Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Web application for ontology comparison

Lukáš Vévar

Supervisor: Ing. Petr Křemen, Ph.D.
Field of study: Software Engineering and Technology
May 2022

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vévar** Jméno: **Lukáš** Osobní číslo: **483767**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Webová aplikace pro porovnávání ontologií

Název bakalářské práce anglicky:

Web application for ontology comparison

Pokyny pro vypracování:

OWL ontologies are shared conceptual models of knowledge. The goal of the thesis is to design, implement and evaluate a web service and a web user interface for comparing OWL ontologies over an existing OWLDiff system.
1. Become familiar with OWL, ontology comparison techniques and the OWLDiff system (https://github.com/kbss-cvut/owldiff).
2. Design and implement a REST web service that allows comparing existing web ontologies, either present on the web or provided through file upload. The web service will be able to serve results in a suitable RDF diff format.
3. Design and implement a web application using the REST web service to mimic the comparison operations of current OWLDiff.
4. Evaluate the efficiency of the web service on selected existing ontologies from the Semantic Government Vocabulary (https://slovník.gov.cz/) and relate it to the performance of the standalone version of OWLDiff.
5. Evaluate the usability of the web application by a user study with several ontology designers.

Seznam doporučené literatury:

- P. Kremen, M. Smid and Z. Kouba, "OWLDiff: A Practical Tool for Comparison and Merge of OWL Ontologies," 2011 22nd International Workshop on Database and Expert Systems Applications, 2011, pp. 229-233, doi: 10.1109/DEXA.2011.62
- S. Abburu. A Survey on Ontology Reasoners and Comparison. International Journal of Computer Applications 57(17):33-39, November 2012.
- OWL 2 Web Ontology Language Primer, P. Hitzler, M. Krötzsch, B. Parsia, P. Patel-Schneider, S. Rudolph, Editors, W3C Recommendation, October 27, 2009, http://www.w3.org/TR/2009/REC-owl2-primer-20091027/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Petr Křemen, Ph.D. skupina znalostních softwarových systémů FEL

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

Datum zadání bakalářské práce: **07.02.2022** Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

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

III. PŘEVZETÍ ZADÁNÍ

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

.

Datum převzetí zadání Podpis studenta

 © VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank my supervisor, Ing. Petr Křemen, Ph.D., for guidance, great feedback and for the opportunity to write a thesis on this topic.

Declaration

I declare that this work is all my own work, and I have cited all sources I used in the bibliography.

Prague, May 15, 2022

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 15. května 2022

Abstract

Comparing two different ontologies is difficult task, since same knowledge in one ontology can be written differently in another ontology, but have same meaning. OWLDiff tool can compare ontologies and find differences between them using multiple algorithms, however OWLDiff has only a standalone application. The main goal of this thesis is design and implementation of a user interface and separate API, which will mediate all functionalities of OWLDiff. The result is a web application which allows ontology engineers compare or merge two different versions of the same ontology. Designed API runs in Java using framework Spring and user interface is built using React with help of prebuilt components from Material UI. Usability of the web application is tested in performance testing and with help of volunteering experienced ontology engineers in user testing.

Keywords: OWLDiff, Ontology comparison, Web application, Spring, SpringBoot, React, User testing

Supervisor: Ing. Petr Křemen, Ph.D.
Knowledge Based Software Systems Group,
Department of Computer Science,
FEE CTU

Abstrakt

Hledání rozdílů mezí ontologiemi je náročný problém, jelikož stejná znalost zachycená v ontologii může být zapsána různými způsoby. Nástroj OWLDiff umí nacházet rozdíly mezi ontologiemi pomocí různých algoritmů. Tento nástroj má ovšem pouze samostatnou aplikaci. Hlavním cílem této bakalářské práce je vytvoření návrhu a následná implementace uživatelského rozhraní a samostatného API, které bude zprostředkovávat všechny funkcionality OWLDiffu. Výsledkem je webová aplikace umožňující pracovníkům z ontologiemi ontologie porovnávat nebo mergovat. Navržené API běží v jazyce Java za pomocí frameworku Spring a uživatelské rozhraní je vytvořeno pomocí knihovny React s využitím komponent z frameworku Material UI. Použitelnost aplikace je ověřována výkonostním a uživatelském testováním ze strany zkušených odborníků s ontologiemi.

Klíčová slova: OWLDiff, Porovnání ontologií, Webová aplikace, Spring, SpringBoot, React, Uživatelské testování

Překlad názvu: Webová aplikace pro porovnávání ontologií

Contents

1 Introduction 1
1.1 Goal of the work 2
2 Web Ontology language 5
2.1 Syntax 5
2.2 Description logic and OWL2 Profiles 6
2.3 Axioms 7
3 OWLDiff 9
3.1 Diff algorithms 10
3.2 Merge 10
4 Analysis 13
4.1 Research of similar solutions 13
4.1.1 ecco 13
4.1.2 bubastis 14
4.1.3 ELH-forgetting 14
4.1.4 Protege 4 14
4.1.5 Research summary 14
4.2 Functional requirements 15
4.3 Non-functional requirements 16
5 Analysis of used technologies 17
5.1 Backend - Java 17
5.1.1 Spring 17
5.2 Frontend - Javascript 18
5.2.1 HTML + CSS 18
5.2.2 Typescript 18
5.2.3 React + JSX 18
5.2.4 NPM 18
5.2.5 Gatsby 19
5.2.6 Material UI 20
6 Solution design 21
6.1 Architecture introduction 21
6.2 Communication architecture 22
6.3 Backend architecture 22
6.3.1 Layers 23
6.3.2 Components 23
6.3.3 API architecture 24
6.4 Frontend design concepts 26
6.5 Design 28
7 Implementation 31
7.1 Integrating API into OWLDiff 31
7.1.1 Using algorithms 31
7.1.2 Axiom tree hierarchy as JSON 32
7.2 Implementing endpoints 33
7.3 Tree model on user interface 33
7.3.1 Encountered problem 34
7.4 Deployment 35
8 Testing 37
8.1 Performance testing 37
8.1.1 RAM usage 38
8.1.2 OWLDiff standalone comparison 38
8.2 User testing 39
8.2.1 Testing scenario 39
8.2.2 Asked questions 40
8.3 Test recap 41
9 Conclusion 43
9.1 Future work 43
Bibliography 45
A Acronyms 49
B User testing 51
C List of the attachments 55
C.1 Attached files 55
C.2 Github link 55

Figures

2.1 Example ontology in OWL2 RDF/XML document 6
2.2 Example ontology in form of RDF graph 7
2.3 OWL2 example of class A 8
2.4 Axioms of example class A 8
2.5 Axiom hierarchy tree example . . . 8

3.1 Ontology comparison result 11
3.2 Selection of axioms before merging 11

5.1 Example of simple React component with JSX HTML rendering 19

6.1 Solution design draft 22
6.2 Backend architecture 27
6.3 Design of main layout for user interface 29
6.4 Design after comparison is done and displayed 29
6.5 Design of merge window 30

7.1 Generated diagram of modules with their dependencies to each other 32
7.2 NodeModel visitor diagram 33
7.3 Basic expandable list, top level item can be expanded using arrow on the left. 34
7.4 TreeView component example . . 34
7.5 Visual explanation of how virtualization work (source [2]) . . . 35

Tables

4.1 Research of similar solutions . . . 15

5.1 Comparison of React environments 20

6.1 Comparison of API architectures 24

8.1 Performance testing results table 38
8.2 OWLDiff standalone performance comparison with created web application 39

B.1 Test scenario steps form ready to fill out 52
B.2 Results for all recipients from test scenario testing 53
B.3 Results for all recipients from question answering 54

Chapter 1

Introduction

Over the past decades, the Internet has grown exponentially. The amount of data created are in trillions of megabytes per day. It was impossible for search engines to handle so much information. That is where the idea of Semantic Web[1] came in. Semantic web defined tools and ways how to represent and structure data. Therefore, using these tools it was possible to turn plain text into knowledge.

Development of data science and artificial intelligence required to structure knowledge and give it some form. With the ability of Semantic Web to create Linked data[2] it was possible to build first knowledge graphs[6]. The knowledge graph represents a collection of entities (real-world objects), events, or abstract concepts (e.g., documents) and linked descriptions between them. In other words, every entity, event or concept inside the knowledge graph can have a description and this description contains not only plain text data but also relationships to other entities, events or concepts. Each relationship and data inside descriptions must be readable to both people and computers. This representation help understand data as knowledge.

The absolute basics of knowledge graphs are ontologies. As mentioned, to achieve readability and most importantly same understanding of knowledge for any user, where the user can be human or a software inside a computer, there is a need to universally specify a model. Ontology is a model for knowledge, which consists of concepts and relationships between those concepts. These concepts of knowledge must be formal and defined[9]. The main concepts are:

- **Classes** — general entities and objects (usually nouns) in the domain of interest. Similarities from object-oriented designs. Class hierarchy is allowed.

- **Relationship types** — all kinds of relationships that can exist between classes. Relationships are divided based on their purpose and meaning.

[1] *"The Semantic Web is a vision for the future of the Web in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web."* [9]

[2] Several technologies that together create standards for connecting data and create relationships between them[35]

- **Properties** — properties of classes that are used for better description of knowledge.

So far there is no single ontology that describes all the knowledge in the world. Instead ontologies are made withing domains or are made to specific use cases within an organizations. Ontologies are not only used for knowledge graphs, but almost anywhere where is need to improve knowledge management.

1.1 Goal of the work

Ontologies are created within some domain by ontology engineers, usually represented with OWL2 (more explained in chapter 2). The problem occurs though, when someone tries to make a new version of an existing ontology and wants to compare what knowledge is different in both versions. Even though ontologies represented with OWL2 are written in textual files, the classical diff comparison for textual files is not ideal. The reason being that the same knowledge can be written differently, also saying how is the knowledge different using just textual diff is time-consuming and requires people to have strong experience with ontologies. Also, OWL2 ontologies can be stored in different file types and still describe the same knowledge.

Therefore, there is need for logical comparison based on classes, relationships and properties from ontologies. Project OWLDiff (more in section 3) uses specific algorithms to compare two different versions of the same ontology. However, OWLDiff is not avaliable directly on the Internet in form of web application and users must download the standalone application, if they want to use this tool.

The main goal of this work is to extend OWLDiff project with a new module. This module will create REST API[3], which will provide comparison and merge functionality from OWLDiff as a web service. Next there will be implementation of a user interface - web application communicating with created API. User interface should be well-arranged and user-friendly. The created API should be able to serve all functions of OWLDiff and should be runnable even without the user interface.

Secondary goals are: researching of similar solutions, testing usability of created user interface, comparing performance with standalone OWLDiff application with testing of all kinds and sizes of OWL2 ontologies.

1. Research of similar projects focusing on ontology comparison accessible as web applications.
2. Design and implementation of REST API which will mediates all functions of OWLDiff.
3. Design and implementation of web user interface communicating with created REST API.

[3] Interface that will allow other application to communicate with.

4. Performance testing and defining limits of the created web application.
5. User testing of created web application and determine usability for ontology engineers.

Chapter 2

Web Ontology language

In order for ontologies to be created, a formal language have been developed. OWL2 is an extension for original Web Ontology language (OWL)[36] that supports many syntaxes. The core of OWL2 are RDF triples[1], so it is also possible to view any OWL2 document as an RDF graph. An example ontology of OWL2 document written in RDF/XML syntax (more in subsection 2.1) can be seen in figure 2.1. The following ontology describes a man named John and a woman named Mary being parents to another man named John jr. Knowledge has base in domain example.com. RDF graph[2] of the same ontology can be seen on figure 2.2

2.1 Syntax

Syntax[36] is a collection of symbols and letters that together create documents. OWL2 can be stored in different syntaxes. Only one is mandatory (RDF/XML), others are optional.

- RDF/XML — all OWL2 tools must be able to read this syntax. It allows mapping to RDF graphs.
- OWL/XML — ontology structure is written as XML file. It is used in XML-based tools.
- Functional — formal syntax for ontologies. It thoroughly maps profiles, classes and properties.
- Manchester — compressed and easily readable for people, however not suitable for all OWL2 ontologies.
- Turtle — ontology syntax with alternative RDF mapping.

[1] Basic model for decomposing information into following statement: subject - predicate - object[37]

[2] The graph has been made using GraphViz online tool[8]

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

    <owl:Class rdf:about="#Woman">
    </owl:Class>
    <owl:Class rdf:about="#Man">
    </owl:Class>

    <owl:ObjectProperty rdf:about="#hasChild">
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:about="#hasSpouse">
    </owl:ObjectProperty>

    <rdf:Description rdf:about="#John">
      <rdf:type rdf:resource="#Man"/>
      <hasSpouse rdf:resource="#Mary"/>
      <hasChild rdf:resource="#John_jr"/>
    </rdf:Description>

    <rdf:Description rdf:about="#Mary">
      <rdf:type rdf:resource="#Woman"/>
      <hasSpouse rdf:resource="#John"/>
      <hasChild rdf:resource="#John_jr"/>
    </rdf:Description>

    <rdf:Description rdf:about="#John_jr">
      <rdf:type rdf:resource="#Man"/>
    </rdf:Description>

</rdf:RDF>
```

Figure 2.1: Example ontology in OWL2 RDF/XML document

2.2 Description logic and OWL2 Profiles

OWL and OWL2 ontologies are based on description logic[12]. All the reasoning operations are defined using description logic. For example, basic reasoning operations are: checking if knowledge is correct, checking if knowledge is equal or redundant to another and checking if knowledge has meaning. There are logic reasoning programs and tools which help to work with ontologies using description logic.

It is not the aim of this work to go into details of this topic. However, it is important to note three different OWL2 Profiles[36], that are defined using description logic. Each profile brings advantages for different kinds of applications.

- OWL 2 EL — useful for large applications with huge amount of classes with big amount of properties. EL provides ability to handle existential restrictions.[12]
- OWL 2 QL — should be used when ontologies are not too complex, but are used to structure huge amount of data. It decreases query[3] execution

[3] Query is a command that is run over a knowledge graph to get data from it. Time to

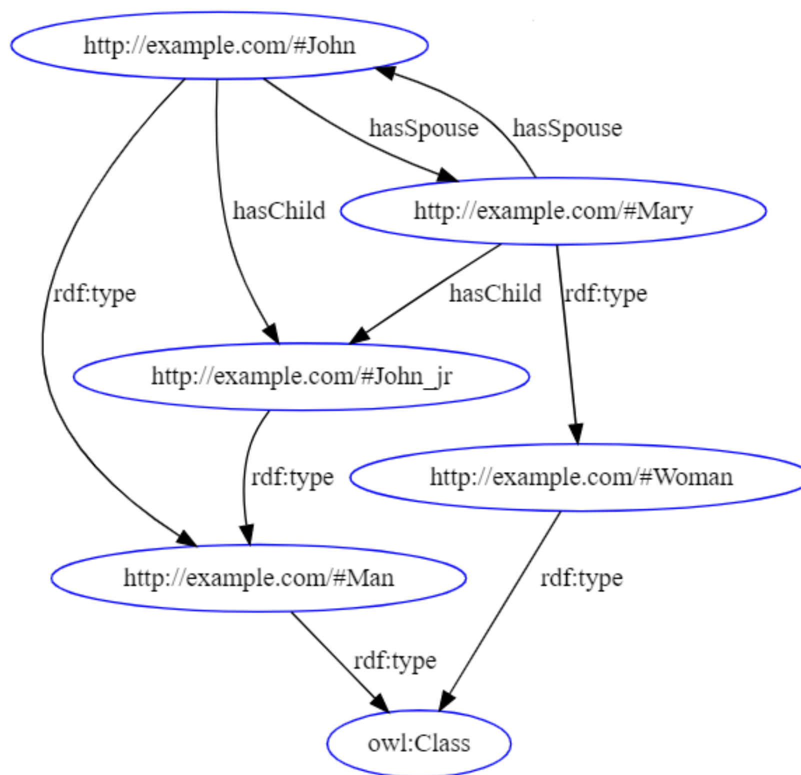

Figure 2.2: Example ontology in form of RDF graph

time.

- OWL 2 DL — majority of ontologies use this profile. It supports all reasoning operations. Includes all OWL constructs however, with some restrictions.

2.3 Axioms

OWL2 ontologies can be described with statements claiming to be true in that ontology. These statements are called axioms. A collection of axioms can form a hierarchy.

Let's consider following OWL2 example class written in RDX/XML syntax shown in figure. 2.3

Axioms from example OWL2 class from figure 2.3 can be seen in following figure 2.4 written in Manchester syntax (more in subsection 2.1).

It is possible to create a whole hierarchy tree structure from this class. Figure 2.5 shows that. Since class A is a subclass of B, C and D, it is possible to see it under all of those classes (please note, other classes may have their own axioms, that were not shown in this example).

get results can be different based on size of knowledge graph or complexity of ontology describing it.

```
<owl:Class rdf:about="#A">
    <rdfs:subClassOf>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="#C"/>
                <rdf:Description rdf:about="#D"/>
                <rdf:Description rdf:about="#B"/>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="#R"/>
                    <owl:someValuesFrom rdf:resource="#E"/>
                </owl:Restriction>
            </owl:intersectionOf>
        </owl:Class>
    </rdfs:subClassOf>
</owl:Class>
```

Figure 2.3: OWL2 example of class A

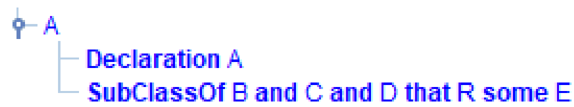

Figure 2.4: Axioms of example class A

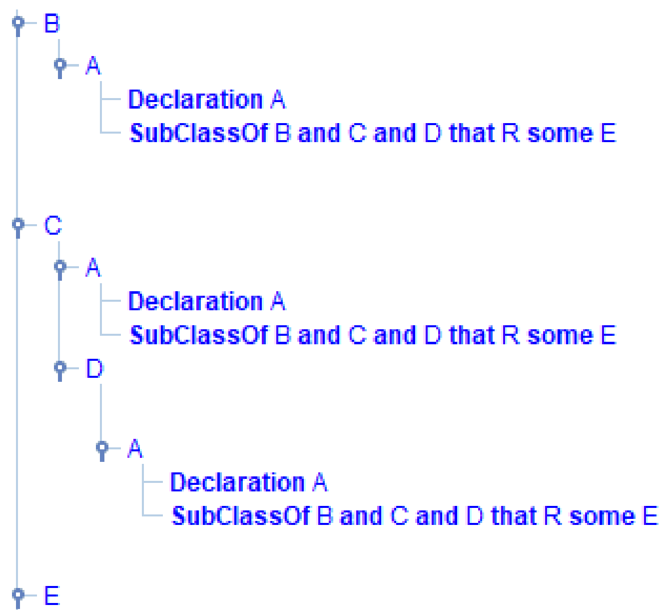

Figure 2.5: Axiom hierarchy tree example

Chapter 3

OWLDiff

OWLDiff[13] has been created as a tool for comparing differences between two ontologies and merging two ontologies into one new ontology. At the time of creating OWLDiff, there were no accessible tools for comparing two versions of the same ontology in human-friendly form. The aim of OWLDiff project was to develop a tool for OWL2 ontologies similar to diff for text files.[31] It was originally designed to compare two different versions of the same ontology. OWLDiff project has many modules, one of which is standalone application. This standalone application is independently executable and provides all features of OWLDiff to users.

Comparing and merging of two ontologies is quite difficult without any visual interpretation, even in readable syntaxes. Main reason is that knowledge inside ontology can be written differently but can have the same meaning. OWL family supports many syntaxes. In specifications or documentations, there is often used functional syntax. On the Internet, in compact human-friendly form, it is possible to find Manchester syntax. Since OWL2 is usually stored as XML or RDF syntax, OWLDiff focuses on mapping files with extension owx, owl or rdf (meaning, RDF/XML and OWL/XML syntax). However, it supports other file types as well, for example ttl (Turtle syntax).

OWLDiff takes two OWL2 files from the user. Firstly it takes the original ontology and after that the update ontology. Then it uses diff algorithms to check, if both ontologies are semantically equivalent. If not, the user is able to see axiom differences in form of axiom lists. The user can switch to different views, like hierarchy tree, which structures axioms into expandable lists by parent axioms. Each axiom is displayed in selected syntax. Default syntax is Manchester, because of its readability feature, however the user can switch to DL syntax.

For each axiom, there is also a note, describing its role in both ontologies. There are three roles distinguished:

- Common - Axiom is in both ontologies.
- Separate - Axiom has no connection to the other ontology.
- Inferred - Axiom is present in one ontology and not in the other, but can be derived. The axioms, which were used for deriving are described in the notmere.

- CEX - Axiom that were found different by CEX algorithm (more in section 3.1

By default, the result contains only the different axioms, however user can check an option for showing common axioms. Before merging, user can select axioms from original and update ontology, to delete or to add into merged ontology. For OWL2 files with EL profile, user can use CEX algorithm for finding differences, however this option is not default.

By default, the merged file is saved into the same location, as the original ontology. Therefore, the original is lost and replaced with the merged one. However, the user can specify location for the merged file, and keep the original unchanged.

3.1 Diff algorithms

Ontology comparison process is divided into two steps. In the first step, OWLDiff runs syntactic diff and entailment check. Second part is optional CEX algorithm and can be done only if both of the ontologies profile is EL. Algorithms are run with help of logic reasoning tool, which is a library that is able to get logic meaning from axioms inside ontologies.

Syntactic diff is very similar to diff between textual files. Basically, it checks whether axioms of ontology in OWL2 files are same or different (it also depends on syntax of those OWL2 files). The system creates two arrays: one with all axioms, that are in original but are not in update. And another array with all axioms, that are in update but are not in original. In standalone OWLDiff version, axioms from both arrays are marked with green color, which indicates differences.

Entailment check runs for every axiom in both arrays. It checks if axiom can be derived from the other ontology. If there are any axioms with this feature, it creates another array for each ontology. Then it places axioms into a respective array based on which ontology it may be redundant from. In standalone OWLDiff version, axioms in both arrays are marked with red color, which indicates redundant axioms. These axioms are recommended for deletion. Also, detail of how was the derived axiom derived is written below the all axioms.

CEX algorithm[14] only works with ontologies with profile EL. It finds out all axioms that are different, even when the difference cannot be found from syntactic diff and cannot be seen from class hierarchy.

3.2 Merge

Before merging, user can select axioms to keep from original and axioms to delete from update. After clicking the merge button, a popup screen with summary shows up. User can select file path for saving. By default, the merged file goes into the same location as the original ontology. Therefore, the original ontology is overwritten by default.

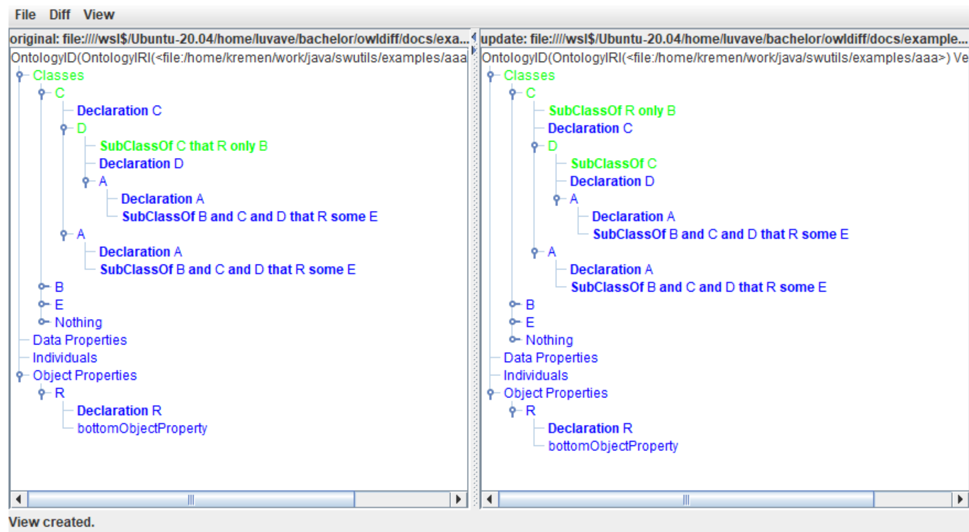

Figure 3.1: Ontology comparison result

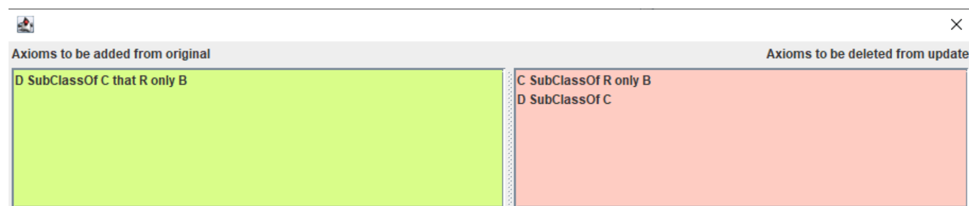

Figure 3.2: Selection of axioms before merging

Chapter 4
Analysis

In order to design the desired solution to accomplish the goal of this work, there needs to be specified requirements. OWLDiff standalone application will be the main source from which the requirements will be mostly derived. However, another good help for defining the complete solution will be research of already existing solutions and pointing out their good features and deficiencies.

4.1 Research of similar solutions

The aim of this section is to find out similar projects for finding differences between OWL ontologies. At this time, there are a lot more similar solutions than it was before, when OWLDiff was developed. However, most solutions are based only on some form of syntactic diff. Results may be different from OWLDiff only because of different approach to syntactic diff. Only small part of searched solutions have some kind of user interface and are available as a web solution.

Since the result of this work (section 1.1) will be REST API and web application with user interface for comparing OWL2 ontologies, the focus of the research was only on solutions implementing similar web application or REST API. Exception was made at the project Protege 4 (4.1.4), which does not have a web application, but it is more than 10 years actively developed, open-source and popular among ontology engineers. This research was done with aim to help understand user expectations and get ideas for features.

4.1.1 ecco

Very soon after release of OWLDiff, researches from Manchester university started developing a similar solution, based on OWL API[27] library. OWLDiff uses the same library for handling ontologies. Ecco[25] is able to find syntactic differences between axioms in ontologies and whether these differences have any logical effect. It finds and shows the source of each difference, which is used for categorization of the difference. Categories determine the strength of the axiom, so it can help the user improve its specificity. On-line website was not working at the time of research, however the user interface can be found inside the standalone application.

4.1.2 bubastis

Simple tool only able to check syntactic differences, using OWL API. Advantage of this project is on-line available GUI. Bubastis[15] is able to check whether class is modified, newly added or deleted from update ontology.

4.1.3 ELH-forgetting

Interesting tool for ontology logical comparison using forgetting method[40]. It has two functions, first is very similar to syntactic diff, however it also uses forgetting method, and it shows which axioms are explicit or implicit in the update ontology. Another function takes one ontology and tries to forget selected axioms, then it is possible to generate new ontology from remaining axioms a check its differences with original one.

4.1.4 Protege 4

Protege[30] is a complex tool for work with knowledge-based systems and ontologies. It has several functions for comparing ontologies. It shows different axioms in form of lists and each difference is categorized whether it was created, renamed, modified or deleted. Even when this tool does not have a web based version, it had to be mentioned, because it is one of the most popular ontology editors, and it is free and open-source.

4.1.5 Research summary

One of the goal of this work is to implement REST API. There were no project found that provides this feature. Only two projects have accessible web application. So only two projects provided a good source of inspiration. Also, none of these solutions have good way to prevent errors, user can upload anything and then gets unreadable error.

The following table shows pointed out positive feature, user interface design idea and overall user experience (UX) note that were taken as inspiration from each project. Mentioned are also things to avoid - deficiencies.

Name	Feature	Design idea	UX	Deficiencies
ecco	categorization of axioms, user can see if axiom is more specific in original or in update ontology	checkboxes for selection	explanations in tooltip at each axiom seems helpful	too much specialized terms
bubastis	shows if axiom is deleted modified or newly added	results are seperate in boxes	ability to choose diff options	confusing and repeated errors
ELH	unique forgetting method	left panel with tools	clear way how to upload ontologies	not great visualisation of axioms
Protege 4	complex tool that has ability to diff, merge and even show RDF graph	great way to show axiom hierarchy in form of tree	tool might not be easy to learn	does not have web application

Table 4.1: Research of similar solutions

What was learned from this research is that OWLDiff web application will be unique in its way to provide simple web accessible solution for users to compare ontologies. While creating requirements for OWLDiff web application, results from this research were taken into an account.

4.2 Functional requirements

Requirements that describe how the application behaves are called Functional requirements (FR). In this case, most of functional requirements are based on OWLDiff standalone module.

- FR1 - OWLDiff web application will serve the user interface, where users can directly upload ontologies into the system and get different axioms in form of tree created from axiom hierarchy.
- FR2 - User interface will distinguish common, separate and derived axioms using colors.
- FR3 - User interface will have option to show or hide explanations. These explanations will be shown at each axiom.
- FR4 - User interface will have option to use CEX algorithm. Affected axioms will be shown in specific color and have CEX description displayed.
- FR5 - User interface will have option to switch between syntaxes.

- FR6 - User interface will have option to show or hide common axioms.
- FR7 - User interface will make possible to select axioms to keep or select axioms to delete before merging.
- FR8 - OWLDiff web application will generate merged ontology from update ontology and selected axioms.
- FR9 - OWLDiff web application will allow downloading merged ontology in specified ontology file type and optionally changed filename.
- FR10 - OWLDiff web application will have available API. It will make possible for external systems to upload ontologies and get their different axioms or merge selected axioms from ontologies into the final ontology. REST API mediates all options that are possible in user interface.
- FR11 - Accepted file types of ontologies will be owl, obo, ttl, owx, omn and ofn. File types are based on supported OWL2 syntaxes (2.1).
- FR12 - Computing of differences between ontologies and merging of two ontologies will be done using OWLDiff.

4.3 Non-functional requirements

Requirements that describe how should application behave, so the desired usage is fulfilled are called Non-functional requirements (NFR). Most of these requirements are focused on quality and usability of the application.

- NFR1 - Application will be easily deployable on any server.
- NFR2 - All kinds of errors must be shown to the user, so the user knows what is happening.
- NFR3 - When an error occurs the user can clear the state and web application with REST API will continue to work.
- NFR4 - User interface will be able to display many differences from ontologies, there should be no limit for number of differences. API should limit the size of ontology based on hardware capacities of the device where is the API running.
- NFR5 - User must be informed if the size of the ontology is too big to handle by the server. Or if computational time will take too long.

Chapter 5
Analysis of used technologies

Selection of technologies has been affected by defined requirements. In some cases, the final choice has been made based on experience with a given technology. It is not the aim of this chapter to go deep into analysis of all possible option and finding the perfect candidate. Therefore, comparison between possible options is not presented in most cases, however always an explanation is provided.

Whole OWLDiff web application will be divided into two parts. First part realizes the API and will use OWLDiff for computing the comparison of ontologies. This part will be called the Backend (explanation and more detail for this division is inside chapter 6). Second part will be responsible for building user interface. Let's call that part the Frontend.

5.1 Backend - Java

Java[26] is the perfect **programming language** for creating a service running on nearly any device, because once it is compiled, it creates a package which can be deployed on any server supporting Java. Also, it is one of the most popular object-oriented language with huge community. Applications running on Java are highly scalable. The main reason why Java was chosen is that OWLDiff is all written in Java.

- **Reason:** OWLDiff written in Java

5.1.1 Spring

Java has a lot of backend frameworks, however each of them has differences and different uses. One of the biggest and most versatile backend **frameworks** is Spring.[34] It is very easy to build API endpoints, or services using simple annotations. It comes with built-in service SpringBoot, which helps to create standalone applications, that just run on its own.

- **Reason:** Popularity, easy to use for building REST API

5.2 Frontend - Javascript

For the frontend part, options are more limited. Since the aim of this project is to build a website, Javascript[11] comes to mind. It is a lightweight, cross-platform and interpreted **programming language**, therefore it is supported by all modern browsers. Javascript helps to create interactive websites. With it huge community support, there are avaliable multiple frameworks and libraries.

- **Reason:** Supported by all modern browsers

5.2.1 HTML + CSS

Plain Javascript does not make much sense, because it does not display anything by its own. Javascript only helps make website interactive. To build the actual structure of website and style each element, there is used a combination of HTML **programming language**[39] and CSS Cascading Style Sheets.

- **Reason:** Provide means to create a component structure for web application

5.2.2 Typescript

Typescript [20] is a **superset** of the Javascript. The main benefit of typescript is that it catches any type errors and syntax mistakes made by developer, before the website is generated and published. Another big feature of Typescript is simply adding types to your code, which helps make code more readable and easy to develop.

- **Reason:** Makes Javascript code easier to develop and scale

5.2.3 React + JSX

One of the most popular **frameworks** for creating user interfaces in Javascript is React.[19] React has been developed by Facebook with aim to create a simple component based framework. Using React, there is an option to use JSX, which is an extension for Javascript for writing HTML code inside javascript files.

- **Reason:** Easy to learn, huge community, great for dynamically changing data (in this case axioms).

5.2.4 NPM

Node package manager is the largest **software registry** for Javascript libraries, frameworks and tools. All libraries, frameworks or tools are called packages and developers can search for them on official NPM website[24]. All

```
export default function JSXExample(){

    const someVariable = "This is JSX Example!";

    return(
        <div>
            {someVariable}
        </div>
    )
}
```

Figure 5.1: Example of simple React component with JSX HTML rendering

packages are maintained by their authors directly from Github. Managing packages can easily be done with short commands directly from the command line of a computer. Reason why NPM is used is simple installation of React, Typescript and all libraries that might be used during development.

- **Reason:** Provider with the largest database of Javascript libraries, easy to use and ideal for React development.

5.2.5 Gatsby

There are few main options to prepare React based development environment.[7] For creating a single-page application with all its content generated with single Javascript file, there is create-react-app bundle. Another popular option is Next.js, this framework is for server-rendered applications with backend in Node.js[18], which is javascript based system. Since OWLDiff is written in Java, the Next.js framework is not ideal. Last option is Gatsby.js[7].

Gatsby **framework** is built on top of React with aim to develop static web pages. This means the website will be fast to load, and easily readable by any browser. Another big advantage of Gatsby is its better support for SEO[1] because unlike create-react-app it doesn't generate whole page in javascript file, but it creates whole HTML structure inside HTML files. Lastly, the main advantage of Gatsby is clean, readable and scalable code, thanks to its routing[2] solutions.

- **Reason:** Faster website loading, easier developing pages, supports SEO, great option for public web applications.

[1]Search Engine Optimization - several methods with only one goal, to show website on top of search engine results.

[2]Page routes are defined urls avaliable on the website and what components should be rendered when user visits those urls.

Detailed comparison of React environments is shown in the following table:

Difference	create-react-app	Next.js	Gatsby.js
website rendering	single-page application (everything dynamically rendered inside one element)	server-side rendering, requires Javascript as backend	multiple page application, statically prerendered
SEO	almost no SEO support	great SEO for dynamic content	great SEO for static, but also dynamic content
page routing	routing must be extra programmed	routes are generated	router are generated
simple config files	difficult to configure complex libraries	templates with config examples and simple configuration management	templates with config examples and simple configuration management

Table 5.1: Comparison of React environments

Considering the implemented web application should be possibly available to public, the SEO seems like a good added value. Also, generated routing and easy configuration makes it ideal candidate for more pleasant coding.

5.2.6 Material UI

One of the big advantages of React is to create reusable components. Not long after React was developed, the community started creating libraries and frameworks with a number of designed components for people to use. The biggest design framework for React is Material UI[33]. With its great documentation it is easy to use and with a growing community it is constantly evolving and improving. Some developers might argue that web applications built using Material UI look all the same, however the simplicity yet still wide choice of components make it a clear choice for this work. Material UI components can be styled using classic CSS or using their invented dynamically generated classes, which makes it easy for developers to style components directly inside Javascript code.

- **Reason:** Wide choice of components, nice looking out of the box, easily configurable, great documentation

Chapter 6
Solution design

This chapter describes architecture and key components of this work. From the goal of work, it is clear that the whole work should be divided into two parts. First part should be focused on implementing REST API service which allows comparing ontologies and second part should implement user interface that should mediate same functions like OWLDiff standalone.

However, both parts should be somehow communicate, because user interface needs REST API to display results for the user. We can call the REST API service ***backend***, because it will not be directly visible for users, and it will handle connection with OWLDiff.

The user interface will be mentioned as ***frontend*** because it will show visual elements for users and will allow users to interact with the backend.

6.1 Architecture introduction

When talking about how both parts should communicate, there should be first defined if frontend and backend will be inside a single application, or they should be separate. Backend was decided to be written in Java with Spring framework, due to OWLDiff being written in Java as well. In order to use all OWLDiff functions without no additional work, it is clear that backend should be another module inside OWLDiff project.

Question is if frontend should be inside that module as well? There are two major options:[1]

- Server-side rendering - backend creates pages and prepopulate them with data. When the user opens a specific URL, the browser fetches the whole page from the backend. There is smaller delay as the whole page is rendered with all the data. The backend doesn't need API as data for user interface are directly written into the user interface.
- Client-side rendering - pages are rendered inside users' browsers and data are fetched from backend afterwards. Improves user experience for dynamic web applications. Expects backend to have API.

Since one of the goals of this work is to make public API anyway, and it is expected for the result of comparison being re-rendered after user selects different options, clear selection is ***Client-side-rendering***.

6.2 Communication architecture

Now it is possible to have a clear view on communication between frontend and backend. Frontend creates pages that will be accessible for users from their browsers. Pages will communicate with created API - backend. Frontend and backend does not need to necessarily run on the same server. External services communicating with API can act as frontend itself and they will not run on the same server. Therefore communication architecture will be type of ***Client-server***[3].

Server will be backend running on any computer and frontend will be running on the same computer or any different computer, but still they must be able to communicate with each other. The frontend will send requests to the backend and get response for each of them. Backend will be waiting for requests and then respond to them. Requests and responses are usually done with HTTP protocol[16].

It is important to note in this section, that backend will communicate with OWLDiff directly thanks to backend being in the same project as OWLDiff. Backend will only extend existing OWLDiff with another module.

Considering frontend - client is running on Gatsby which provides React and backend - server is a Springboot application running Spring framework which will communicate with OWLDiff, we can have already certain idea how whole solution is separated. Figure 6.1 shows that.

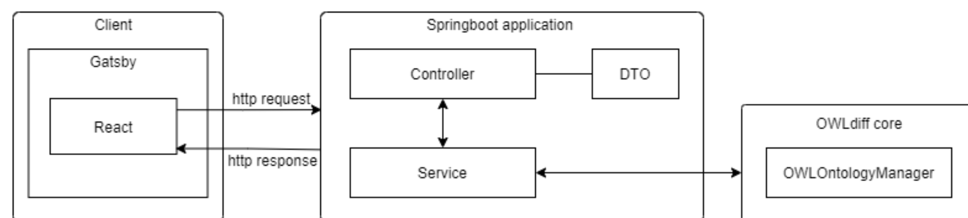

Figure 6.1: Solution design draft

6.3 Backend architecture

It is not the aim of this work to deeply compare all possible architectures, however in choosing the right candidate, one thing must be considered. It is a fact that backend application should consist of at least two parts, first part communicating with OWLDiff and parsing results and second part being API endpoints.

However, these parts are not entirely independent, and they communicate with each other directly. Therefore, it makes more sense to structure backend into layers. Another reason is the fact, that the most common architectural pattern for Springboot applications is ***Layered architecture***[32].

Layered architecture decompose components inside the application based on their role. It is not specified the number of layers or clear role of each layer.

What is important is that each layer has specific responsibility. Another important note is that layers should have defined order and flow of data should by default follow that order (there can be exceptions).

One of the key features is ***separation of concerns*** among components. Components inside a specific layer should implement only the logic assigned to that layer. This makes it easy to develop, maintain and test the whole application, thanks to well-defined component scope.

6.3.1 Layers

The top level layer should handle incoming requests, therefore it must implement API endpoints. These endpoints are functions that accept HTTP requests coming from outside of the application. Classes implementing these endpoints are called controllers. Controllers are usually inside a layer called ***Presentation layer***.

Presentation layer will communicate with another layer that handles main logic of backend. For example, when the user uploads ontologies via API endpoint inside presentation layer, that endpoint will call a function inside the second layer. The layer that handles main logic is called business layer, but in this case the ***Service layer***, because it will consist of classes called Services, that handles the main logic of backend.

Layered architecture can have more layers, like Persistence layer or Database layer, which both handles saving of data into some storage. OWLDiff web application does not have need for that. Instead, the Service layer directly calls OWLDiff functions and maps results using parser into specific objects that are returnable by endpoints.

6.3.2 Components

As mentioned, Presentation layer will consist of a controller with endpoints and Service layer will consist of a Service connecting OWLDiff and Presentation layer. However, there is a need for specific ***Data Transfer Objects*** (DTO), that does not belong to specific layer because it will be used by both Presentation and Service layer. These objects will be used to hold and transfer data in the form of parameters. Service layer will create those DTO objects from results from OWLDiff and pass them to Presentation layer, which will use them as return type of endpoints.

Another shared component will be ***SessionConfig***. It will be necessary to extend the default Session behavior. Once a comparison result from OWLDiff is done and returned to user, it must be used later when user changes some parameters and wants to get just a slightly different result (for example if user runs comparison and gets different axioms, but then he asks to get also common axioms, the same comparison result should be used). It is also possible that a user will run comparison using an external service, but then he will want to show the result inside the user interface. For that, it is needed to pull his session from that external service and give it to the user interface.

6.3.3 API architecture

API is an acronym for Application Programming Interface, meaning it is an intermediary between two different applications.[22] For example, when a user opens a web application like Facebook on his browser, the browser starts communicating with Facebook API. Usually, one application sends a request to the API and the API returns a response or the other way round. One of a reasons applications prefer using API instead direct communication is security. Application is never fully exposed to other software programs.

API endpoint is a place where the communication takes place. API can have multiple endpoints based on their functions and purposes. If part of communication is sending some resources, then the resources live inside endpoints.

There are few different API architectures. A specific API architecture was chosen based on following comparison from selected options[17]:

Difference	REST	JSON-RPC, XML-RPC	SOAP
approach	works with resources	works with actions (functions)	resources avaliable as functions
transfer protocol	HTTP	HTTP, WebSockets	HTTP, UDP and others
formatting type	JSON, XML plain text files and others	JSON for JSON-RPC, XML for XML-RPC	only XML
key strengths	flexible, scalable and cacheable	simple to implement	standardized rules and high security
weaknesses	not suitable for all environments]	small set of commands	more complex, worse performance

Table 6.1: Comparison of API architectures

The chosen API architecture was ***REST***, as it is sort of a middle class between all the options. It offers great flexibility, sufficient set of methods ,and it is simple to handle files over HTTP protocol. These features will be needed for handling ontologies.

Now it is possible to define endpoints which will be needed to implement to accomplish the goal of this work:

uploadAndCompareOntologies - This endpoint accepts two files, one file being the original ontology and another is the update ontology. Files can be any type that is supported by OWLDiff (owl, obo, ttl, owx, omn, ofn). Endpoint returns current session and comparison, meaning two arrays of different axioms. One describing what is different in original

ontology, another describing what is different in update ontology. Other optional parameters will be show or hide common axioms, usage of CEX algorithm, syntax of axioms and selection of displayed view (hierarchy tree or lists).

- Method - POST
- Request body form data
  - originalFile - original ontology file in binary
  - updateFile - update ontology file in binary
  - diffType - algorithm to use (syntactic, entailment or cex)
  - diffView - how the axioms should be shown and ordered (axiom list, classified view, frame view)
  - syntax - in what syntax should the axioms be written (manchester or DL)
  - generateExplanations - boolean, if set to true show explanations to each axiom
  - showCommon - boolean, if set to true it will also show common axioms, not only different
- Return codes and data
  - 200 - returns result of comparison, and session information
  - 400 - error from OWLDiff saying the computation went wrong with message explaining the error
  - 500 - any other internal server error

getComparison - Simple endpoint that expects uploadAndCompareOntologies to be run first. Accepts parameter with session identifier and returns comparison that was computed using uploadAndCompareOntologies endpoint. There is one main reason for this endpoint. If the user uses API to compute the differences without a user interface, but then he wants to view the differences in clear visualized form inside the user interface, the user interface should be able to connect to his session and use the computed comparison instead of computing it again. (for large ontologies computation can take a long time)

- Method - GET
- URL path parameter
  - id - id of session
- Return codes and data
  - 200 - returns result of comparison for specified seesion
  - 404 - error saying the session does not exist
  - 500 - any other internal server error

mergeOntologies - This endpoint expects uploadAndCompareOntologies endpoint to be run first as well, because it needs the comparison of

ontologies to make the merge. It accepts two arrays, the first being array of axioms that should be added from the original ontology and the second being array of axioms that should be removed from update ontology. Returns a merged ontology. Optional parameters are file type and file name of merged ontology.

- Method - POST
- Request body form data
  - id - id of session
  - fileName - optional specification of file name for merged ontology, default is file name of update ontology
  - format - optional file type format for merged ontology, default is file type of update ontology
  - add - array of axioms to add to merged ontology from original ontology
  - remove - array of axioms to delete from update ontology which is the base for merged ontology
- Return codes and data
  - 200 - returns file of merged ontology based on comparison from specific session
  - 400 - error from OWLDiff saying the computation went wrong with message explaining the error
  - 404 - error saying the session does not exist
  - 500 - any other internal server error

Detail of complete backend architecture is shown inside figure 6.2. The figure shows also specific API endpoints inside Presentation layer, main service with parser inside Service layer and specific DTO objects that will be used inside the backend. Pointed out are also SessionConfig and CORSConfiguration objects that defines how SpringBoot should behave.

6.4 Frontend design concepts

The advantage of Client-Server architecture is that backend and frontend does not need to run on the same machine. Or backend does not need frontend to run (meaning, it is possible to run API without the need to run user interface). However, if they run on the same machine, then frontend can act as an another layer of the Layered architecture - user interface layer.

Frontend implementation does not have a specific architecture by itself, because it is possible to see it as an another layer. But it is important to point out key design concepts[19] that should be followed:

- ***Stateless components***

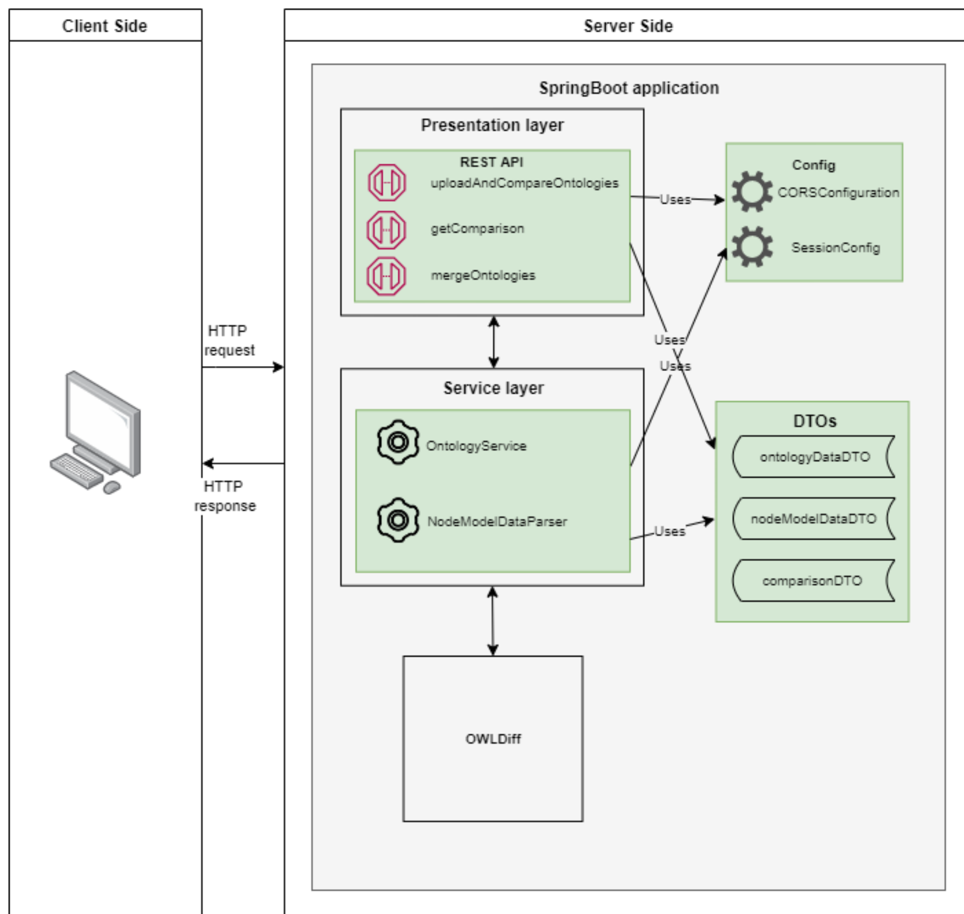

Figure 6.2: Backend architecture

- Description - React components should be stateless. Meaning, components have no lifecycle and don't save any state to the memory. Components should look like a basic Javascript function returning a React component.
- Reason - Components are more elegant, easier to understand and test. Stateful components will not be supported in the future.

- ***React hooks, conditional rendering***
  - Description - Used to dynamically re-render stateless components.
  - Reason - Complex components become easier to understand.
- ***Controlled components***
  - Description - Parent component can control child components and respond to their behavior.
  - Reason - Overwriting default behavior and handle callbacks from child components.
- ***Loose coupling***

  - Description - Parent component can control child components and respond to their behavior.
  - Reason - Overwriting default behavior and handle callbacks from child components.

- ***Compound components***
  - Description - Sets of components that work together to achieve a common goal, they belong to each other.
  - Reason - Components like menu or hierarchy tree of axioms should be assembled from smaller components. (For example, single menu items or single tree items)

Last but not least, Gatsby makes easier creating pages with their routes. To achieve that, there should be directory called "pages" and inside it there should be components, that are meant to present whole page.

6.5 Design

Design of the user interface for each page was based on results from research of similar solutions, section 4.1. The main goal was to make the user interface easy to understand and easy to use.

Important was that users will know where and how to upload ontologies and that differences will be displayed clear in the form of a tree formed from axiom hierarchy. OWLDiff supports different views, like just a list of axioms, so the tree must look understandable even when there will be no hierarchy of axioms and so it should look more like a list than a tree.

For creating design mocks, the Figma[5] software was used. It has a large community, and it was even possible to find an official design components library for Material UI.

Following figures 6.3, 6.4 and 6.5 shows the simple design that will be used as foundation for implementation. Please note that these are not finished looks and result of implementation may be different. Also, these screens does not show all the states of application that will be possible in completed implementation, details were not clear at the time of design. Designs should be tentative and result can be slightly different.

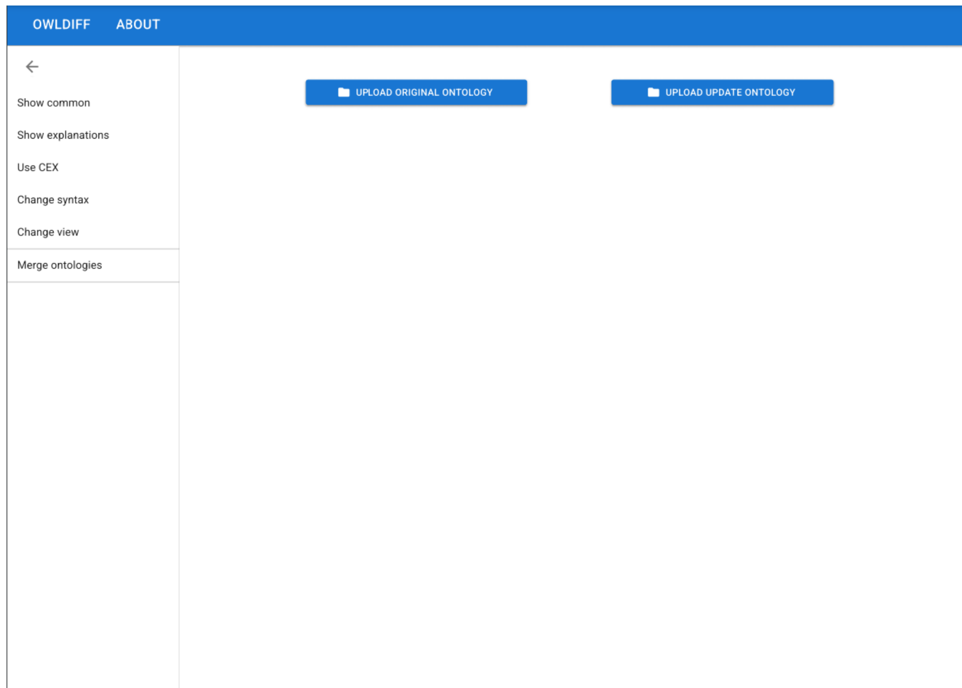

Figure 6.3: Design of main layout for user interface

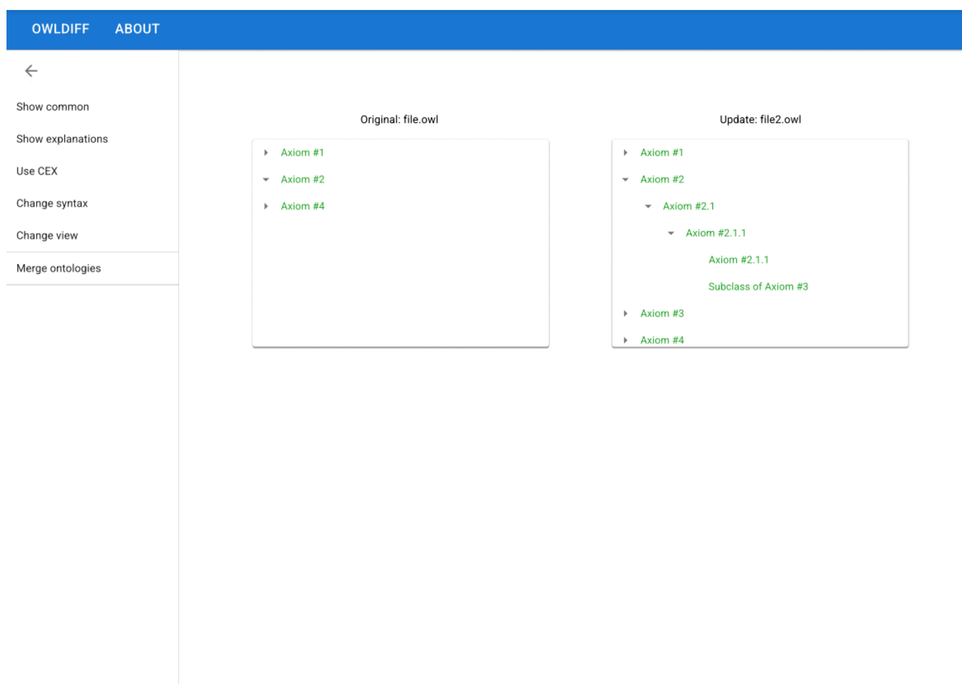

Figure 6.4: Design after comparison is done and displayed

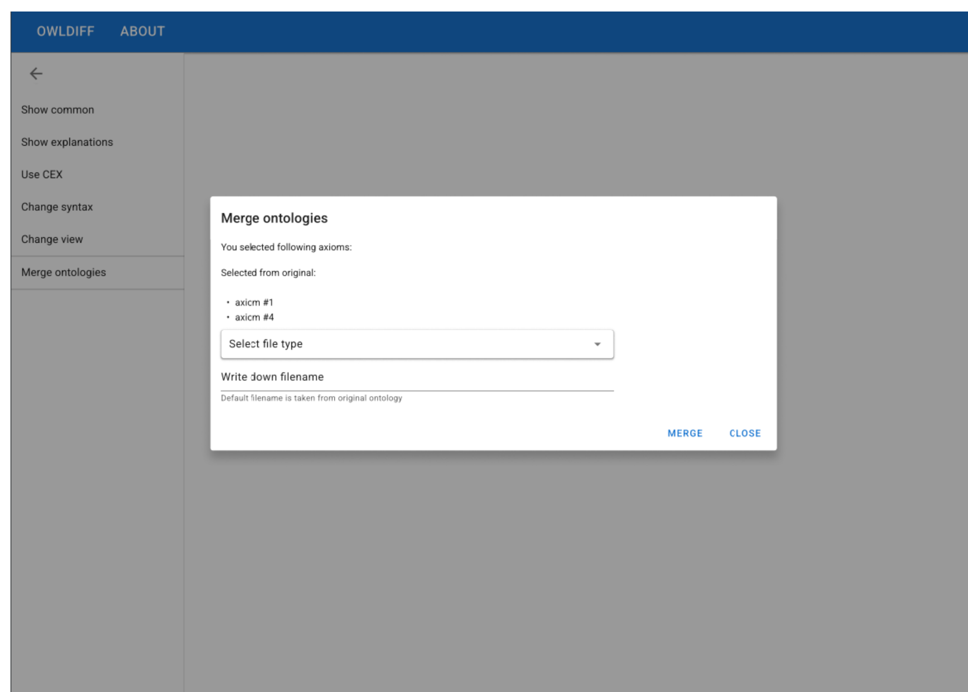

Figure 6.5: Design of merge window

Chapter 7
Implementation

This chapter goes through encountered problems with selected solutions and details of how the solution was implemented. Even though API and user interface are two different parts of the solution, the API inside the backend had to be implemented first because the user interface needs to communicate with the API using HTTP requests.

7.1 Integrating API into OWLDiff

OWLDiff project uses Apache Maven[21] as a build tool. Maven helps to manage dependent libraries and modules. API had to be integrated into OWLDiff project, which is structured into several modules:

- *owldiff-parent* - Holds information about the whole project like developers, organization, license and Maven repository.
- *owldiff-core* - The core of OWLDiff, it contains all the algorithms for comparing ontologies, functions for merging of two ontologies and helper functions for structuring the output of comparison.
- *owldiff-standalone* - Current runnable version of OWLDiff in form of a standalone application.
- *owldiff-cli* - Terminal based version which only supports syntactic diff between two ontologies with no extra options.
- *owldiff-protege-plugin* - Plugin version that was used for integration with Protege 4.

Another module has been added with given name *owldiff-api*. This is the module where the designed API is implemented. A generated diagram of modules structure with their dependencies can be seen on figure 7.1.

7.1.1 Using algorithms

OWLDiff core module has a communication interface, but it expects Java Swing to be setup for displaying output. Meaning, in order to display axioms,

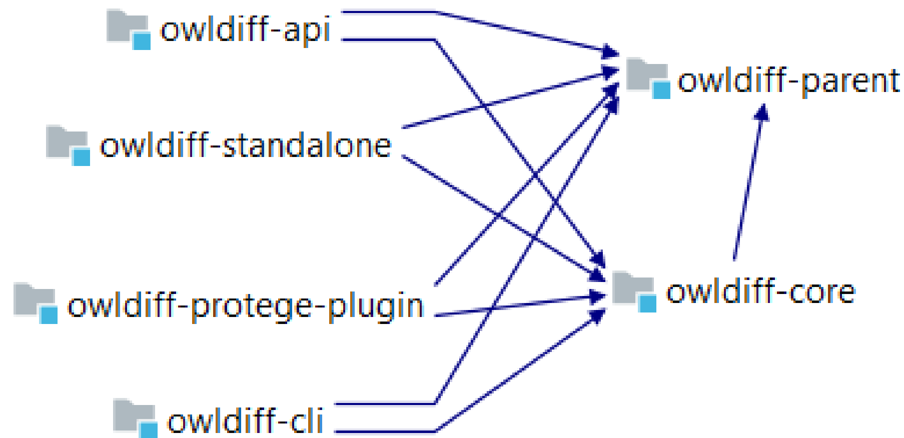

Figure 7.1: Generated diagram of modules with their dependencies to each other

comparison result and merge window the Java Swing layout needs to be used for user interface. However, since the aim of this work was to build a web application, which is not possible to make with Java Swing, it was necessary to access algorithms directly.

Each of the algorithms returns a list of axioms that are different in original ontology and a list of axioms that are different in update ontology. OWLDiff implements a tree model (called NodeModel) object that holds axioms, classes and properties (referred to as nodes) and structure them into tree from their hierarchy. This model was also used to structure lists of axioms from algorithms.

7.1.2 Axiom tree hierarchy as JSON

One of the biggest encountered problems in implementing the API was to figure out how to return the axiom hierarchy tree to users. Spring has a default mapper that can maps basic object into JSON[28] like String or Integers. JSON would be preferable output from API endpoints, however it was impossible to return tree node model that holds axiom in tree structure. It had to be mapped into JSON String with Jackson library. However, it was more complicated.

Reason, being the fact, that the tree node model was generated by OWLDiff, and it was designed to be generic and nested. The issue with nested objects is that the Jackson never knows when the nesting ends. If an object of a generic class has a reference to another object of the same generic class, implying that the another object has also reference to an object of the same generic class and so on, the Jackson gets into a loop when it tries to return this nested structure as a JSON String.

Luckily, the tree model from OWLDiff was designed by visitor design pattern. All it needed to be done was to implement a visitor that would visit

each of the nested objects and get data about the axiom it holds. This visitor would then take each node and write its name, type and description into the specified place. This place would be nodeModelDataDTO, which is already returnable by Jackson in form of a JSON String.

The implemented visitor and its behavior is shown on diagram 7.2.

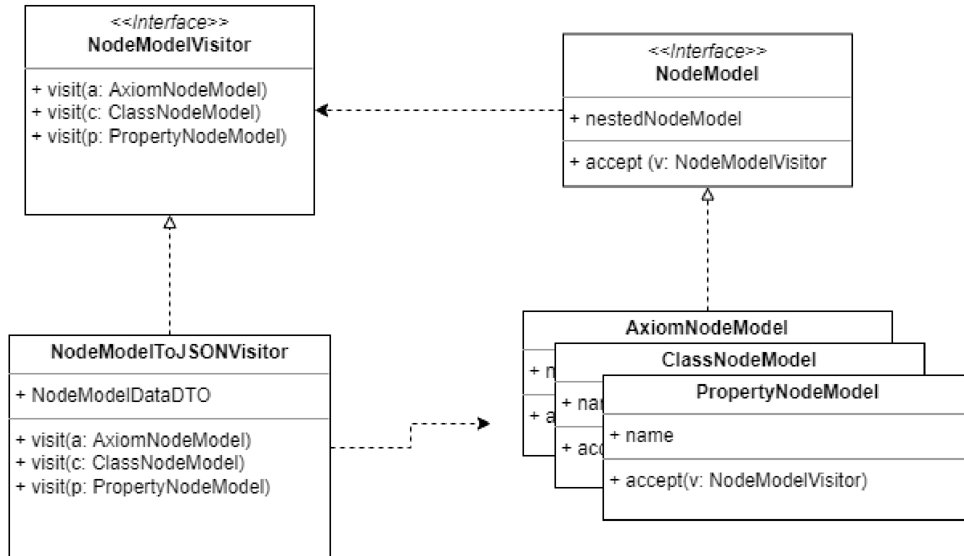

Figure 7.2: NodeModel visitor diagram

7.2 Implementing endpoints

The only issue that occurred while implementing endpoints for the API was the format of returning ontology from merge endpoint. It was clear from design to cover all formats that are accepted by OWLDiff to be accepted by API as well.

In uploadAndCompareOntologies endpoint, the ontology files for comparison are parsed to OWLOntology objects with OWLOntologyManager from OWLApi library. These OWLOntology objects are then saved into session and when the user decides to use merge endpoint for merging ontologies, he gets the modified update ontology.

User can select file type of this modified ontology, to do that the formatter had to be implemented. This formatter was highly inspired by similar formatter from ROBOT[10] tool. ROBOT uses OWLApi as well as OWLDiff does, so it was simple to create an enum which returns a specific format for each file type.

7.3 Tree model on user interface

While implementing the user interface, there was only one issue worth mentioning. It was rendering of tree model of axioms into the screen.

Material UI comes with a prebuilt component called TreeView (figure 7.4), which helps developers create nested expandable lists. An example of a basic expandable list can be seen on figure 7.3. Top level item can be expanded using the arrow on the left. Expanded items are shown under the top level item.

College

College
Declaration College
College SubClassOf Organization
College : Annotation label "school"^^http://www.w3.org/2001/XMLSchema#string

Figure 7.3: Basic expandable list, top level item can be expanded using arrow on the left.

If there are multiple expandable lists under another parent item, then they form a tree hierarchy.

Organization
Declaration Organization
Organization : Annotation label "organization"^^http://www.w3.org/2001/XMLSchema#string
College
Declaration College
College SubClassOf Organization
College : Annotation label "school"^^http://www.w3.org/2001/XMLSchema#string
Department
Institute
Program
ResearchGroup
University

Figure 7.4: TreeView component example

TreeView component was used to display axioms in form of a tree.

7.3.1 Encountered problem

It is important to note that at the time of implementation of this work, the TreeView component was still under development from Material UI. Even when the used TreeView component was stable, it lacked some configuration. The important one being some kind of pagination or lazy loading of not visible items.

To specify the problem, when a parent item in TreeView has hundreds if not thousands of child items, the expanding can take a while or even overload the browser. This is because all the child items are rendered after expansion. However, if there are thousands of child items, only a small part will be visible and to see the rest user must scroll down the page. Therefore, it is not needed for all child items to be rendered.

This issue was solved with library react-window, which allows React virtualization. Virtualization in React is still rather new concept that developers improve. What it means is that when the rendered page is too large for

browser to handle (and it is not necessary to render all of it) React renders only the visible part and the rest keeps ready for showing (figure 7.5). When user scrolls down the page it positions the hidden elements to the visible part of page using CSS and show them to the user.

This is useful for rendering super large tables or lists.

Figure 7.5: Visual explanation of how virtualization work (source [2])

React-window library was integrated into Material UI TreeView for all lists that have more than 100 child items. This was useful for large ontologies or when axioms were shown as plain lists (meaning the hierarchy is ignored and all axioms are in a single list).

7.4 Deployment

The final part of implementation was deployment on home server. It was necessary to properly test all the functions and verify that all functional and non-functional requirements has been fulfilled. Also, it made user testing much easier (more in chapter 8).

Hardware of the server is Raspberry Pi 4 model B with 1GB RAM. This will be important in testing chapter 8. Server has public domain address assigned to it, which makes it easy to access it from anywhere in the world.

API was created with SpringBoot in Java, therefore server needs Java to be installed. One of the biggest advantages of using SpringBoot is the fact, that it uses built-in Tomcat[29] configured to be running on specific port. Basically, SpringBoot applications can run on its own.

User interface are static HTML and CSS files which are presented to user on the specified URL address. They are served to users with Nginx[4]. Nginx

will help manage incoming HTTP requests from the clients[1]. When the client makes HTTP request to the domain address, Nginx will serve him the user interface files.

[1]Client can be a browser inside a computer, external service or basically any application from any computer around the world

Chapter 8
Testing

This chapter describes the approach that was used to test the API and user interface with results of each testing. Testing was divided into two main parts: performance testing and user testing.

Performance testing focuses on ability to handle all types and sizes of ontologies. Result should be deduction of limits.

User testing defines a testing scenario with specific test cases. These tests are then performed by volunteering ontology engineers. Purpose of this testing is defining future work and checking usability of the implemented solution.

8.1 Performance testing

Main goal of performance testing is to define limits for size of ontologies or number of differences. Since testing was done on two different machines (one being notebook with 8GM RAM, other being Raspberry Pi 4 server with 1GB RAM).

OWLDiff and API are written in Java, meaning most of the logic is handled by Java. Java heap memory lives in RAM, which means all the objects are temporarily stored in RAM. This is an issue for big ontologies, as comparison of ontologies creates an object for each axiom, class or property. Session holds most of the information which is partially stored on RAM as well. Even when sessions are cleared after configured timer, it is still possible to overload the server with multiple computations at the same time.

Part of this testing is also comparison between OWLDiff standalone application and implemented API + user interface.

For this purpose, there were compared following example ontologies from the Semantic Government Vocabulary[23] and OBO foundry[38] on syntactic diff:

1. OWL DL ontology with size of 1MB
2. OWL DL ontology with size of 10MB
3. OWL DL ontology with size of 100MB
4. OWL DL ontology with size of 500MB

5. OWL EL ontology with size of 10MB

8.1.1 RAM usage

Two different versions of example ontologies numbered 1.-4. with sizes 1MB, 10MB, 100MB and 500MB and also the 5. ontology type of EL with size of 10MB have been tested on implemented solution. Following table captures computational time in minutes (min) and seconds (s) of the initial syntactic diff. Basically the first computation that is done after user uploads original and update ontology.

Number	Size	Notebook	Raspberry Pi 4 server
1.	1MB	0.7s	4s
2.	10MB	20s	1min 12s
3.	100MB	2min 23s	17min 14s
4.	500MB	14min 47s	no result
5.	10MB	24s	2min 15s

Table 8.1: Performance testing results table

No result for number 4. on Raspberry Pi 4 server was caused because computation took too long for browser to handle. The response for the request didn't return under 30min so the browser timed out the request.

There is clear difference between both pieces of hardware. The main cause is probably RAM but is hard to define as most computation takes place inside OWLDiff and a logic reasoner. It is not the aim of this work to define computational hardware requirements of OWLDiff and logic reasoners. But it is still clear that implemented solution will more reliably run on stronger devices with high capacity of RAM memory.

Defining limits must be done specifically for each device that would run the API with user interface. Also considered might be actual preferences from future users. Based on feedback from supervisor, the limit for Raspberry Pi 4 server was set on 30MB per ontology. Reason being that computation higher than 5 minutes is too confusing for users, because it is more possible they might close the web application early with conclusion that it is simply not working.

8.1.2 OWLDiff standalone comparison

Same example ontologies 1.-5. have been tested on OWLDiff standalone application and computation time of initial syntactic diff has been compared with computation time on implemented API + user interface. The following table captures computational time in minutes (min) and seconds (s) of the initial syntactic diff on both applications.

Both applications were run on the same machine (the notebook) separately.

Number	Size	API + user interface	OWLDiff standalone
1.	1MB	0.7s	0.7s
2.	10MB	20s	19s
3.	100MB	2min 23s	2min 10s
4.	500MB	14min 47s	13min 30s
5.	10MB	24s	23s

Table 8.2: OWLDiff standalone performance comparison with created web application

Both applications use OWLDiff for computation logic, therefore it is understandable that computational times of initial syntactic diff are almost the same. OWLDiff standalone is a little faster mostly thanks to results being directly displayed in the app. In the API solution the results are first sent to user interface which uses React to render them in the browser.

However, this test proves that implemented solution behaves same as OWLDiff standalone version. The only difference is in the way of displaying results to users.

8.2 User testing

The implemented API and user interface web application should provide an accessible way for any user or service to compare two different versions of the same ontologies from anywhere in the world. However, it is expected that mostly ontology engineers might use the application.

Therefore, the testing scenario with test cases was prepared for people already having experience with OWL ontologies and then the testing was performed on volunteering ontology engineers. After the testing was done, the ontology engineers were given questions which should summarize their feedback for usability of each function.

8.2.1 Testing scenario

Testing scenario was designed with purpose to introduce main functionalities to users. Main functionalities are: comparison of two different versions of one ontology and merging of selected axioms of each version into one resulting ontology.

Testing scenario was divided into several steps - test cases. Each test case represent an action user must perform in order to continue to the next test case. It was not possible to split testing scenario into two separate test scenarios, because merging of ontologies is dependent that user already successfully performed the comparison.

Test cases were selected in the the following order:

Open the web application in any browser -> Upload original ontology -> Upload updated ontology -> Check that different axioms in form of a tree have rendered on the screen -> Expand axioms that are expandable, have nested child axioms -> Click show common button -> Check that tree now also shows common axiom and different axioms together -> Click merge button -> Select axioms to merge -> Create merged ontology

Detailed description of each step can be seen in the appendix table B.1. Results from a total of two recipients that attended this scenario testing are in the appendix table B.2.

All test cases were successfully passed, with exception to single test case for one recipient (Different axioms did not show for one ontology, only message "No axioms to display"). Main feedback from performing the test cases was that computing take a long time. It is possible to see from performance testing (section 8.1) that this was caused mainly because of weak hardware running the web application.

8.2.2 Asked questions

Recipients were asked 10 questions, trying to collect their overall feedback when using the application. Only closed questions were asked, because they can be easily analyzed and user don't have to type too much (which results in higher response chance to all questions).

Combination of single-word answers YES or NO and rating scale 1-10, when 1 being the worst experience and 10 the being the best were used. Each question included place to write down a note. Notes were highly useful when defining deficiencies of implemented solution and future work.

Detail of each questions with answers from recipients are in appendix table B.3. The main points based on answers are:

- Recipients had no problem uploading the ontologies.
- Explanation for colors of axioms should be more visible and clear as the different colors of axioms cause confusion.
- Recipients find showing common axioms useful.
- The default list view of axioms is only usable if the amount of expected differences between ontologies is very small.
- Web application and created API would be used for several scripts.
- The view showing axioms in form of tree based on which class they belong to is useful.
- Merging of ontologies were not clear and recipients were confused what will happen after merging.
- Recipients would use this web application in their everyday work, however they would like to improve displaying of different axioms.

- The user interface was easy to learn, but people not having experience with ontologies might find it confusing and would have no idea what is being done.

8.3 Test recap

Testing showed that even when implemented solution accomplishes the goal of this work it still needs some improvements to be used by wider public. User interface should be more clear and intuitive in a way that everyone can understand it, not only ontology engineers. And overall performance of the implemented API should be further tested and optimized.

Performance leaks might come from OWLDiff itself, however analysis of algorithms and usage inside OWLDiff was not part of this work. Deployment of the web application should be done on strong device, because comparison of small ontologies or ontologies with small amount of differences is not enough to be used in everyday work for ontology engineers.

Overall, the implemented solution met the expectations (functional and non-functional requirements), however performance issues for big ontologies were visible only after the solution was done and deployed.

Chapter 9
Conclusion

The main goal of this thesis was to design and implement a web application for comparison and merging of two different ontologies. Part of web application was the API, which allows external services to perform the comparison. The work followed up on the project OWLDiff, which is a tool, written in Java, that allows comparison and merging of two different ontologies. OWLDiff comes with a standalone application, which serve as the main basis for the design of the web application. Another inspiration came from research of similar solutions - available web applications allowing comparison of ontologies.

The designed API was written in Java with Spring framework, using the Layered architecture. User interface was a React web application using Material UI as a main source of components. API communicates directly with OWLDiff and uses OWLDiff for all computations. Together API and user interface forms a client-server architecture

Usability of implemented solution was tested with performance testing and user testing. Results from performance testing were not satisfactory as it came clear the computation takes a long time for big ontologies. Feedback from volunteering ontology engineers attending the user testing was helpful and proved that the application accomplishes its goal, however the user interface should be more clear.

9.1 Future work

Future work should include improvements on user interface, so it is more clear and viable for ontologies with big amount of ontologies. Also, explanations of each function should be added, different axioms should be on a same row for each ontology, so the difference is more clear and the colors of axioms should be better explained. Merging process should be more clear for users of what will exactly happen.

Also, performance needs optimization as well, part of it should be analysis of current OWLDiff implementation efficiency and definition of requirements for server where the application is deployed based on desired performance.

Bibliography

[1] Guillaume Breux. Client-side vs. server-side vs. pre-rendering for web apps, Sep 2018. https://www.toptal.com/front-end/client-side-vs-server-side-pre-rendering(Last visited on: 2022-05-04).

[2] Houssein Djirdeh and Jason Miller. Virtualize large lists with react-window. https://web.dev/virtualize-long-lists-react-window/(Last visited on: 2022-05-15).

[3] Inc. Encyclopædia Britannica. Client-server architecture. https://www.britannica.com/technology/client-server-architecture(Last visited on: 2022-05-04).

[4] Inc. F5 Networks. Advanced load balancer, web server & reverse proxy, Jan 2022. https://www.nginx.com/(Last visited on: 2022-05-02).

[5] Inc. Figma. The collaborative interface design tool. https://www.figma.com/(Last visited on: 2022-05-04).

[6] Ontotext Fundamentals. What is a knowledge graph?, Jul 2021. https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/(Last visited on: 2022-01-08).

[7] Inc. Gatsby. Fastest static-site generation web framework. https://www.gatsbyjs.com/(Last visited on: 2022-05-02).

[8] The Graphviz. Graphviz online. https://dreampuf.github.io/GraphvizOnline/(Last visited on: 2022-05-02).

[9] Jeff Heflin. OWL web ontology language use cases and requirements. W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-webont-req-20040210/.

[10] Rebecca C Jackson, James P Balhoff, Eric Douglass, Nomi L Harris, Christopher J Mungall, and James A Overton. Robot: a tool for automating ontology workflows. *BMC bioinformatics*, 20(1):1–10, 2019.

[11] JavaScript.com. Javascript.com. https://www.javascript.com/(Last visited on: 2022-05-02).

[12] Leif Harald Karlsen. Descriptive logic presentation. https://www.uio.no/studier/emner/\matnat/ifi/INF3170/h15/undervisningsmateriale/dl1.pdf(Last visited on: 2022-01-16).

[13] Kbss-Cvut. Kbss-cvut/owldiff: OWL comparison tool. https://github.com/kbss-cvut/owldiff(Last visited on: 2022-01-15).

[14] Boris Konev, Dirk Walther, and Frank Wolter. The logical difference problem for description logic terminologies. In *International Joint Conference on Automated Reasoning*, pages 259–274. Springer, 2008.

[15] James Malone. An ontology diff tool bubastis. https://www.ebi.ac.uk/efo/bubastis/(Last visited on: 2022-01-06).

[16] MDN. An overview of HTTP - MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview(Last visited on: 2022-05-04).

[17] mertech. Know your api protocols: SOAP vs. REST vs. JSON-RPC. https://www.mertech.com/blog/know-your-api-protocols(Last visited on: 2022-05-08).

[18] Inc. Meta Platforms. Create a new react app. https://reactjs.org/docs/create-a-new-react-app.html(Last visited on: 2022-01-15).

[19] Inc. Meta Platforms. React – a javascript library for building user interfaces. https://reactjs.org/(Last visited on: 2022-05-02).

[20] Microsoft. Javascript with syntax for types. https://www.typescriptlang.org/(Last visited on: 2022-05-02).

[21] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Apache Maven*. Alpha Press, 2010.

[22] MuleSoft. What is an API? (application programming interface). https://www.mulesoft.com/resources/api/what-is-an-api(Last visited on: 2022-05-08).

[23] OpenData MVCR. Semantic government vocabulary. https://opendata-mvcr.github.io/ssp/(Last visited on: 2022-05-15).

[24] Inc. npm. NPM. https://www.npmjs.com/(Last visited on: 2022-05-03).

[25] University of Manchester. Ontology diffing ecco, Jul 2021. http://owl.cs.manchester.ac.uk/research/diff/(Last visited on: 2022-01-06).

[26] Oracle. Oracle - java. https://www.java.com/en/(Last visited on: 2022-05-02).

[27] owlcs. Owl api. http://owlcs.github.io/owlapi/(Last visited on: 2022-05-02).

[28] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[29] Apache Tomcat Project. Apache tomcat®. https://tomcat.apache.org/(Last visited on: 2022-05-02).

[30] Protegeproject. Protegeproject/owl-diff-engine: Engine for calculating differences between two owl ontologies. https://github.com/protegeproject/owl-diff-engine(Last visited on: 2022-01-06).

[31] Jaromír Pufler. Owldiff documentation, 2008. https://kbss.felk.cvut.cz/tools/owldiff/(Last visited on: 2022-05-01).

[32] Mark Richards. Software architecture patterns. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html(Last visited on: 2022-05-04).

[33] Material UI SAS. The react component library you always wanted. https://mui.com/(Last visited on: 2022-05-03).

[34] Spring. Spring makes java simple. https://spring.io/(Last visited on: 2022-05-02).

[35] W3C. What is linked data? https://www.w3.org/standards/semanticweb/data(Last visited on: 2022-01-15).

[36] W3C. OWL 2 web ontology language document overview (second edition). Technical report, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

[37] W3Schools. XML/RDF. https://www.w3schools.com/xml/xml_rdf.asp(Last visited on: 2022-01-16).

[38] OBO Technical WG. The open biological and biomedical ontology (OBO) foundry.

[39] WHATWG. Html standard. https://spec.whatwg.org/(Last visited on: 2022-05-03).

[40] Yizheng. Gdhzlz/elh-forgetting: This tool could compute the logical diff between two large-scaled elh-ontology. https://github.com/gdhzLZ/ELH-forgetting(Last visited on: 2022-01-06).

Acronyms

API	Application Programming Interface
CEX	Centralized exchange
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
CTU	Czech Technical University
DL	Description logic
DTO	Data Transfer Object
FEE	Faculty of Electrical Engineering
FR	Functional Requirement
GB	Gigabyte
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSON	JavaScript Object Notation
JSX	JavaScript XML
KBSS	Knowledge Based Software Systems Group
NFR	Non-functional requirements
NPM	Node package manager
OWL	Web Ontology Language
RAM	Random Access Memory
RDF	Resource Description Framework
REST	Representational state transfer
RPC	Remote procedure call
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language

Appendix B

User testing

Following appendix contains imports from Excel file that was sent to volunteering testers. Answers are formatted so each question/test case has answers from all recipients.

Table B.1 describes test scenario with each step description. This scenario was sent to recipients to fill out. Results from scenario testing can be seen in table B.2.

After scenario testing was done, recipients were asked 10 closed questions. Description of each question with answer from all recipients can be seen in table B.3.

Test step	Test step description detail	Expected result	Status (PASS/FAIL)
1. open website	Open your chosen browser on following website: http://devdom.org/	loaded user interface	
2. select original	Click "UPLOAD ORIGINAL ONTOLOGY" button and find original ontology from test data folder, select it and click "OPEN".	button turned into text of original ontology filename	
3. select update	Click "UPLOAD UPDATED ONTOLOGY" button and find update ontology from test data folder, select it and click "OPEN".	button turned into text of update ontology filename	
4. ontology tree has rendered	Wait till loading finish and you see white boxes with text inside them, rendered under upload buttons.	white boxes with ontology data rendered, ontologyID should be on the top of them	
5. open axioms	Inside white boxes you should see axioms. Axioms that are expandable have arrow pointing down on the left. Click on that arrow, you should expand more axioms, if they have arrow as well, expand them as well. This will look like axiom tree.	there should be no performance drops when openning subtrees of axioms	
6. show common	Open tools with clicking on "SHOW TOOLS" on top left. Click "Show common" on top of the toolbar. Loading should show up.	left toolbar has opened and Show common has turned blue (selected)	
7. repeat steps 4 and 5	Repeat steps 4 and 5. Loading should again take similar time like you had to wait at step 4.	axiom tree should be bigger and contains black axioms	
8. select different View	Click "Change View" in the toolbar and select Classified frame view	confirmation dialog should pop up	
9. repeat step 4 and 5	Click "Continue" in the dialog and repeat steps 4 and 5	axioms in the tree should be more nested, and there should be different subtrees	
10. click merge ontologies	Click "Merge ontologies" in the toolbar	merge ontologies dialog should open	
11. select axioms	Select any axioms or click "Select all" in left or in the right part of dialog, then click "CONTINUE" button in bottom left	in the next screen you should see selected axioms	
12. create merge ontology	If you want you can type custom filename and select custom file type, click "MERGE" button	result merged ontology file should download	
Other testing feedback:			

Table B.1: Test scenario steps form ready to fill out

Test step number	Status from recipient #1	Status from recipient #2
1.	PASS	PASS
2.	PASS	PASS
3.	PASS	PASS
4.	PASS	PASS
5.	PASS	FAIL "No axioms to display" for one ontology
6.	PASS	PASS
7.	PASS	PASS
8.	PASS	PASS
9.	PASS	PASS but with major rendering issues
10.	PASS	PASS but with major rendering issues
11.	PASS	PASS
12.	PASS	PASS
Other testing feedback	Everything worked ok	Computation takes a while and would display color legend at all times. Merge dialog can end up as a very long scrollbar.

Table B.2: Results for all recipients from test scenario testing

Question	Answer from recipient #1	Answer from recipient #2
Did you know where and how to upload ontologies? (yes/no)	yes	yes
Did you understand the OWLDiff web application is trying to compare two ontologies? (yes/no)	yes	yes
On scale 1-10 (1 being the worst, 10 being the best), how much were the differences between ontologies clear in form of axiom list?	9	7 It took me some time, but after some initial confusion it becomes quite clear. The confusion is probably due to the colors explanation being initially hidden.
Did you find SHOW COMMON axioms useful? (yes/no)	yes	yes
On scale 1-10 (1 being the worst, 10 being the best), how would you rate List View visualization (the default one) usable for big ontologies?	1 Usable only for very small number of differences	8 It is just a guess, I do not compare ontologies on daily basis
On scale 1-10 (1 being the worst, 10 being the best), how much would you use OWLDiff web application for visual comparison of two ontologies?	7 I would like to use it in scripts to launch web browser from command-line and pre-upload files.	10
If you would use this project for comparing two ontologies. Would you use other views, or would the default List View be sufficient? (yes/no)	yes Any axioms that relate to a class are better to see together, so person see at once important context.	yes
On scale 1-10 (1 being the worst, 10 being the best), how much did you find the merging process clear?	6 Only after hitting merge button to see view what will be done made it clear.	3 I don't understand why the selector for original means "add" and for the update means "remove". I would expect both do the same.
Would you use this application as a tool in your everyday work? If yes, please describe how it would help you. (yes/no)	yes Yes but not sure, due to the issue in „Classified frame view“ where opened classes are not aligned in same row.	yes Probably mainly when there are changes in a domain ontology we use
Do you find user interface easy to learn and use?	no person that do not know what is CEX or explanations I feel like have no chance to understand well.	yes But there are major rendering and performance issues.

Table B.3: Results for all recipients from question answering

Appendix C

List of the attachments

C.1 Attached files

- `source-code.zip`
  – source code of implemented API and user interface
- `README.md`
  – file describing how to run the code locally or build the application

C.2 Github link

Implemented solution can be found on Github inside forked OWLDiff project:

`https://github.com/luvave/owldiff`