id'], dynamicPrefixes), shape: 'dot',
    ... };
38            edges.push({ from: subjectId, to: val['@id'], label: shorten(
    prop, dynamicPrefixes), arrows: 'to', ... });
39          } else if (val['@value']) {
40            const literalId = `literal_${literalCounter++}`;
41            nodesMap[literalId] = { id: literalId, label: String(val['
    @value']), shape: 'box', color: '#FFD700', ... };
42            edges.push({ from: subjectId, to: literalId, label: shorten(
    prop, dynamicPrefixes), arrows: 'to', ... });
43          }
44        });
45      });
46    });
47
48    return { nodes: Object.values(nodesMap), edges };
49  }
```

Listing 4.3: `processJsonLd` function used for graph construction

Each RDF triple is mapped as an edge between a subject and an object, labeled by the predicate. Type relationships (i.e., `rdf:type`) are visualized using dashed lines to distinguish them from regular property connections.

After defining the nodes and edges, the data is loaded into a `vis-network` object. The layout engine distributes the graph nodes automatically, and a stabilization process improves readability. To enhance user experience, when a node is clicked, all non-neighboring nodes are blurred out to focus attention.

4.5 Implementation of the Visual Studio Code Extension

Visual Studio Code, originally intended as a lightweight source code editor, has evolved into a fully functional IDE thanks to its extensive plugin ecosystem.
Since VSCode is built on top of Electron—a framework for building desktop applications using web technologies—it is possible to build extensions using familiar web tools (HTML, CSS, JavaScript).
This extension is developed using the Visual Studio Code Extension API, which provides interfaces and classes for interacting with the editor's core features.
The following section describes the implementation of the VSCode extension and its key features.

4.5.1 Commands Architecture

Commands trigger actions in Visual Studio Code. In this project, commands are used by the RIGVS extension to expose the following features to users:

- `extension.viewGraph` – View RDF graph
- `extension.runGraph` – Generate instances and view graph
- `extension.searchProperty` – Generate instances with properties and view graph

```
...
"commands": [
  {
    "command": "extension.runGraph",
    "title": "RIGSV generate instance & view graph"
  },
  {
```

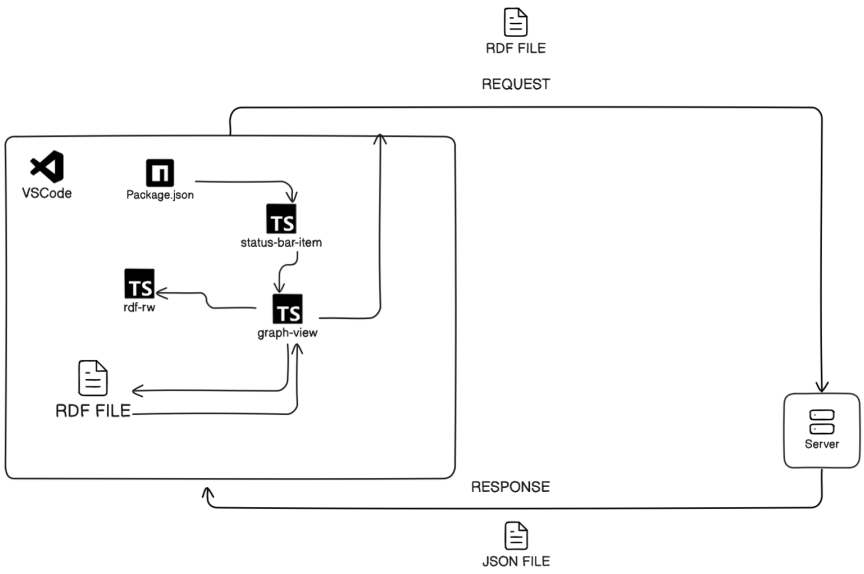


Figure 4.4: VSCode extension

```
    "command": "extension.searchProperty",
    "title": "RIGSV generate instance with properties"
  },
  {
    "command": "extension.openMenu",
    "title": "RIGSV Options"
  },
  {
    "command": "extension.settings",
    "title": "RIGSV settings"
  },
  {
    "command": "extension.viewGraph",
    "title": "RIGSV view graph"
  },
  {
    "command": "extension.editRDF",
    "title": "RIGSV edit RDF"
  }
],
...
```

Listing 4.4: package.json commands declaration

All commands in VSCode can be executed using `cmd + shift + p`. After opening the

input dialog, users can run any command by typing its title and pressing enter.

The first command, `extension.viewGraph`, opens a WebView inside the VSCode workspace. This WebView displays the graph returned by the server.
Currently, this command only works while an RDF file is focused in the editor. When executed, it creates a new panel via `vscode.window.createWebviewPanel`, with scripts enabled and placed beside the editor window. The `getWebviewContent` function is then called.

This function extracts the RDF content from the focused file using `getRDFContent` and sends it to the server via `sendRDFContent`. Then it injects static URLs for the necessary scripts (`visualizer.js` and `uploader.js`) into the HTML content.

Because VSCode WebViews operate in a sandboxed environment, Content Security Policies (CSP) restrict access to external scripts and styles. To work around this, a meta tag is added to the HTML head:

```
<meta http-equiv="Content-Security-Policy" content="${csp}">
```

After the content is prepared, it is rendered inside the WebView panel. The page also listens for two specific button clicks: `editButton` and `exportGraph`.
Clicking `editButton` triggers the command `extension.editRDF`, which opens the new RDF graph in the editor using the original RDF format.
The `extension.exportGraph` command allows exporting the rendered graph as a PNG image.

The second command, `extension.runGraph`, performs the same functions but adds a pop-up input that asks the user for the number of instances to generate.
The final command, `extension.searchProperty`, behaves like `runGraph` but includes the additional property search functionality.

```
export function openWebView(context: vscode.ExtensionContext) {
  const viewGraph = vscode.commands.registerCommand(
    "extension.viewGraph",
    async () => {
      vscode.window.showInformationMessage("Graph view is loaded");

      let panel = vscode.window.createWebviewPanel(
        "rdfVisualizer",
        "RDF Visualizer",
        vscode.ViewColumn.Beside,
```

```
        {
          enableScripts: true,
          retainContextWhenHidden: false,
        }
      );

      if (envConfig.serviceEndpoint) {
        const htmlContent = await getWebviewContent(
          panel.webview,
          envConfig.serviceEndpoint,
          true
        );
        panel.webview.html = htmlContent;
      }

      panel.webview.onDidReceiveMessage(
        async (message) => {
          if (message.command === "editRDF") {
            vscode.commands.executeCommand("extension.editRDF");
          }
        },
        undefined,
        context.subscriptions
      );
      exportGraph(panel)
    }
  );
  const runGraph = vscode.commands.registerCommand(...)
  const searchProperty = vscode.commands.registerCommand(...)
  const editRDFCommand = vscode.commands.registerCommand(...)
  ...
}
```

Listing 4.5: openWebView

Currently, these commands can only be used while an RDF file is focused in the editor. To simplify access, a quick-access menu button has been added to the bottom-right corner of the VSCode window. Clicking the button opens a modal that allows users to select between the three available commands.

4.6 Implementation of the IntelliJ IDEA plugin

The following section describes the implementation of the IntelliJ IDEA plugin and showcases its key feature.

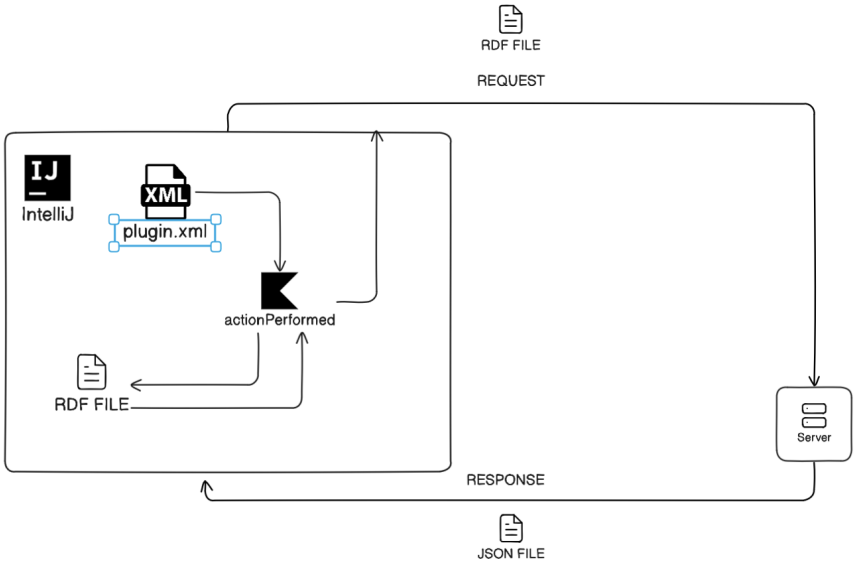


Figure 4.5: IntelliJ Extension

Similar to the VSCode extension, the goal of the IntelliJ IDEA plugin is to provide a user-friendly interface for generating RDF instances and visualizing them in a side panel as a graph.
This extension, developed primarily to demonstrate the flexibility of the tool's distributed architecture, currently implements only the feature responsible for rendering RDF graphs in a side panel.

The plugin is developed using the Kotlin programming language, the Gradle plugin manager, and the JetBrains Plugin DevKit.
Like in VSCode, the IntelliJ plugin defines **actions** (comparable to VSCode's `commands`) that execute specific functions in response to user interactions.

```
<?xml version="1.0" encoding="UTF-8"?>
<idea-plugin>
    <id>com.example.rdf</id>
    <name>RDF Visualizer</name>
    <vendor>Wesley G. Obi</vendor>
    <description>Visualizing RDF Files Graphically Using an External
   Python Service</description>

    <depends>com.intellij.modules.platform</depends>

    <actions>
```

```
        <action id="RDFVisualizer.Convert"
                class="com.example.rdf.RDFVisualizerAction"
                text="Visualize RDF"
                description="Converts the current RDF file into a graphical visualization">
            <add-to-group group-id="EditorPopupMenu" anchor="last"/>
        </action>
    </actions>
</idea-plugin>
```

Listing 4.6: IntelliJ Actions

When a user right-clicks on an RDF file, the **Visualize RDF** action appears in the context menu.

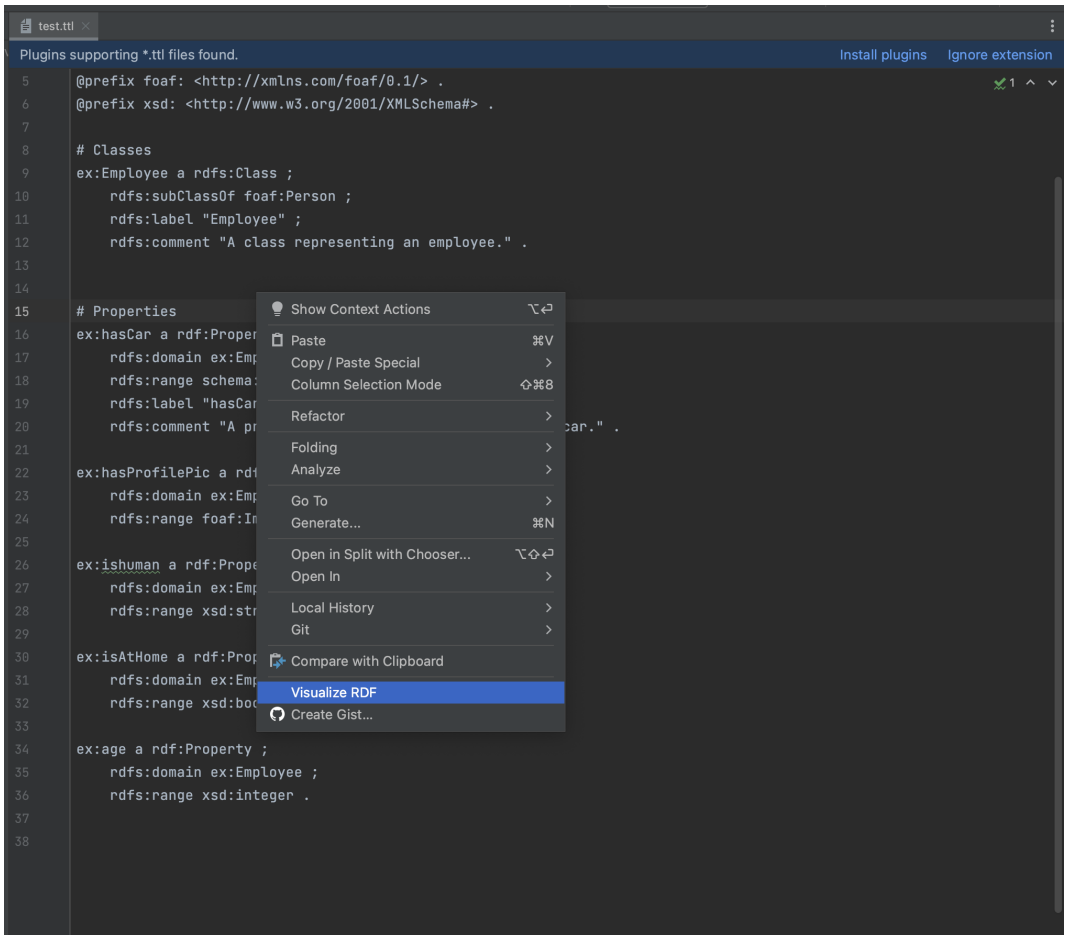


Figure 4.6: Action call in IntelliJ

By simply clicking on it, the action is triggered and executes the `actionPerformed` function.

This function extracts the RDF schema from the currently focused file, wraps it in a `formData` object, and sends it to the Python server.
The response is passed to the `showGraphViewer` function, which renders the graph in a side panel.
As in the VSCode extension, the HTML content is adjusted to bypass CSP restrictions, ensuring that external resources (scripts and styles) can be loaded and rendered properly inside the IntelliJ WebView.

```
override fun actionPerformed(e: AnActionEvent) {
    ...
    val formData = MultipartBody.Builder()
        .setType(MultipartBody.FORM)
        .addFormDataPart("fileUpload", file.name, requestBody)
        .build()

    val request = Request.Builder()
        .url("http://localhost:8000/")
        .post(formData)
        .build()

    val client = OkHttpClient()

    try {
        val response = client.newCall(request).execute()
        if (response.isSuccessful) {
            val jsonResponse = response.body?.string() ?: throw
    Exception("Empty response body")
            val responseBody: ServerResponse = Json.decodeFromString(
    jsonResponse)

            showGraphViewer(project, responseBody.content)
        } else { ... }
        ...
    }
}
```

Listing 4.7: Render graph in WebView in IntelliJ

4.7 Documentation and Deployment Architecture

The documentation for this project is available in the GitHub repository in the `README.md` file. It provides two types of installation instructions: Developer installation and User installation.

For developers, the setup involves installing the required programming languages, runtimes, and dependencies (e.g., Python, Node.js, pip, npm). Additionally, they must start the Python server and load the extension inside the VSCode development environment.

For end users, the installation process is significantly simpler. The repository already includes a pre-built version of the VSCode extension, which can be installed like any other VSCode extension.

To ensure compatibility across different systems and avoid requiring users to manually install any development tools, the web service has been containerized using Docker. This allows users to run the system by simply installing Docker Desktop, without needing to configure environments or install dependencies manually.

Chapter 5

Evaluation

This chapter describes the evaluation method and results for the developed software solutions. The evaluation is divided into three main sections:

- Functional Evaluation
- Performance Evaluation
- Edge cases Analysis

All the test has been executed on the same machine, with the same configuration.

Test Environment Intel Core i7, 16 GB RAM, MacOS 15.4.1

5.1 Functional Evaluation

5.1.1 Instance Generation

Goal The first core functionality to evaluate is the **automatic generation of RDF instances**. This feature assist users by generating RDF instances based on a given RDF schema .

Test Description To validate this capability, two test scenarios were conducted:

- **Test 1.1**: Generate instances from an RDF schema without additional properties.
- **Test 1.2**: Generate instances from an RDF schema and search for additional properties.

Expected Result

In the first case, the system was expected to:

- Correctly generate RDF instances for each class.

- Ensure that instances respect domain and range restrictions.
- Keep the schema original schema structure intact, without any modifications in the classes or properties definitions.

In the second case, the system was expected to:

- Correctly generate RDF instances for each class.
- Correctly generate additional properties for instance of non-implicitly referolive classes.
- Ensure that instances respect domain and range restrictions.
- Keep the schema original schema structure intact, without any modifications in the classes or properties definitions.

Test Environment

- Test device: Intel Core i7, 16 GB RAM, MacOS 15.4.1

Actual Results

The following two code sample, are the results of the two tests cases. For both the cases, the same RDF schema was provided. The code lines highlighted in olive green represent the synthetic instances generated by system. Three new instances were requested to be generated.
Each of them has been marked as passed.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema1: <http://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Employee a rdfs:Class ;
    rdfs:label "Employee" ;
    rdfs:comment "A class representing an employee." ;
    rdfs:subClassOf foaf:Person .

ex:Employee_Instance1 a ex:Employee ;
    ex:age 0 ;
    ex:hasCar ex:Car_Instance1 ;
    ex:hasProfilePic ex:Image_Instance1 ;
```

```
        ex:isAtHome false ;
        ex:ishuman "unknown"^^xsd:string .

    ex:Employee_Instance2 a ex:Employee ;
        ex:age 0 ;
        ex:hasCar ex:Car_Instance2 ;
        ex:hasProfilePic ex:Image_Instance2 ;
        ex:isAtHome false ;
        ex:ishuman "unknown"^^xsd:string .

    ex:Employee_Instance3 a ex:Employee ;
        ex:age 0 ;
        ex:hasCar ex:Car_Instance3 ;
        ex:hasProfilePic ex:Image_Instance3 ;
        ex:isAtHome false ;
        ex:ishuman "unknown"^^xsd:string .

    ex:Person_Instance1 a foaf:Person .

    ex:Person_Instance2 a foaf:Person .

    ex:Person_Instance3 a foaf:Person .

    ex:age a rdf:Property ;
        rdfs:domain ex:Employee ;
        rdfs:range xsd:integer .

    ex:hasCar a rdf:Property ;
        rdfs:label "hasCar" ;
        rdfs:comment "A property linking an employee to their car." ;
        rdfs:domain ex:Employee ;
        rdfs:range schema1:Car .

    ex:hasProfilePic a rdf:Property ;
        rdfs:domain ex:Employee ;
        rdfs:range foaf:Image .

    ex:isAtHome a rdf:Property ;
        rdfs:domain ex:Employee ;
        rdfs:range xsd:boolean .

    ex:ishuman a rdf:Property ;
        rdfs:domain ex:Employee ;
        rdfs:range xsd:string .
```

```
    ex:Car_Instance1 a schema1:Car .

    ex:Car_Instance2 a schema1:Car .

    ex:Car_Instance3 a schema1:Car .

    ex:Image_Instance1 a foaf:Image .

    ex:Image_Instance2 a foaf:Image .

    ex:Image_Instance3 a foaf:Image .

```

Listing 5.1: Result Test Case 1.1

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema1: <http://schema.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:Employee a rdfs:Class ;
    rdfs:label "Employee" ;
    rdfs:comment "A class representing an employee." ;
    rdfs:subClassOf foaf:Person .

ex:Employee_Instance1 a ex:Employee ;
    ex:age 0 ;
    ex:hasCar ex:Car_Instance1 ;
    ex:hasProfilePic ex:Image_Instance1 ;
    ex:isAtHome false ;
    ex:ishuman "unknown"^^xsd:string .

ex:Employee_Instance2 a ex:Employee ;
    ex:age 0 ;
    ex:hasCar ex:Car_Instance2 ;
    ex:hasProfilePic ex:Image_Instance2 ;
    ex:isAtHome false ;
    ex:ishuman "unknown"^^xsd:string .

ex:Employee_Instance3 a ex:Employee ;
    ex:age 0 ;
    ex:hasCar ex:Car_Instance3 ;
    ex:hasProfilePic ex:Image_Instance3 ;
```

```
    ex:isAtHome false ;
    ex:ishuman "unknown"^^xsd:string .

ex:Person_Instance1 a foaf:Person ;
    foaf:name "FedX" .

ex:Person_Instance2 a foaf:Person ;
    foaf:name "FedX" .

ex:Person_Instance3 a foaf:Person ;
    foaf:name "FedX" .

ex:age a rdf:Property ;
    rdfs:domain ex:Employee ;
    rdfs:range xsd:integer .

ex:hasCar a rdf:Property ;
    rdfs:label "hasCar" ;
    rdfs:comment "A property linking an employee to their car." ;
    rdfs:domain ex:Employee ;
    rdfs:range schema1:Car .

ex:hasProfilePic a rdf:Property ;
    rdfs:domain ex:Employee ;
    rdfs:range foaf:Image .

ex:isAtHome a rdf:Property ;
    rdfs:domain ex:Employee ;
    rdfs:range xsd:boolean .

ex:ishuman a rdf:Property ;
    rdfs:domain ex:Employee ;
    rdfs:range xsd:string .

ex:Car_Instance1 a schema1:Car ;
    schema1:image ex:systems_and_procedures.svg_Instance1 .

ex:Car_Instance2 a schema1:Car ;
    schema1:image ex:systems_and_procedures.svg_Instance2 .

ex:Car_Instance3 a schema1:Car ;
    schema1:image ex:systems_and_procedures.svg_Instance3 .

ex:Image_Instance1 a foaf:Image ;
    foaf:name "Noel Ignatiev" .
```

```
76
77 ex:Image_Instance2 a foaf:Image ;
78      foaf:name "Noel Ignatiev" .
79
80 ex:Image_Instance3 a foaf:Image ;
81      foaf:name "Noel Ignatiev" .
82
83 ex:systems_and_procedures.svg_Instance1 a
       <https://raw.githubusercontent.com/OpenHPS/POSO/main/docs/images/systems_and_procedures.svg
84
85 ex:systems_and_procedures.svg_Instance2 a
       <https://raw.githubusercontent.com/OpenHPS/POSO/main/docs/images/systems_and_procedures.svg
86
87 ex:systems_and_procedures.svg_Instance3 a
       <https://raw.githubusercontent.com/OpenHPS/POSO/main/docs/images/systems_and_procedures.svg
```

Listing 5.2: Result Test Case 1.2

Comments The simple instance generation was handled smoothly, completing instance generation in under 1 second. In the instance generation with property search case, although generation succeeded, a delay of approximately 40 seconds was observed during the function execution. This is due to the time required to search for additional properties in the Linked Open Vocabularies system. No critical issues were found.
Future work could include optimization of generation and property search algorithm, maybe with the use of multithreading and parallel processing.

5.1.2 Graph visualization

Goal The second-biggest functionality to evaluate is the **Graph visualization**. This feature allow user to better see the instances that have been generated thanks to their graph representation.

Test Description To validate this capability, the following test scenarios were conducted:

- **Test 2.1**: After the server response, a graph visualization is generated in the provided canvas - on Browser.
- **Test 2.2**: After the server response, a graph visualization is generated in the provided canvas - on VSCode.

Expected Result

In the both the scenarios, the expected results are:

- Correctly visualize the canvas with the related graph.
- Be able to interact with the graph (i.e. clicking and highlighting nodes, move nodes, zooming in and out).

Actual Results

The following image, is an example resutl of the tests cases.
Each of them has been marked as passed.

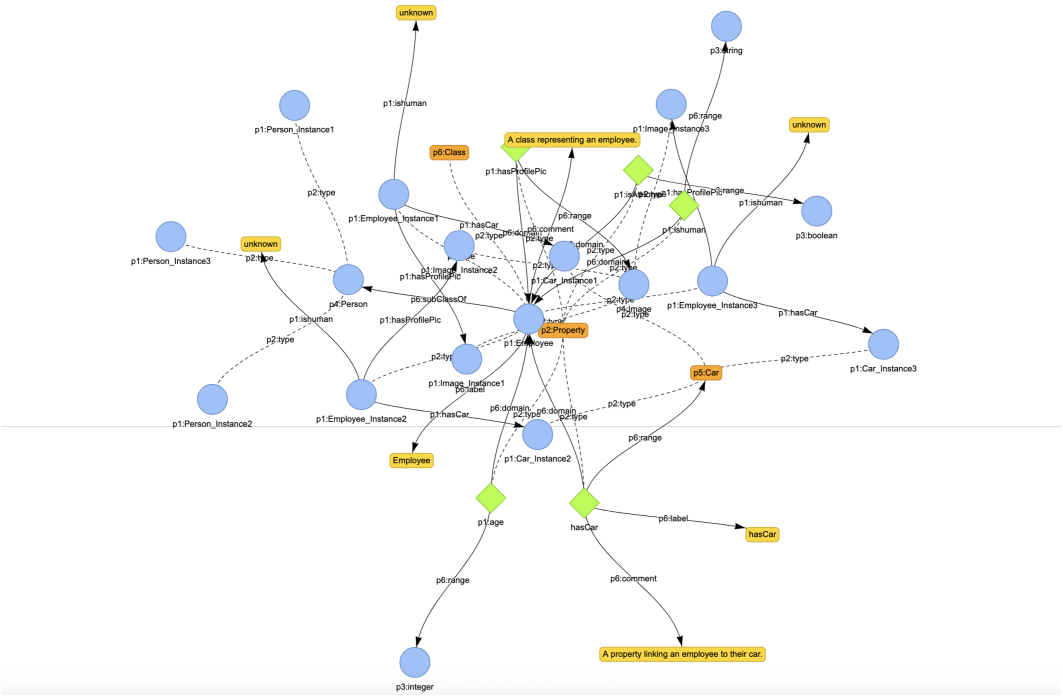


Figure 5.1: Rending of the RDF graph

Comments For both the scenarios the graphs have been rendered correctly and no critical issues were found. Unfortunately, I could find a way to stabilize the graph rendering in order to have the very same in the same place at each render. This would have help the user to better visualize the graph in case of some live edits in the RDF schema.
Some future work could include a better rendering engine and node distribution on the canvas.

5.1.3 RDF Graph and Schema Export

Goal The next capability to evaluate is the **RDF Graph and Schema Export**. The user must be able to export the RDF schema and the generated graph.

Test Description To validate this functions, the following test scenarios were conducted:

- **Test 3.1**: After the server response, a button allows to download the newly generated RDF schema - on Browser.
- **Test 3.2**: After the server response, a button allows to visualize the newly generated schema in a new IDE window - on VSCode.
- **Test 3.3**: After the server response, a button allows to export the RDF graph in a png format - on Browser.
- **Test 3.4**: After the server response, a button allows to export the RDF graph in a png format - on VSCode.

Expected Result

For two initial tests (**Test 3.1** and **Test 3.2**) scenario, the expected results are:

- The correct generation of the download button.
- No anomalies in the download file.

The remaining two tests (**Test 3.3** and **Test 3.4**), are expected to:

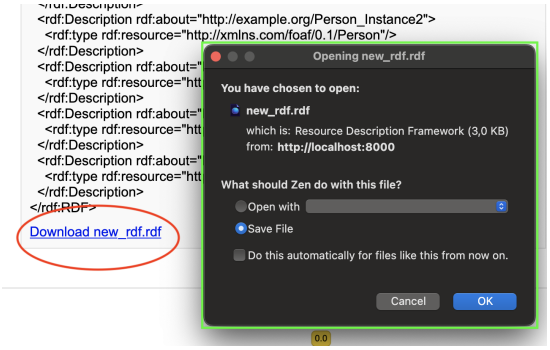
- Generate a button that allow the graph to be exported in a PNG format.
- The image graph in the PNG file is consistent with the one visualized in the application canvas.

Actual Results

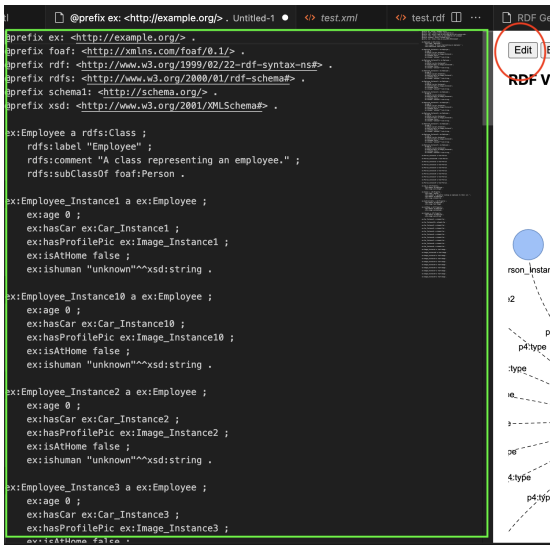
The figure 5.2 showcase the first two test cases. The performed action has been marked with a red circle. The outcome of the action is marked in a light green square.

The figure 5.3 is the result of the last two test cases. The same marking process has also been in use in this figure.

All of them has been marked as passed.

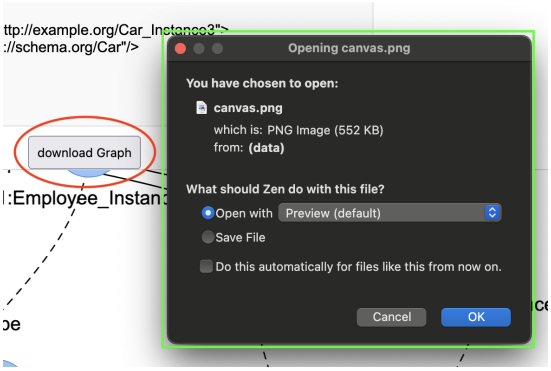


(a) Click action on the download button to download the RDF schema

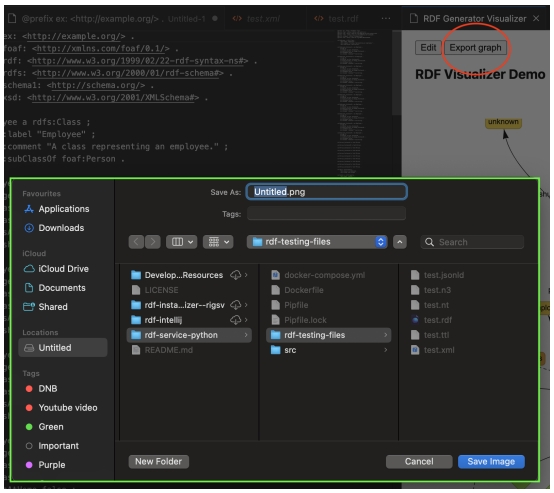


(b) Click action on the edit button to download-/edit the RDF schema

Figure 5.2: RDF Schema download methods for Browsers client and IDE client



(a) Click action on the download button to download the RDF Graph



(b) Rendering of RDF graph B

Figure 5.3: RClick action on the download button to download the RDF Graph

Comments All the test have passed and not critical issue were found the testing process. Although one minore but has been found while testig the edit button in the IDE enviroment. When the button is clicked and the schema is visualized in a new VSCode tab, if the user edits the file, it cannot be sent to the server directly. The file must be first saved and then sent to the server. Any action performed on a not saved file will cause the server to answer with an error.

Future updated will provide a solution to the described bug and improvements in the RDF scheme edit mode, like allowing edits in the browser mode before the download and the options to re-render the graph automatically after a change in the file.

5.2 Performance Evaluation

5.2.1 Rendering Time

Goal This test evaluates the responsiveness of the RDF visualization system by measuring the time required to render RDF graphs of increasing complexity. The rendering time is defined as the duration from server response to the completion of layout and DOM paint on canvas.

Method To test the rendering performance, three different schema sizes were generated:

- **Small schema**: 10 instances generated
- **Medium schema**: 50 instances generated
- **Large schema**: 100 instances generated

Each rendering test was executed 3 times, and the average rendering time was recorded using `performance.now()` browser function.

Results

Schema Size	Instances	Avg. Render Time (ms)
Small	10	121
Medium	50	4638
Large	100	9871

Table 5.1: Rendering time of RDF graphs with increasing complexity

Comments The results demonstrate a steep increase in rendering time with the number of nodes. While the small schema rendered almost instantly (121 ms), the medium and large schemas took 4.6 and 9.8 seconds respectively. This nonlinear growth in rendering time is attributed to the physics-based layout engine used by vis.js.

The delays observed in medium and large graphs can negatively affect user experience. To guarantee a nice UX the instances that can be generated are limited to a maximum number of ten. Future optimizations could include:

- Reducing the number of stabilization iterations for larger graphs.
- Using simplified layouts (e.g., hierarchical layout).

- Introducing progressive or chunked rendering for large datasets.

5.2.2 Instance generation time

Goal This test evaluates the responsiveness of the RDF instance generation system (with no properties search) from the incoming request to the server response.

Method To test the rendering performance, three different schema sizes were generated:

- **Small schema**: 10 instances generated
- **Medium schema**: 50 instances generated
- **Large schema**: 100 instances generated

Each generation test was executed 3 times, and the average response time was recorded using the `Time` python library.

Results

Schema Size	Instances	Avg. Render Time (ms)
Small	10	36.2
Medium	50	98.0
Large	100	142.22

Table 5.2: Response time of RDF instance generator function with increasing complexity

Comments The results indicate that the instance generation time increases moderately with the number of instances requested. The small schema completed generation in approximately 36 ms, while the medium and large schemas required 98 ms and 142 ms respectively. Although the growth is not exponential, the system's responsiveness could degrade if the quality of the hardware is lowered.

To ensure a consistently smooth user experience, be coherent with the current graph visualization limit and avoid potential performance bottlenecks for machine with a low budget hardware, the current implementation limits instance generation to a maximum of ten instances per request.

Future enhancements may explore asynchronous processing, or batched generation to support larger requests without compromising server responsiveness.

5.2.3 Instance generation time with property search

Goal This test evaluates the responsiveness of the RDF instance generation system, with properties search, from the incoming request to the server response.

Method To test the rendering performance, three different schema sizes were generated:

- **Small schema**: 1 instances generated
- **Medium schema**: 5 instances generated
- **Large schema**: 10 instances generated

Each generation test was executed one time only to not stress the LOV api service with testing request, and the average response time was recorded using the `Time` python library.

Results

Schema Size	Instances	Avg. Render Time (ms)
Small	1	206592
Medium	5	215507
Large	10	245350

Table 5.3: Response time of RDF instance generator function with propertiesn search

Comments The results reveal a surprising insight: the number of instances requested had minimal influence on the overall generation time. Instead, the dominant factor affecting performance was the number of properties that required semantic enrichment with the vocabulary lookup, specifically those whose ranges point to other classes.

This is explained by the fact that, for each such property, the system performs a query against the Linked Open Vocabularies (LOV) API to search for semantically relevant property names. As the number of properties requiring lookup increases, so does the total response time, regardless of the number of instances being generated.

Due to this dependency on an external service and its associated latency, the current implementation limits instance generation with property search to a maximum of 3 instances with only 1 property enhancements per class. This helps reduce the number of API calls and ensures acceptable response times.

Future improvements may include result caching, local vocabulary indexing and batching to enhance performance.

5.2.4 Edge Case Analysis

Goal This section evaluates how the system handles unexpected input scenarios that may occur during RDF instance generation. The objective is to ensure robustness and graceful failure handling, even when inputs deviate from normal expectations.

Method A series of edge case tests were conducted to assess the stability and correctness of the system in the following situations:

- **Invalid file:** The input file is not an rdf file
- **Empty Schema:** Schema with no classes or properties defined.
- **Circular Class References:** A Class (A) has a property pointing to another Class (B), which points back to the previous Class.
- **Unsupported RDF Constructs:** Use of blank nodes or advanced RDF syntax that has not specifically handled.
- **Highly Connected Schema:** A class with more than 20 properties linking to multiple other classes.
- **Malformed Input:** Invalid RDF syntax.

Each scenario was executed through the instance generator and monitored for application stability, error messages, and correctness of the output.

Results The application performed robustly in most edge case scenarios. Table 5.4 summarizes the outcomes.

Test Case	Status	Notes
Invalid file	Partial	Server returned a generic error response (*Error loading content*).
Empty Schema	Pass	Server returned an empty canvas with no errors.
Circular Class References	Pass	Instance generation completed; circular references were handled without infinite loops.
Unsupported RDF Constructs	Pass	Blank nodes were handled without failure; no warning issued.
Highly Connected Schema	Partial	Significant performance degradation observed in the rendering of the graph.
Malformed Input	Partial	Server returned a generic error response (*Error loading content*).

Table 5.4: Summary of edge case testing outcomes

Comments The system demonstrates strong resilience against most edge cases. In particular, circular class references and highly nested schemas did not lead to crashes or infinite loops. However, two weak points were identified: the invalid file and malformed input. Addressing these cases would improve user experience and system robustness.

Chapter 6

Conclusion

This final chapter describes the work that has been done throughout the course of this thesis. It revises the main ideas behind the application developemt and summarizing the key steps taken during the implementation. It then highlights some of the challenges that has been addressed and problems that still remain unsolved. The chapter ends with a brief outlook on how the current solution could be improved and extended in the future.

6.1 Summary

The first step that has been done for this project is a research and ananysis of RDFs and the technologies that are currently orbiting around like the semantic web or the owl ontology language.

After ahving new know, a a review on the programs and applications that are made for the rdf generation or visualization like protege and the GAIA project.

6.2 Dissemination

Who uses your component or who will use it? Industry projects, EU projects, open source...? Is it integrated into a larger environment? Did you publish any papers?

6.3 Problems Encountered

Summarize the main problems. How did you solve them? Why didn't you solve them?

6.4 Outlook

Future work will enhance Component X with new services and features that can be used ...

Appendix

Bibliography

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web". In: *Scientific American* (2001). https://www.w3.org/RDF/Metalog/docs/sw-easy,Accessed: 2025-03-05. URL: https://www.scientificamerican.com/article/the-semantic-web/.

[2] *fastapi*. https://fastapi.tiangolo.com/. fastapi.

[3] Ivan Herman. *Tutorial on Semantic Web Technologies*. https://www.w3.org/People/Ivan/CorePresentations/RDFTutorial/Slides.html. Accessed: 2025-03-05. 2003.

[4] *issemantic.net*. https://www.sciencedirect.com/topics/computer-science/semantic-web-ontology/.

[5] JetBrains Marketplace. *RDF Plugins for JetBrains IDEs*. Accessed: 2025-03-05. 2024. URL: https://plugins.jetbrains.com/search?excludeTags=internal&products=androidstudio&products=appcode&products=aqua&products=clion&products=dataspell&products=dbe&products=fleet&products=go&products=idea&products=idea_ce&products=mps&products=phpstorm&products=pycharm&products=pycharm_ce&products=rider&products=ruby&products=rust&products=webstorm&products=writerside&search=RDF.

[6] *ldf.fi/*. https://www.sciencedirect.com/topics/computer-science/semantic-web-ontology/.

[7] Yohan Raynaud et al. "GAIA: An OWL-Based Generic RDF Instance Generator". In: *The Semantic Web: ESWC 2014 Satellite Events*. Vol. 8798. Lecture Notes in Computer Science. Springer, 2014, pp. 114–119. DOI: 10.1007/978-3-319-11955-7_16.

[8] *rdflib*. https://rdflib.readthedocs.io/en/stable/. rdflib.

[9] Lucien Rebuffel. *rdf-preview*. 2022. URL: https://github.com/LucienRbl/rdf-preview (visited on 04/17/2025).

[10] *Resource Description Framework*. https://www.w3.org/RDF/. W3C.

[11] *Semantic Web Ontology*. https://www.sciencedirect.com/topics/computer-science/semantic-web-ontology/. Science Direct.

[12] *SPARQL*. https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/. W3C.

[13] Stack Overflow. *Stack Overflow Developer Survey 2024*. Accessed: 2025-03-05. 2024. URL: https://survey.stackoverflow.co/2024/technology/.

[14] *The Semantic Web*. https://www.researchgate.net/profile/Michel-Klein/publication/2809069_The_semantic_web_the_roles_of_XML_and_RDF/links/0912f50ba15cd91ff5000000/The-semantic-web-the-roles-of-XML-and-RDF.pdf?origin=publication_detail&_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9u__cf_chl_tk=5MWh4tG1PQ1AeHjb.jhHi7j2wcKMsMturHDH14EGG80-1740929090-1.0.1.1-paRQqVpRTjoS9JAcvW33aSA_RLAce6qf9gO7K_sJWzc. INTERNET COMPUTI.

[15] Visual Studio Code Marketplace. *RDF Plugins for VS Code*. Accessed: 2025-03-05. 2024. URL: https://marketplace.visualstudio.com/search?term=RDF&target=VSCode&category=All%20categories&sortBy=Relevance.

[16] *World Wide Web*. hhttps://info.cern.ch/hypertext/WWW/TheProject.html. CERN.

Acknowledgment

Since with a thesis usually a personal phase of life ends, it is customary to end the work with an acknowledgment. But it should not be phrased too pathetically, that it seems exaggerated.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

type of thesis

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This thesis has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.

Author

.....
Author

.....
Author 2

Selbstständigkeitserklärung

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende XXX selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

type of thesis

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Autor

.....
Autor

.....
Autor 2