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 [15].
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 [13].

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 al-

gorithm 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 [8].
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

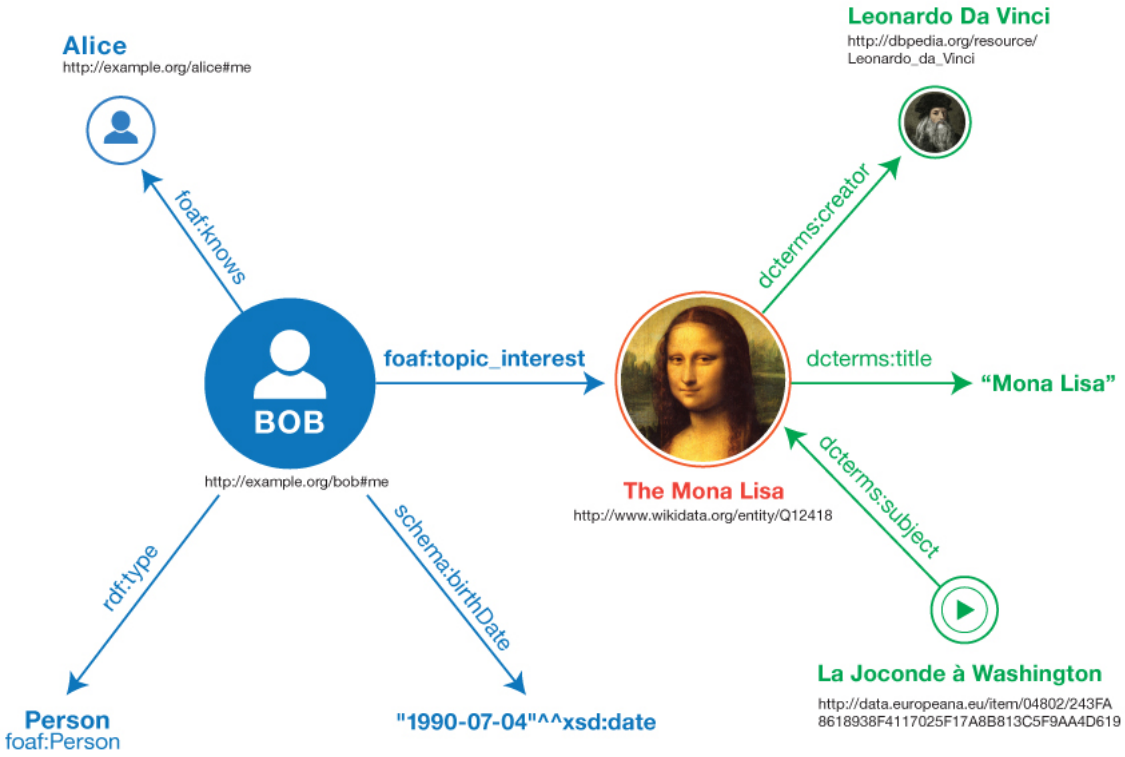

Figure 1.1: Example of RDF Graph

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" [10].

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 [14, 5, 12] (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.

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

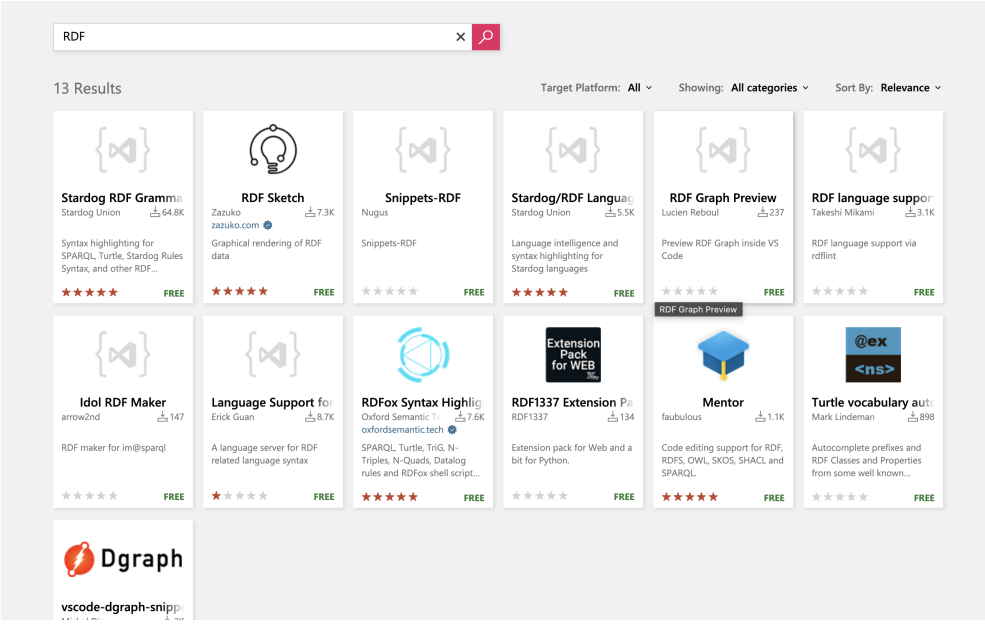

Figure 1.2: Limited Support: A Search for 'RDF' in the Plugin Marketplace Yields Only 13 Results, Highlighting the Scarcity of RDF-Specific Extensions for Developers.

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.

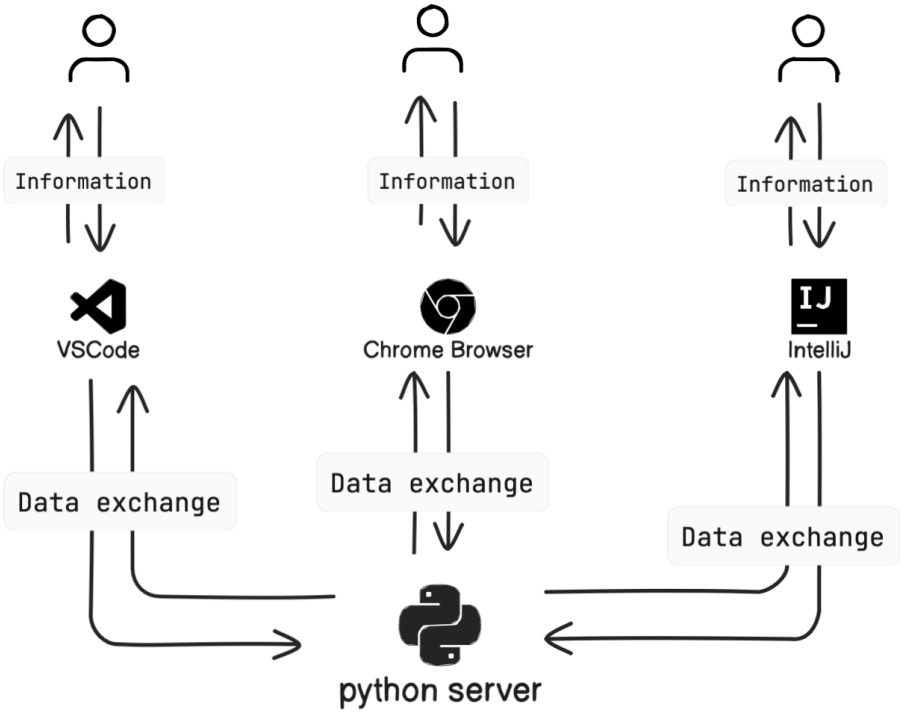

Figure 1.3: System Architecture: Information Flow and Data Exchange between Users, Development Environments (VSCode, IntelliJ), Chrome Browser, and a Python Server.

1.4 Structure of the Work

This thesis is separated into 7 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 ??: 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 and Limitations summarizes the thesis, describes the problems that occurred.

Chapter 7: Outlook Discusses the future of the project, and the possible improvements that can be done, like the integration of an AI model and the support for further ontologies.

Chapter 2

State of the Art

This section 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 [8].

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 [8].

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 [8].
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 [8].

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 [11].

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

2.2 Related Work

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

2.2.1 Comparison of Technologies

Name	Vendor	Release Year	Platform
A	Microsoft	2000	Windows
B	Yahoo!	2003	Windows, Mac OS
C	Apple	2005	Mac OS
D	Google	2005	Windows, Linux, Mac OS

Table 2.2: Comparison of technologies

2.3 Standardization

This sections outlines standardization approaches regarding X.

2.3.1 Internet Engineering Task Force

The IETF defines SIP as '...' [9]

2.3.2 International Telecommunication Union

Lorem Ipsum...

2.3.3 3GPP

Lorem Ipsum...

2.3.4 Open Mobile Alliance

Lorem Ipsum...

2.4 Concurrent Approaches

There are lots of people who tried to implement Component X. The most relevant are ...

Chapter 3

Consept

This section determines 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 requirements are defined:

- Functional requirements, for core features like, instance generation, schema validation, and visualization.
- Non-functional requirements, to take into account performance, usability and compatibility.
- Technical requirements, specified for underlying technologies and system design.
- Data privacy and Security, useful to ensure the security and privacy 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 will seamlessly integrate in their workflow.

3.2 Functional requirements

In this section covers the functional requirements for the entire system.

3.2.1 Instance Generation

The very first requirement for this project is the generation of synthetic RDF instances. When the user is writing an RDF file, in turtle, N3, XML or other RDF formats, it will generate some synthetic RDF instances that could save him a lot of time avoiding the insertion of testing data.
The application will not just generate compatible triples instances, but also allow the user 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 conforms to syntactic constraints (e.g., proper use of IRIs, literals, and blank nodes) and the asserted relationships lead to any logical contradictions or unexpected inferences.
In case of errors in the schema description, feedback should be provided to the user, via 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 their visualization as graphs.

3.2.4 Import and Export

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

3.2.5 Schema Visualization

To simplify the exploration and analyzes of an RDF from a human eye, the program should be able to display the described RDF schema with its graphical representation. The displayed graph should have some interactive features like zooming, panning and node collapse.
The graphical representation should adapt based on the 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 their working on.

3.3 Non-functional requirements

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

3.3.1 Performance

The system should ensure high performance rendering of large RDF graphs to facilitate seamless visualization and interaction. Additionally, operations such as node selection and zooming should be executed with minimal latency to maintain a responsive user experience. The system should optimize rendering pipelines to support real-time exploration of complex RDF schemas without compromising efficiency.

3.3.2 Usability

Both the IDE extension and the web application should feature an intuitive and user-centric interface that enhances accessibility and minimizes the learning curve for users. To achieve this objective, the design should incorporate familiar UI elements or skeuomorphic design principles, ensuring a seamless and intuitive user experience.

3.3.3 Compatibility

To ensure seamless integration across various development environments, the web service should be designed with cross-platform compatibility, enabling effortless adoption in different IDEs with minimal modifications. This approach enhances interoperability, allowing the service to function efficiently within diverse software ecosystems while maintaining consistent performance and usability.
To validate this, minor features in a secondary IDE should be implemented.

3.4 Technical Requirements

3.4.1 Web Architecture

The system should be designed with a client-server architecture. There are no constraints on the specific technologies (e.g. SOAP, REST, gRPC), but it must follow the web principles and use Web technologies, such as HTML, CSS, JavaScript or WebAssembly, and protocols (e.g. HTTP, WebSockets).

3.4.2 IDE extension

The application should be accessible via a web-based client to ensure broad availability; however, primary emphasis should be placed on its 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 implement secure data transmission protocols, such as HTTPS, to protect sensitive information during transmission.

3.5.2 Confidentiality

Based on the current design, no information should be saved from the Business Logic Layer.
All data processing occurs in-memory, either on the client-side or server-side, ensuring that no persistent storage of user data takes place within the system. This ephemeral data handling approach significantly reduces the risk of data retention and unauthorized access. As a result, compliance requirements related to long-term data storage, such as those outlined in the General Data Protection Regulation (GDPR), are minimized.

3.6 Solution strategy

As stated previously in the requirements section, the system is designed with a client-server architecture, where the client and server play distinct roles in the overall functionality.

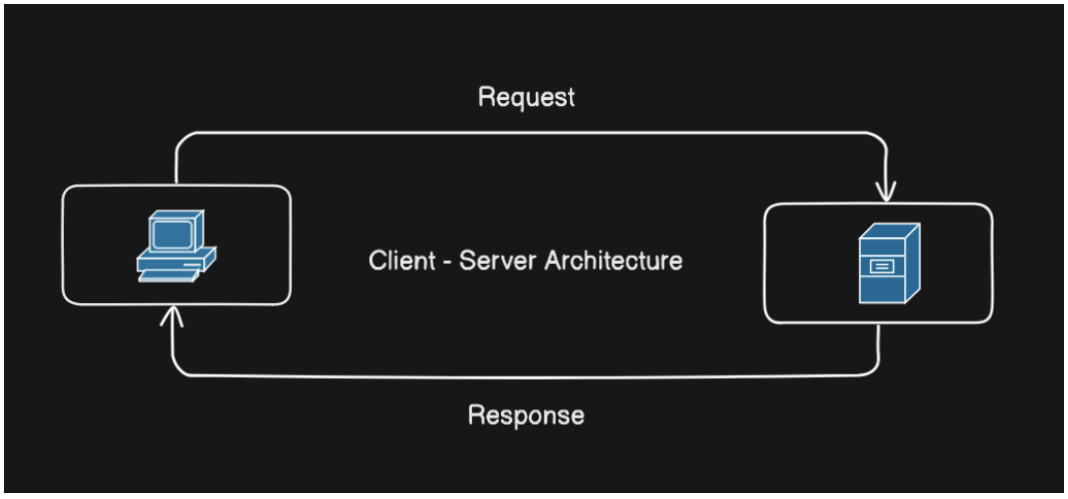

Figure 3.1: Client / Server

3.6.1 Client

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

- Browsers
- Integrated Development Environments

The Browser chosen by the user, should be able to access the web application via HTTP requests protocol. The user will then upload their RDF file and send it to the server. The Client will then receive the processed information and display the resulting graph (see Figure 3.2).

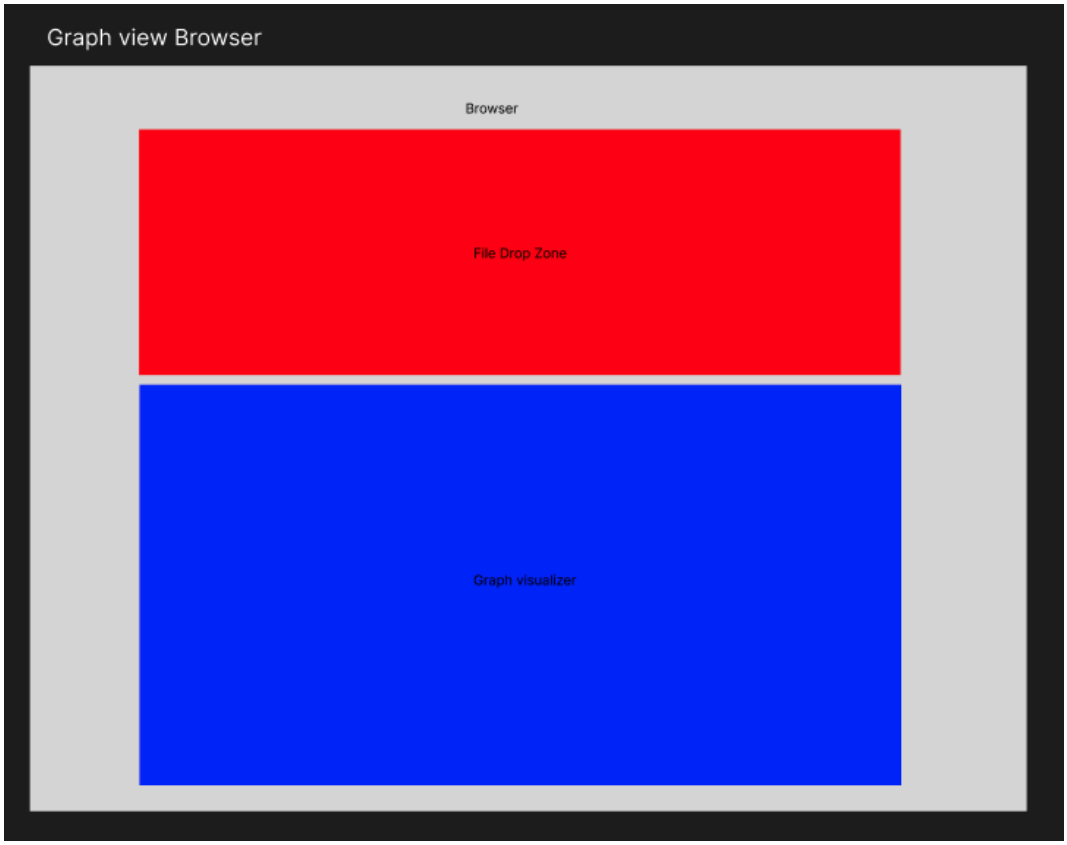

Figure 3.2: Browser file drop zone and graph view

In addition to this, the user should be able to read the RDF file with the generated instances next to its graphical representation (see Figure 3.3).

Within the IDE, the user experience UX will be slight different from the Browser one. Instead of manually uploading an RDF file, users will interact with the system directly within the IDE. By working on an RDF file and executing a predefined command, a web-based visualization panel of the file on focus will be dynamically launched as a side panel within the IDE. This panel will render an interactive graphical representation of the RDF schema in real time, allowing users to analyze relationships, structures, and dependencies without disrupting their coding environment (see Figure 3.4).

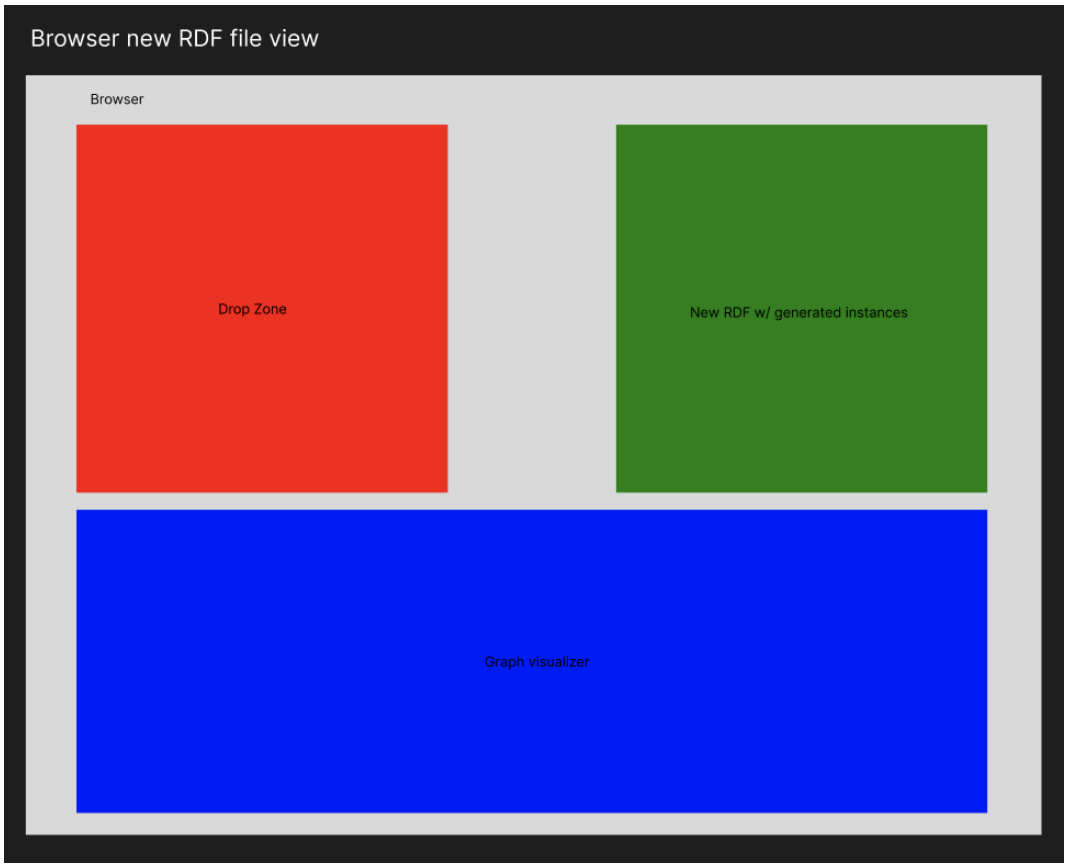

Figure 3.3: Browser new RDF side panel view

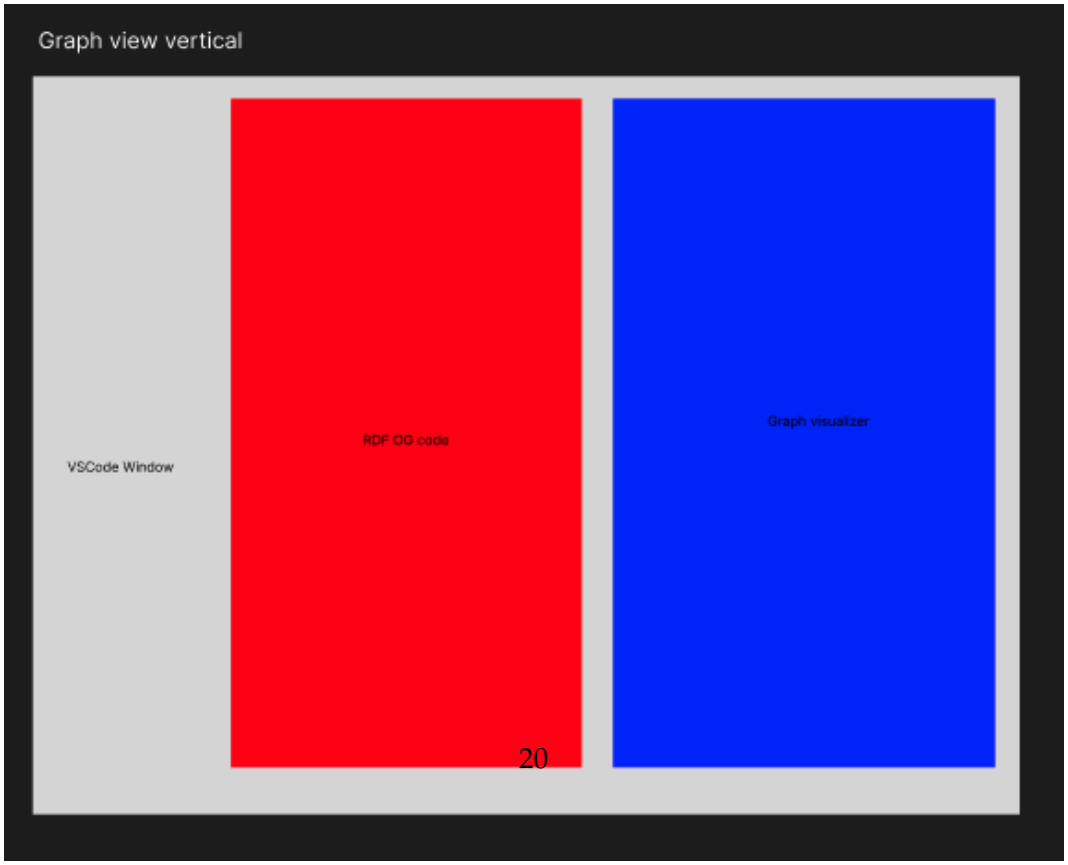

Figure 3.4: IDE Graph side panel view

The extension will not only support the visualization of RDF files in the format requested by the user, similar to its browser counterpart, but it will also enable real-time modification of the file (see Figure 3.5).

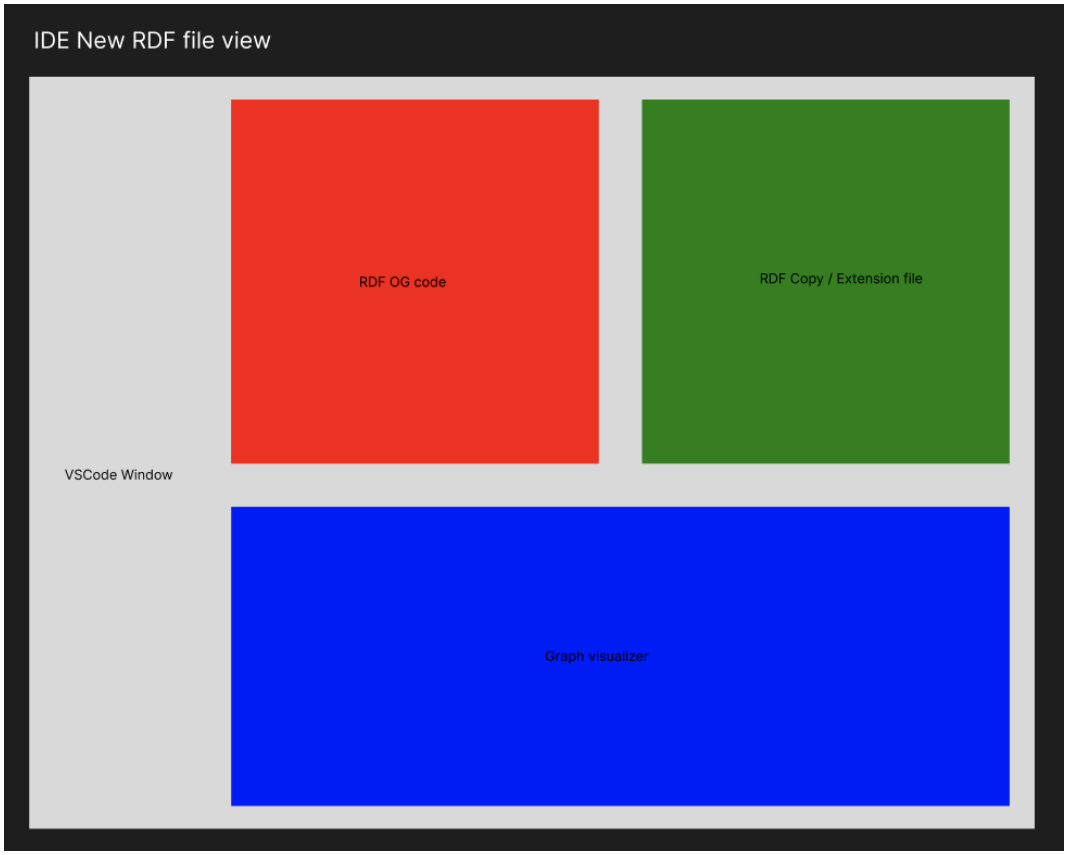

Figure 3.5: IDE new RDF side panel view

3.6.2 Server

To assure system intercompatibility across various device, Operative Systems, and User-Agents, the server is responsible for all computational tasks and data processing. By centralizing calculations and graph data manipulation on the server side, the server minimizes client-side resource consumption and enhances performance consistency.

The server place a crucial role in managing RDF data ensuring schema validation, instance generation, SPARQL query processing, and graph construction.
When the server receives an RDF file, it must scan the file to see potential floss, syntactic or semantic errors. It should be able to handle multiple RDF serialization formats such as Turtle, N3, XML, and JSON-LD.

The service generates synthetic RDF instances based on a given schema. It ensures logical consistency between the generated schema and the original one.
The system identifies classes and properties within an RDF schema and generates additional instances, where necessary, to maintain semantic correctness.
This is particularly important in cases where a property references an instance of a specific class that has not been explicitly defined in the dataset. For example, if a schema includes a hasCar property that requires an instance of the Car class, but no such instance exists, the system should automatically create one. This ensures that the schema remains logically consistent and that all property constraints are met. By dynamically generating required instances, the system helps maintain data integrity, prevents incomplete assertions, and improves the reliability of reasoning processes within RDF applications.
The additional generated instances have realistic properties and values in order to fulfill the graph constraints.

The service serves the new RDF file in the requested format with its graphical representation.

Chapter 4

Implementation

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

4.1 Environment

The following software, respectively operating systems, were used for the implementation:

4.1.1 Software and development tools

- MacOS Sequoia: used during the entire development process as main Operative System.
- Windows 11: used to test the extension for VSCode and IntelliJ installed in a Windows environment.
- Visual Studio Code: used to develop the web service and the VSCode extension.
- IntelliJ IDEA: used for the development of 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 primary programming language and package management for the VSCode extension.

- Kotlin and Gradle: used for the development and packaging of the IntelliJ IDEA extension.
- Python and pip: used for the implementation and package management of the web service.
- FastAPI: main Python library for the creation of a REST web service.
- rdflib: Python library for handling and processing RDF.
- sparqlwrapper: library that simplifies the use of SPARQL in Python.
- vis.js: JavaScript library for creating interactive RDF graphs.

4.2 Project Structure

The implementation is separated into 3 distinguished projects as depicted in figure 4.1.

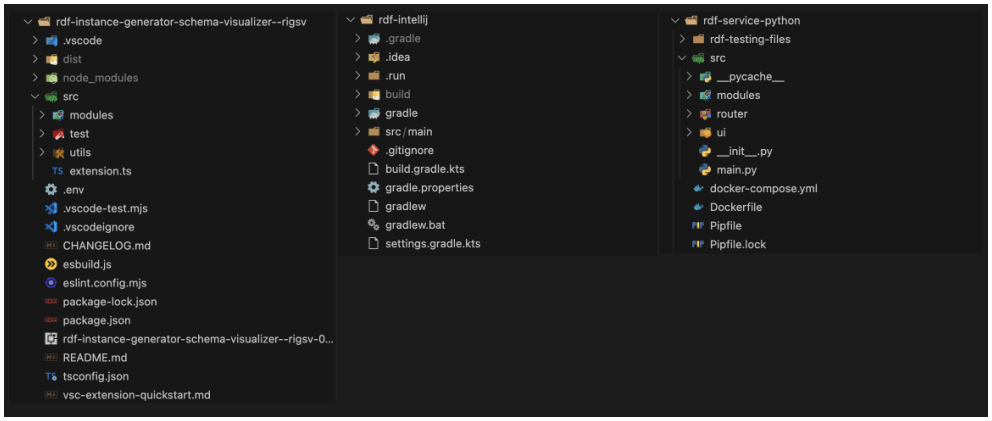

Figure 4.1: Project Structure

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

4.3 Implementation of the Web Service

The following section describes in details the implementation of the web service with all its key components.

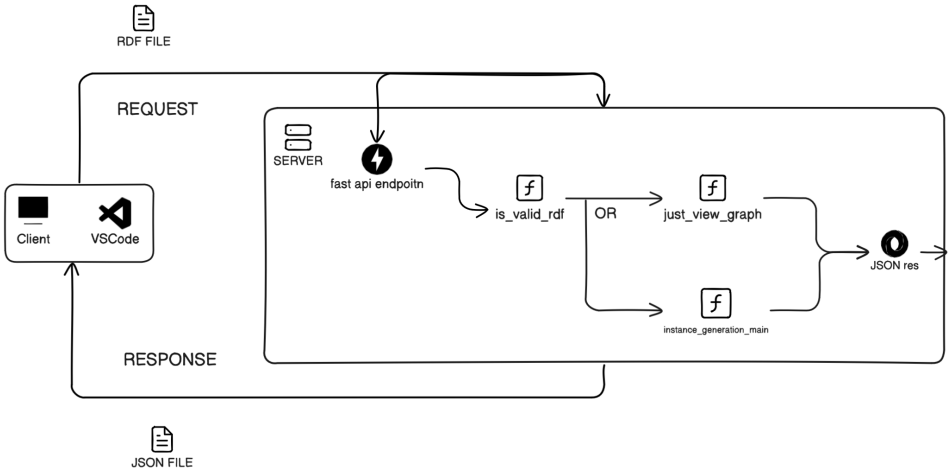

Figure 4.2: Server Schema

4.3.1 Server

There are many Python libraries that allow to create a REST web service. Among them, FastAPI was the chosen one. It is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints [2].
The first step, in order to start the server, was to install FastAPI and its dependencies. After than, by simplying initializing the app object, **app = FastAPI()** and execute the **fastapi dev src/main.py** command, the server will be up and running.

The next step was to create the endpoints. In src/router/router.py, the **router = APIRouter()** was initialized and three endpoints where created: to for the Browser Client and one for the extensions.

The first endpoint, $router.get("/", response\_class = HTMLResponse)$, is simply used to server the index.html file to the web Browser. This page allows the upload of RDF (POST to the next endpoint), visualize the generated graph and see the RDF file with the new generated instances.

The next endpoint is the $router.post("/generate", tags = ["users"])$. This endpoint accepts incoming POST requests from the Web Browser with the incoming RDF file. It returns the required data in a JSON format.

The last endpoint, $router.post("/", response\_class = JSONResponse)$, accepts incoming POST requests from the VSCode and IntelliJ extensions. It returns data in a JSON format.

4.3.2 RDF Validation

When an endpoint function receive an RDF file, the first function to be called is the $is\_valid\_rdf(file)$. This asynchronous function is responsible for validating the incoming RDF file. First it extracts the file extension from the incoming file. The extension name must be present in the predefined dictionary. If it's not there, an error is returned otherwise it proceed to the next step.
To see if the code in the file is correct syntactically and semantically, the function uses the **rdflib** library. This function allows to parsing and serialization of RDF data, storing of RDF triples, accessing to remote SPARQL endpoints, and working with RDF graphs [7]. The core of this function relies heavily on the $Graph.parse()$ method. If the file is syntactically correct and a new RDF graph is generated, then it means that the RDF is valid and the function returns true. Otherwise, it returns false.

4.3.3 Graph generator

After verifying the file is valid, the next step was to create the actual graph generation function. The $getGraph()$ uses the previously stated method for parsing the RDF file and return the graph, the file extension name and file format name to the $just\_view\_graph()$ function.
The retrieved graph is then serialized in the requested format and in json-ld.

4.3.4 Instance and Graph generator

This was the most important and complex functionality of the entire project. Like it has been done in the previous function, the graph, the format and the extension name are returned from the $getGraph()$. Then a new graph is initialized and, to keep things consistent and readable, all the namespace prefixes are copied from the initial retrieved graph (or_graph). This helps make the RDF output cleaner and easier to read, since it avoids repeating long URIs and keeps the format consistent.

The next step is the scanning of the RDF graph. The $scan()$ function goes through the entire graph and extract classes and properties. At the start, the $detect_classes()$ function extracts the classes from the graph. Then, the $detect\_properties()$ extracts all the properties and associate their domains and ranges with classes. In conclusion, it ensures that all the classes indirectly referenced are included in the graph.

The code then splits and creates two different functionalities: instance generation without properties search and instance generation with properties search.

In RDF vocabularies, classes can be either explicitly declared (e.g. $rdf : typerdfs : Class$) or implicitly referred with their usages in properties definitions (e.g. $rdfs : rangeschema : Car$). In the scenario where the instance generation is executed without the properties search, the code needs to generate and instance for every single class that has been found in the code, both explicitly or implicitly referred.
The $generate\_instance()$ function, take as parameters the class uri that has been extracted previously, the new graph initialized before, the number of instances that the user wants to generate for each declared class and the properties that has been defined. It's important to sate that the user has no control over the number of instances that come from a implicitly referred class.

Once the parameters are set, the function starts by checking i the class uri starts with **http://www.w3.org/2001/XMLSchema** and in case of a positive answer it skips the instance generation for that class.

After that, the $processed\_instances$ and $initialized\_instances$ are declared. The first, since $generate\_instance$ is a recursive function, is used for tracking which instances have already been generated to avoid duplicates. The second tracks which properties have been assigned to each generated instance.

In the next step, the function iterates N times to create multiples instances for the provided class. For each iteration, a new URI is created by using the EX prefix, the class name, and the number of the iteration (e.g. $ex : Employee\_Instance1$). If the instance has already been processed, it is simply reused from $processed\_instances$. The newly created instance is then linked to its class uri and added to the graph.

If the class has any associated property, the function checks every property and handles three main cases:

- If the provided value is a URI and it is a primitive datatype, thanks to the support of a hashmap (i.e. XSD) containing the primitive types with example values, a new literal is generated and appended to the graph. If the URI is not primitive, it is treated as a reference to another class. The function calls itself recursively to generate the required number of sub-instances for the referenced class. The resulting sub-instance is then added the graph.
- If the value is a XSD datatype URI as string, a default string literal (i.e., "default"8sd:string) is generated and the triple added to the graph.
- If the value is already a literal it is directly added to the graph.

- If the value is unsupported, an error is thrown.

Each successful initialized property, is then stored inside *initialized_instances*. At the end of the loop, the function adds the generated instance to the instances list and concludes with its return.

```
EX = Namespace("http://example.org/")

def generate_instance(
  class_uri,
  graph,
  num_instances=1,
  property_definitions=None,
  processed_instances=None,
  initialized_instances=None,
):
  from rdflib.namespace import XSD

  xsd = {
    "http://www.w3.org/2001/XMLSchema#integer": 0,
    "http://www.w3.org/2001/XMLSchema#decimal": 0.0,
    "http://www.w3.org/2001/XMLSchema#float": 0.0,
    "http://www.w3.org/2001/XMLSchema#double": 0.0,
    "http://www.w3.org/2001/XMLSchema#string": "unknown",
    ...
  }

  # Skip primitive XSD types
  if str(class_uri).startswith("http://www.w3.org/2001/XMLSchema#"):
    return []

  instances = []
  processed_instances = processed_instances or {}
  initialized_instances = initialized_instances or {}

  class_name = class_uri.split("/")[-1]

  for i in range(num_instances):
    instance_uri = EX[f"{class_name}_Instance{i + 1}"]

    if instance_uri in processed_instances:
      instances.append(instance_uri)
      continue

    processed_instances[instance_uri] = True
```

```
      graph.add((instance_uri, RDF.type, class_uri))

      if property_definitions and class_uri in property_definitions:
        for prop, value in property_definitions[class_uri].items():
          if isinstance(value, URIRef):
            if str(value).startswith("http://www.w3.org/2001/XMLSchema#
    "):
              graph.add((instance_uri, prop,
                     Literal(xsd[str(value)], datatype=value)))
            else:
              sub_instances = generate_instance(
                value,
                graph,
                num_instances=i + 1,
                property_definitions=property_definitions,
                processed_instances=processed_instances,
                initialized_instances=initialized_instances
              )
              if sub_instances:
                value = sub_instances[i]
                graph.add((instance_uri, prop, value))
          elif str(value) in xsd:
            graph.add((instance_uri, prop,
                   Literal(xsd[str(value)], datatype=URIRef(value))))
          elif isinstance(value, Literal):
            graph.add((instance_uri, prop, value))
          else:
            raise ValueError(
              f"Unsupported value type: {type(value)} for property {
    prop}"
            )

          if instance_uri not in initialized_instances:
            initialized_instances[instance_uri] = set()
          initialized_instances[instance_uri].add(prop)

      instances.append(instance_uri)

     return instances
```

Listing 4.1: Instance Generation Function

In the scenario where the instance generation is executed with the properties search, some previous steps are required before starting the instance generation. After the classes' detection, the $find\_properties()$ function is executed. It starts with retrieving all the

implicitly referred classes from the list. The function then iterates over the list of URIs and checks if the URI is present in the $vocab$ map (it contains a map of the most common RDF vocabularies). Each match is then the passed to the $query()$ function.

The $query()$ function is responsible for executing the SPARQL query on the LOV (Linked Open Vocabularies) Endpoint and returning the results. It takes three parameters: the class term for which we want to find some properties, the number of properties to retrieve, and the ontology to search in.
The first step of this function is to build up the SPARQL query for the required ontology (i.e. $fetchQuery()$). Then the query can be executed thanks to the sparqlwrapper library. The results are then polished, filtered and bound to the search class term. At the end, the function randomize the selection of the retrieved property.
In summary, this function helps make automatically generated RDF instances more meaningful by selecting properties that best fit the context of each instance—even when the class hasn't been explicitly defined in the RDF schema.

Find properties will then return the result that will be used by the $update\_propsS()$ function to update the property values passed to the $generate\_instance()$ function.

As the final step, the original graph (obtained from the uploaded RDF file) is merged with the newly generated graph. The combined graph is then returned in a JSON response, along with the corresponding file name and its JSON-LD representation.

```
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 4.2: Main Function for RDF Instance Generation

4.4 Implementation of the Browser Application

The following section describes the implementation of the browser application with focus on the code that runs client side.

4.4.1 The Fetch

When the user wants to use the RDF Instance Generator and Visualizer, the fastest way is by doing via a web browser.
By simply getting the default route (/) of the server, an HTML page is retrieved. This page allows to upload a RDF file to the server (/*generate* route). Upon a successful upload, the

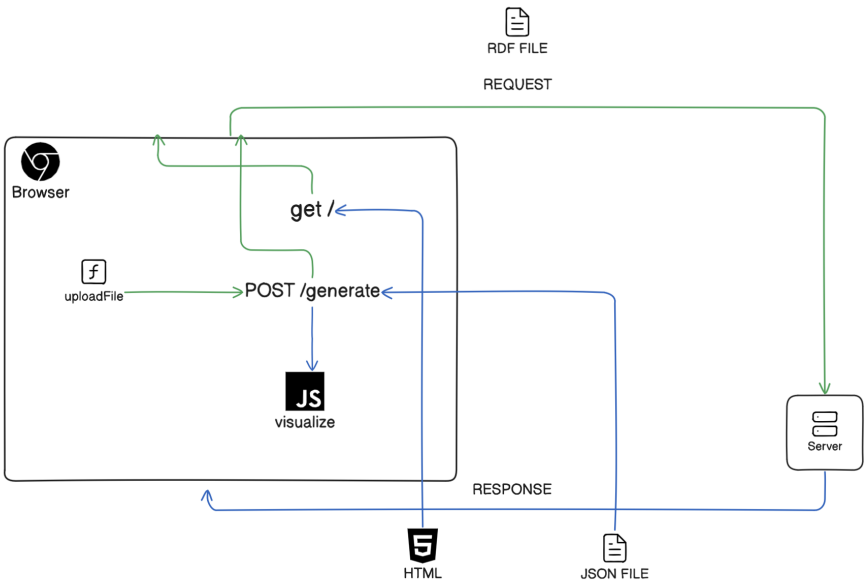

Figure 4.3: Web extension

page allows to read the new generated RDF file and renders the json-dl graph, thanks to the support of the vis.js library. The following describes the pipeline for the conversion from json-dl into an interactive RDF graph.

4.4.2 Graph Rendering

When the *index.html* file renders, it loads two main javascript files: the *uploader.js* and the *visualizer.js*. The first one, generate a web component (the *TurtleFileUploader*), that allows to upload a file and make POST requests. Upon receiving a positive response, the component injects the json-dl data into the custom rdf-visualizer as jsondata attribute. Furthermore, the response data (*res.data*), containing the new RDF file with the new generated instance is then attached to the DOM, so the user can inspect and download the upgraded version of its RDF.

```
1  class TurtleFileUploader extends HTMLElement {
2    constructor() { ... }
3
4    connectedCallback() { ... }
5
6    async uploadFile(edit = false, search = false) {
7    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 4.3: `TurtleFileUploader` component

The second, the rdf-visualizer component, listen to this attribute and triggers a parsing function to safely deserialize the data. To ensure coherence and render compatibility, 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 semantic to a network compatible logic is done by the *processJsonLd* function. Every RDF subject is converted into a node and the classified as a Class (rdfs:Class rendered as orange boxes), a Property (rdf:Property represented as green-yellow diamonds), an Instance (URI-based entities shown as blue dots) or a Literal (visualized as yellow box).

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

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

  // Type classification
  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';
  }

  // Subject node creation
  if (!nodesMap[subjectId]) {
    let label = item["...#label"] ? item["...#label"][0]["@value"] :
shorten(subjectId, dynamicPrefixes);
    let shape = nodeType === 'class' ? 'box' : nodeType === 'property
' ? 'diamond' : 'dot';
    let color = nodeType === 'class' ? '#FFA500' : nodeType === '
property' ? '#ADFF2F' : '#97C2FC';
    nodesMap[subjectId] = { id: subjectId, label, shape, color,
baseColor: color, nodeType, font: { color: "#000000" } };
  }

  // Type edge creation
  if (item['@type']) {
```

```
27        types.forEach(t => {
28        let typeId = typeof t === 'string' ? t : t['@id'];
29        if (typeId && !nodesMap[typeId]) {
30          nodesMap[typeId] = { id: typeId, label: shorten(typeId,
      dynamicPrefixes), shape: 'box', color: '#FFA500', ... };
31        }
32        edges.push({ from: subjectId, to: typeId, label: shorten("...#
      type", dynamicPrefixes), dashes: true, color: { color: '#000' } });
33        });
34      }
35
36      // Property edges and literal nodes
37      Object.keys(item).forEach(prop => {
38        if (prop === '@id' || prop === '@type') return;
39        item[prop].forEach(val => {
40        if (val['@id']) {
41          if (!nodesMap[val['@id']]) nodesMap[val['@id']] = { id: val['
      @id'], label: shorten(val['@id'], dynamicPrefixes), shape: 'dot',
      ... };
42          edges.push({ from: subjectId, to: val['@id'], label: shorten(
      prop, dynamicPrefixes), arrows: 'to', ... });
43        } else if (val['@value']) {
44          const literalId = `literal_${literalCounter++}`;
45          nodesMap[literalId] = { id: literalId, label: String(val['
      @value']), shape: 'box', color: '#FFD700', ... };
46          edges.push({ from: subjectId, to: literalId, label: shorten(
      prop, dynamicPrefixes), arrows: 'to', ... });
47        }
48        });
49      });
50      });
51
52      return { nodes: Object.values(nodesMap), edges };
53    }
54
```

Listing 4.4: `processJsonLd` function used for graph construction

Each RDF triple is mapped to an edge between subject and object, labeled by the predicate. Type relationships (rdf:type) are highlighted with dashed edges to differentiate them from standard property connections.

After the nodes and edges definition, they are initialized into a vis-network object, the vis algorithm partially distribute the nodes onto the canvas. The graph is then stabi-

lized to enhance readability and explorability. To improve furthermore the user experience, when a user clicks on a node, all the non-neighboring nodes are blurred out.

4.5 Implementation of the Visual Studio Code Extension

Visual Studio Code, initially intended as lightweight source code editor, has now evolved into a fully functional IDE thanks to its extensive ecosystem of plugins.
Since VSCode it's built on top of Electron, a framework for building desktop applications using web technologies, it's possible to build extensions with the traditional web tolls (HTML, CSS, JavaScript).
The following extension is built upon the Visual Studio Code Extension API, which provides a set of interfaces and classes to implement when working with the core capabilities of the editor.
The following section describes the implementation of the Visual Studio Code extension with all its key features.

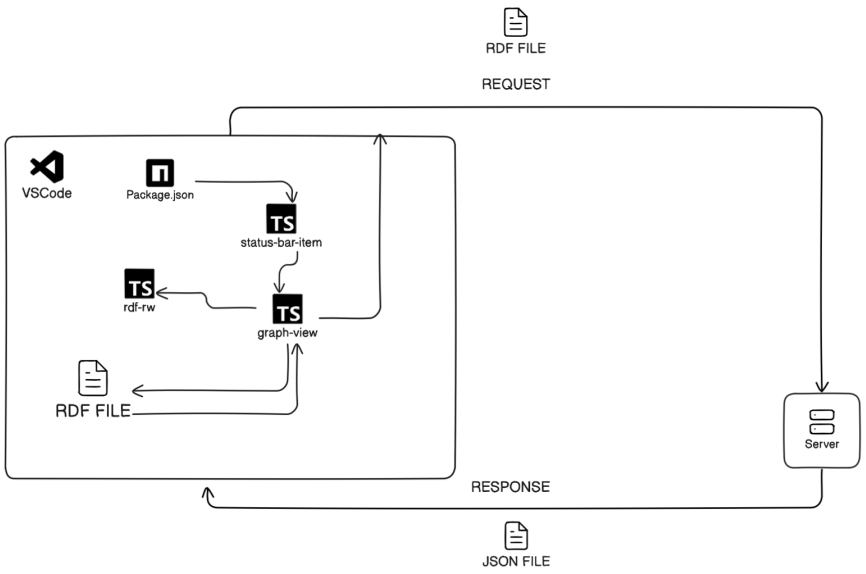

Figure 4.4: VSCode extension

4.5.1 Commands Architecture

Commands trigger actions in Visual Studio Code. In this case, commands have been used by the RIGVS extension to expose the following functionality to users: graph view ($extension.viewGraph$), simple instances generation and graph view ($extension.runGraph$) and instances generation with search properties and graph view ($extension.searchProperty$)

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

Every command in VSCode can be executed by simply pressing $cmd + shift + p$. After the input dialog window opens up, each command present in the $package.json$ 4.5 can be executed by typing the title and pressing the enter key.

The first command ($extension.viewGraph$) allows to open Webview inside the VSCode workspace. This webview renders the information coming from the server.
At the current project state, this command can be only executed while focusing the RDF file that needs to visualized as a graph. When the command is executed, a new panel ($vscode.window.createWebviewPanel$) is generated with specific options (side panel view and enabling scripts execution inside the panel). After this the $getWebviewContent$ is called.

The **getWebviewContent** function prepares the web view for the graph visualization. It executes the *sendRDFContent* function, that allows to extract data from the file focussed by the user (*getRDFContent*) and to send it to the server. After getting the response, static urls for the incoming scripts (*visualizer.js* and *uploader.js*) and then replaced in the html file.
Since VSCode Webviews are intended to load HTML content in a sandboxed environment, the default CSP (Content Security Policy) is restricting external resources like scripts and external stylesheets.
Including the following meta tag in the head of the html content returned by *getWebviewContent* is essential to implement a runtime Content Security Policy that permits script execution. The necessary adjustments are stored in the *csp* variable:

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

After the content is returned, it is rendered in the webview panel. The page also listed for 2 specifics clicks events: the **editButton** and the **exportGraph**. When the user clicks on the edit button, the *extension.editRDF* commands is triggered. This opens the new RDF graph in the VSCode editor, in the original RDF format.
The *extension.exportGraph* command is used to export the graph in a png format.

The second command (*extension.runGraph*) has the exact same features as the previous command. After its execution and additional input element appears as pop window and requires the amount of instances that need to be generated by the server.
The last one (*extension.searchProperty*) has the exact same behavior of the previous one, but with addiction of the properties search functionality.

```
1     export function openWebView(context: vscode.ExtensionContext) {
2       const viewGraph = vscode.commands.registerCommand(
3         "extension.viewGraph",
4         async () => {
5         vscode.window.showInformationMessage("Graph view is loaded");
6
7         let panel = vscode.window.createWebviewPanel(
8           "rdfVisualizer",
9           "RDF Visualizer",
10          vscode.ViewColumn.Beside,
11          {
12          enableScripts: true,
13          retainContextWhenHidden: false,
14          }
```

```
        );

        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 4.6: openWebView

At the current project state, this command can be only executed while focusing the RDF file that needs to be enhanced
To facility the access to these commands, a quick menu button has been design. The button locates at the button right corner of the VSCode window. By click the button a modal with a simple select allows the selection between these three commands.

4.6 Implementation of the IntelliJ IDEA plugin

The following section describes the implementation of the IntelliJ IDEA plugin with the showcase of the feature.

Like the VSCode extension, the goal of the IntelliJ IDEA plugin is to provide a user-friendly

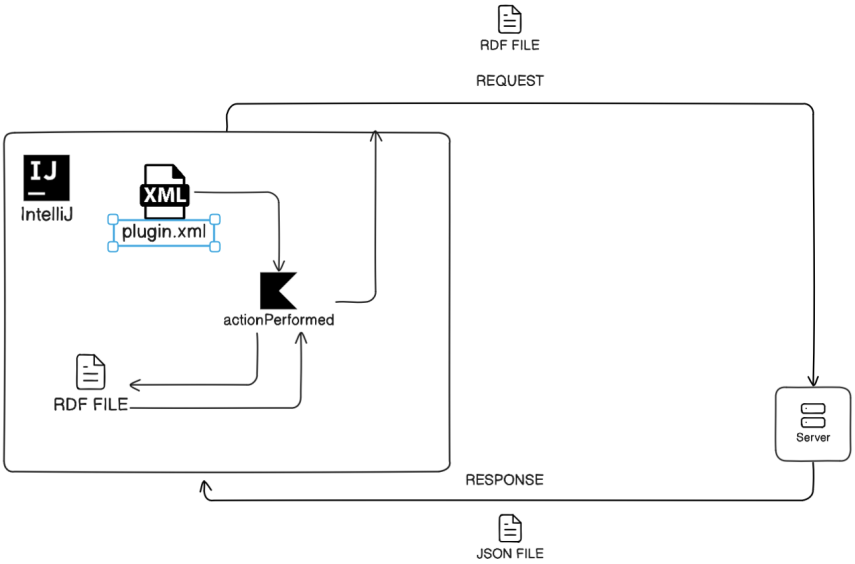

Figure 4.5: IntelliJ Extension

interface for generating RDF instances and visualizing them in a side panel as graph. This extension, developed mainly for the pourpuse of demonstrating the flexibility of this tool's architecture as a distributed system, only has implemented the feature responsible for rendering the RDF Graph in a side panel.

The extension is developed using Kotlin programming language, Glade plugin manager and the JetBrains Plugin DevKit.
Like the VSCode world, the IntelliJ IDEA plugin has **actions** (*commands* in VSCode) that allow to execute functions, based on users input.

```
<?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 4.7: IntelliJ Actions

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

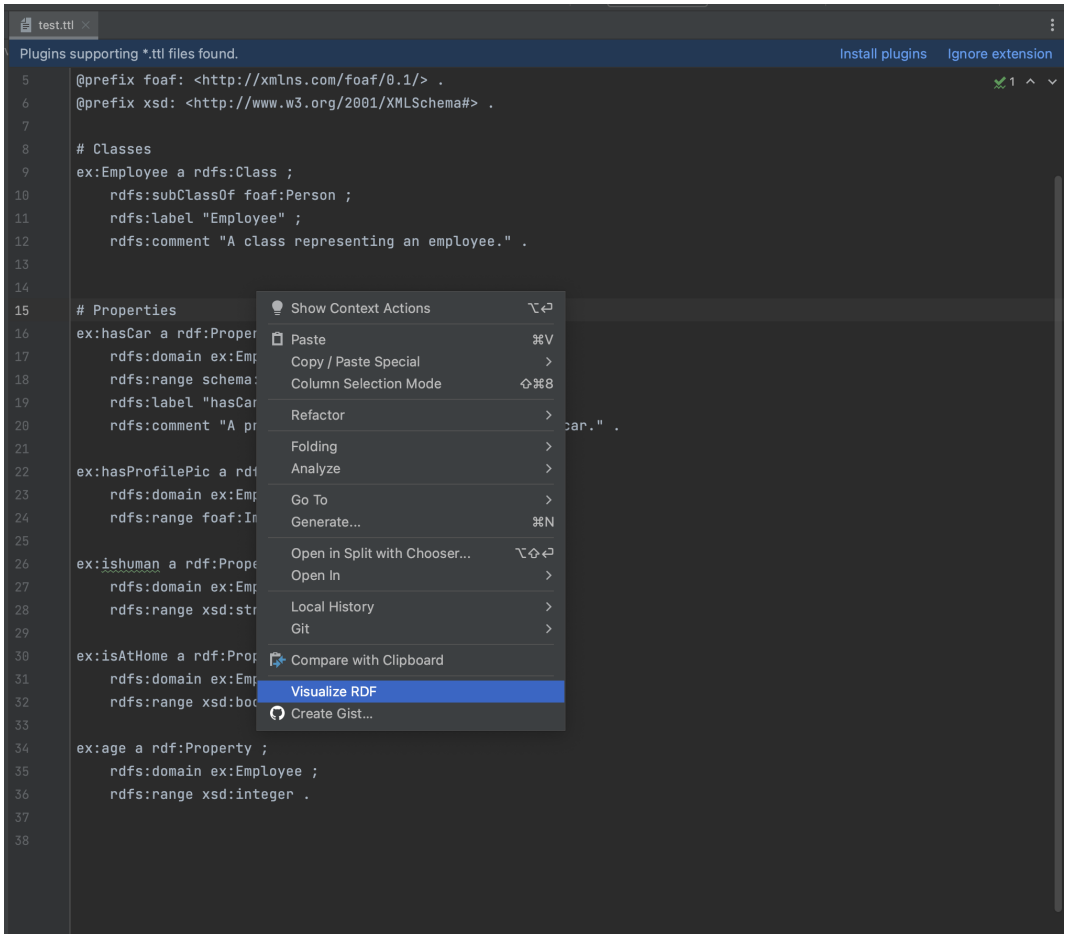

Figure 4.6: Action call in IntelliJ

By simply clicking on it the action gets triggered and executes the $actionPerformed$ function.
This function extracts the RDF scheme from the focused file, and wraps in a formData object. Then, it sends the relevant information to the python server. The response is then passed to the $showGraphViewer$ function, responsible for rendering the graph in a side panel. Just like in the VSCode extension, the html contet is processed to ensure that CSP restrictions are removed and resources can be called and rendered inside the Intellij Web view.

```
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 4.8: Render graph in WebView in IntelliJ

Chapter 5

Implementation

This chapter describes the implementation of component X. Three systems were chosen as reference implementations: a desktop version for Windows and Linux PCs, a Windows Mobile version for Pocket PCs and a mobile version based on Android.

5.1 Environment

The following software, respectively operating systems, were used for the implementation:

- Windows XP and Ubuntu 6
- Java Development Kit (JDK) 6 Update 10
- Eclipse Ganymede 3.4
- Standard Widget Toolkit 3.4

5.2 Project Structure

The implementation is separated into 2 distinguished eclipse projects as depicted in figure 5.1.

The following listing briefly describes the single packages of both projects in alphabetical order to give an overview of the implementation:

config
Lorem Ipsum...

server
Lorem Ipsum...

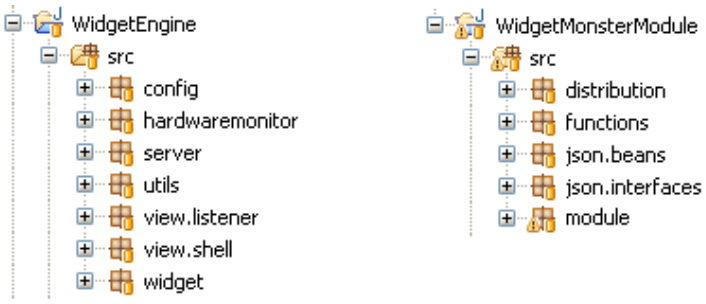

Figure 5.1: Project Structure

utils
Lorem Ipsum...

5.3 Important Implementation Aspects

Do not explain every class in detail. Give a short introduction about the modules or the eclipse projects. If you want to explain relevant code snippets use the 'lstlisting' tag of LaTeX. Put only short snippets into your thesis. Long listing should be part of the annex.

```
{
  id: 1,
  method: "myInstance.getGroup",
  params: ["Teammates", 2, true]
}

{
  id: 2,
  result: [
      "groupDesc":"These are my teammates",
      {
      "javaClass":"src.package.MemberClass",
      "memberName": "Bob",
      }
    ]
}
```

Listing 5.1: JSON String Code Snippet

You can also compare different approaches. Example: Since the implementation based on X failed I choosed to implement the same aspect based on Y. The new approach resulted in a much faster ...

5.4 Graphical User Interface

Lorem Ipsum...

5.5 Documentation

Lorem Ipsum...

Chapter 6

Evaluation

In this chapter the implementation of Component X is evaluated. An example instance was created for every service. The following chapter validates the component implemented in the previous chapter against the requirements.

Put some screenshots in this section! Map the requirements with your proposed solution. Compare it with related work. Why is your solution better than a concurrent approach from another organization?

6.1 Test Environment

Fraunhofer Institute FOKUS' Open IMS Playground was used as a test environment for the telecommunication services. The IMS Playground ...

6.2 Scalability

Lorem Ipsum

6.3 Usability

Lorem Ipsum

6.4 Performance Measurements

Lorem Ipsum

Chapter 7

Conclusion

The final chapter summarizes the thesis. The first subsection outlines the main ideas behind Component X and recapitulates the work steps. Issues that remained unsolved are then described. Finally the potential of the proposed solution and future work is surveyed in an outlook.

7.1 Summary

Explain what you did during the last 6 month on 1 or 2 pages!

The work done can be summarized into the following work steps

- Analysis of available technologies
- Selection of 3 relevant services for implementation
- Design and implementation of X on Windows
- Design and implementation of X on mobile devices
- Documentation based on X
- Evaluation of the proposed solution

7.2 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?

7.3 Problems Encountered

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

7.4 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] *rdflib*. https://rdflib.readthedocs.io/en/stable/. rdflib.

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

[9] J. Rosenberg et al. *SIP: Session Initiation Protocol*. RFC 3261 (Proposed Standard). Updated by RFCs 3265, 3853, 4320, 4916, 5393. June 2002. URL: http://www.ietf.org/rfc/rfc3261.txt.

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

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

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

[13] *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.

[14] 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.

[15] *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