A Form Generator for Ontology-driven Data Acquisition

Masterarbeit

Dominik Sauerer

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik

Friedrich-Alexander
Universität
Erlangen-Nürnberg

A Form Generator for Ontology-driven Data Acquisition

Masterarbeit im Fach Informatik

vorgelegt von

Dominik Sauerer

geb. 17.11.2000 in Bamberg

angefertigt am

Lehrstuhl für Informatik 6 (Datenmanagement)
Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Betreuer: Prof. Dr. Richard Lenz
Caspar Felix Hanika, M.Sc.

Beginn der Arbeit: 01.06.2024
Abgabe der Arbeit: 02.12.2024

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Frensdorf, den 27.11.2024

Dominik Sauerer

Dominik Sauerer

Abstract

A Form Generator for Ontology-driven Data Acquisition

Ontologies are powerful tools for structuring and formalizing knowledge. They can be particularly useful in capturing research data, establishing terminologies, setting consistency rules, and linking facts in knowledge graphs. However, ontologies are often difficult to handle and data collection is cumbersome, which is why they remain a niche. Also, the further development of ontologies with new concepts poses a major challenge for domain experts without deep ontology knowledge.
This thesis investigates an approach to facilitate data collection for ontology-based information: ontology-based form generators that create data collection forms from ontologies.

In this thesis, a novel approach for such a form generator is developed, combining the key features of existing systems. Additionally, the prototype should enable guided ontology extension for domain experts without deep ontology knowledge through intelligent additional functions and simplifications.
In the first part of the thesis, the current state of research is examined and an overview of existing systems and approaches is provided. It is shown that existing form generators often only offer basic functions and do not consider the further development of ontologies. In the second part, the own approach is derived and described. The prototype is evaluated regarding its basic functions, and it is discussed how the additional functions can be meaningfully implemented and whether the prototype offers an added value compared to other systems.
It is shown that further development and extensions of ontologies in such a system is possible, and ontology-driven form generators generally represent an interesting approach, although certain limitations and challenges exist.

Zusammenfassung

A Form Generator for Ontology-driven Data Acquisition

Ontologien sind mächtige Werkzeuge zur Strukturierung und Formalisierung von Wissen. Besonders bei der Erfassung von Forschungsdaten können sie nützlich sein, um Terminologien zu etablieren, Konsistenzregeln aufzustellen und Fakten in Wissensgraphen zu verknüpfen. Allerdings sind Ontologien oft kompliziert in der Nutzung und die Datenerfassung ist umständlich, weshalb sie eine Nische bleiben. Zudem stellt die Weiterentwicklung von Ontologien mit neuen Konzepten eine große Herausforderung für Fachexperten ohne tiefgehende Ontologie-Kenntnisse dar.
Diese Arbeit untersucht einen Lösungsansatz zur erleichterten Datenerfassung für ontologiebasierte Informationen: Ontologie-basierte Formular-Generatoren, die Formulare zur Datenerfassung aus Ontologien erstellen.

In dieser Arbeit soll ein eigener Ansatz für einen solchen Formular-Generator entwickelt werden, der die wichtigsten Funktionen bestehender Systeme vereint. Zusätzlich soll der Prototyp auch eine geführte Ontologie-Erweiterung für Fachexperten ohne tiefes Ontologie-Wissen ermöglichen, durch intelligente Zusatzfunktionen und Vereinfachungen.
Im ersten Teil der Arbeit wird dafür der aktuelle Forschungsstand untersucht und ein Überblick über existierende Systeme und Ansätze gegeben. Es wird gezeigt, dass bestehende Systeme oft nur Basisfunktionen bieten und die Weiterentwicklung von Ontologien nicht berücksichtigen. Im zweiten Teil wird der eigene Ansatz hergeleitet und beschrieben. Der Prototyp wird hinsichtlich seiner Grundfunktionen evaluiert und es wird diskutiert, wie die zusätzlichen Funktionen sinnvoll implementiert werden können und ob der Prototyp einen Mehrwert gegenüber anderen Systemen bietet.
Es zeigt sich, dass eine Weiterentwicklung von Ontologien in einem solchen System möglich ist, und Formulargeneratoren generell einen interessanten Ansatz darstellen, jedoch auch gewisse Einschränkungen und Herausforderungen bestehen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Research data, which are produced in various organizations, are generally not comparable because they are created in different conceptualizations: For example, different terminologies and definitions are used for the same data, which makes an automatic matching very hard, and human support is generally needed. This issue could be mitigated if, already at the acquisition of the research data, fitting domain ontologies are used to structure them. In many scientific disciplines there already is a huge amount of domain ontologies freely available for various topics. These ontologies allow the structuring of data and the classification of facts in standardized terminologies. Data acquired in this manner gains semantic interpretability to a certain degree, and can then be compared to other data, which was also acquired on the same semantic basis.
But although ontologies seem like a reasonable approach, the usage of them is in general rarely seen in practice, aside for a few disciplines especially in medical areas. A reason for this might be that ontologies are generally considered a niche topic, where you first have to understand many theoretical concepts and technologies of creating and populating ontologies, which deters many people.
This problem is further exacerbated by the fact that for specific tasks a general domain ontology must be refined to an application ontology with more specific concepts, which is a task that is even more difficult than just adding data.

A possible solution for this problem, populating ontologies with concrete data without having that much knowledge about the ontologies themselves, is to use fillable forms, derived by the ontologies, to gain research data fitting in the respective semantic bases. By this a knowledge base of the ontology could be built up also without ontology knowledge because only prepared forms have to be filled.

In a further thought, not only populating an ontology with new data via forms may be possible, but also the aforementioned extension of the ontology itself, for instance, if new fields in the given forms are added, and thus also new concepts in the ontology itself.

1.2 Goal and Research Questions

This thesis aims to investigate which approaches to an ontology-driven data acquisition based on forms are already presented in literature, and which requirements these data acquisition systems must fulfill. This will be achieved by compiling a survey of form-based ontology-driven data acquisition tools. The survey will present an overview of relevant literature in this topic, as well as examine the proposed systems in detail to thoroughly understand, how these systems work, what the forms actually do, and what additional features these systems implement. For these additional features the focus in this survey is set on usability, especially for non-ontology-experts, and the possibility to not only populate an ontology, but also develop the ontology and therefore the semantic basis itself, as already mentioned earlier.

As a second step an own form generator prototype will be implemented. The survey results will be used here to provide a foundation and then further develop the concepts especially regarding the already mentioned topic of ontology extension. The goal of this implementation is to get a prototype including the main features discussed in various papers in a modern technology stack, to clarify the difficulties and challenges in implementing such a system, and to test features which were not implemented in other papers yet. This will allow for a discussion, how an ontology extension, done by domain experts without ontology knowledge, can be done using a form generator, and where challenges and problems regarding this goal occur.

In short, these goals can be formulated as the following research questions, which will be addressed in this thesis:

1. Which ontology-driven data acquisition form generators already exist in research projects?
    a) What objectives do these systems pursue?
    b) How do they influence the knowledge bases of the given ontologies?
    c) How do these systems actually utilize the ontology?
    d) What additional features, especially regarding usability and ontology evolution, do these tools implement?
2. How can an own ontology-driven data acquisition form generator be implemented?

   a) How should such a tool be designed if basic software engineering rules should be obeyed and if this tool should be usable also by non-ontology-experts?
   b) Which technologies should be used if web forms are the preferred target?
3. How can additional features regarding ontology extension, especially ones not implemented in previous projects, be added to this tool?
   a) Which features make sense for such a generator tool in contrast to a standard ontology editor?
   b) Can these features be implemented, so that they can be sensibly used by non-ontology-experts?

1.3 Structure of this Thesis

This thesis is structured to address all research questions successively. First, an introduction to the whole topic of ontologies and especially OWL and ontology engineering is provided in chapter 2 to establish a common understanding for the following chapters. Next, the mentioned survey to answer research question 1 is presented in chapter 3. In this chapter the methodology of the survey as well as the results are presented. Chapter 4 then discusses the basic requirements and the own approach of the implemented prototype, including general software engineering patterns used in this project, for research question 2, while chapter 5 then covers the design and implementation, with a special focus regarding the ontology extension features for research question 3. Following that, in chapter 6 the prototype is evaluated against the requirements and the additional features for questions 2 and 3. It is particularly discussed here, if the novel features could be implemented in a way, so that they can be used by non-ontology-experts, and if this prototype provides an added value in comparison to the other tools. Finally, in chapter 7 a conclusion is drawn, and an outlook on future work is given.

1.4 Terminology

Before starting with the main content, some basic terminology is explained, which is used in the next chapters.
The topic is about already mentioned “ontology-driven form-based data acquisition systems”. This term refers to tools, that:

- Acquire data for ontology knowledge bases as primary task,

- Are driven by an ontology, which means that an ontology is the basis for this tool, and the data is structured in the way the ontology is structured, and
- Produce or generate forms for the data acquisition.

Because there does not exist any widely accepted term for this kind of tools, and the former used one is a rather long term, the simpler term “ontology-driven form generators” is used in this thesis from now on.
The tool developed in this thesis is referred to as “prototype” because the tool should not be a finished product but a first, preliminary version of an own ontology-driven form generator intended to test the general concept and additional features.

Other important terms, especially about ontologies and OWL/RDF in general, are explained in the next chapter.

Chapter 2

Basics of Ontologies

In this chapter, fundamental concepts regarding ontologies are explained, as they are essential for understanding the subsequent chapters.

2.1 Definition

Defining the term "Ontology" precisely is challenging due to its usage across various disciplines [Hep08, p. 4]. Guarino notably identified two primary interpretations of the term [Gua98, p. 4]: Firstly, there is a philosophical interpretation, which defines an ontology as a system of categories for some part of our world. Secondly, within the realm of Artificial Intelligence (AI), ontologies are viewed as vocabularies that describe reality, incorporating assumptions about the semantics of this reality [Gua98, p. 4]. This AI-centric definition aligns closely with the philosophical one but further considers an ontology as an engineering artifact designed for specific use cases [Hep08, p. 5] [DM08, p. 29].
One of the earliest and most important precise AI definitions is provided by Thomas R. Gruber: He describes an ontology as an "explicit specification of a conceptualization", where a "conceptualization" is the knowledge you want to describe within a domain, including objects, entities and their relationships [Gru93]. So in the context of ontologies, the goal is to create formal specifications using specially designed ontology languages for a particular knowledge domain, where rules, objects, and other elements are encoded. An ontology typically does not include the actual knowledge of the domain itself but rather establishes the rules and terms used to represent that knowledge [Hep08, p. 6]. For instance, an ontology of the animal world would therefore include the taxonomy and the structure, but not the animals itself. This would then be called a Knowledge Base. In literature these both terms are often used interchangeably, also because in practice when ontologies are exchanged via files, these files often also include the knowledge bases. But in this thesis the term "ontology" is used for the formal description of the domain knowledge, and "knowledge base" for

the actual knowledge. In this thesis, the term knowledge base also always refers to knowledge derived from an ontology.

There are several motivations for utilizing ontologies: The primary objective is to achieve interoperability between different representations of reality [Hep08, p. 4], which is particularly relevant across various scientific disciplines. With a formal description of the domain knowledge everyone can agree to a unified vocabulary and everyone has a clear understanding of each term.
There are various other applications of ontologies described by Lee Gruninger [GL02]: In general, ontologies helps to organize and structure knowledge. Beyond enhancing human communication, ontologies also improve interaction with computational systems, as the ontology definitions are formal languages, which can easily processed by a computer. By this, computers can also infer new knowledge from a given ontology: For instance, if an ontology specifies that a human is a type of ape and apes are animals, a reasoning system can deduce that humans are animals. This powerful feature of ontologies has led to extensive research on the use of logic (e.g. description logics) in this field [BHS04], although logical rules are not considered an essential feature for every ontology [Hep08, p. 15].

As Hepp notes, the expressiveness of an ontology can vary significantly, ranging from a simple vocabulary with a few unconnected terms and definitions to a complex higher-order logical system with numerous axioms for each term [Hep08, pp. 8 sq.]. In addition to this classification based on expressiveness, ontologies can also be categorized by their level of specificity for a given use case. According to Guarino, there are three types of ontologies [Gua98, pp. 9 sq.]:

- Top-Level Ontologies: Broad ontologies with very general concepts among multiple domains
- Domain Ontologies: More specific vocabulary and rules, but still generic enough for a broad field
- Application Ontologies: Highly specific ontologies for some particular task

To illustrate this classification, consider the following examples: A top-level ontology could be the General Formal Ontology (GFO) [1], which describes very general concepts of objects, processes, categories and sets without assigning them to specific knowledge domains. Instead, these terms can be applied in various fields, including medicine, physics, computer science and philosophy.
More concrete is then a domain ontology like the Disease Ontology [2], which provides

[1]See: `https://onto-med.github.io/GFO/release/latest/index-en.html` (Accessed: 11/10/2024)

[2]See: `https://disease-ontology.org` (Accessed: 11/10/2024)

a structured representation of human diseases with specific information and relationships to other classification schemes. This ontology is used in one domain only, the medical domain.
An application ontology might be derived from the GFO and be designed for a specific hospital, detailing the knowledge of that hospital, including the specific diseases treated, treatments used, and doctors employed.

The existence of these different types of ontologies is particularly beneficial in the context of reusability. In the Semantic Web field, the “heavy re-use of ontologies was part of the original conception”[SHH23, p. 1]. General top-level and domain ontologies can be highly useful for various applications, as they can be reused across different contexts. However, in practice, ontologies are often defined too specific for a particular application, lacking the granularity needed for broader reuse [SHH23].

The Semantic Web is also in general one of most significant application areas for ontologies. It is the idea to extend the basic World Wide Web so that both humans and computers can automatically search the Web based on the semantic meaning of Web pages [Hit+08, p. 11]. These concepts were already discussed in the early 2000s by Tim Berners-Lee, the founder of the WWW, among others [Hit+08, p. 13], leading to the development of various technologies by the World Wide Web Consortium (W3C) that are still used in ontology research today. The most critical technologies for this project include:

- The Resource Description Framework (RDF): A framework for modeling information about resources and their relationships as knowledge graphs [Sch+14].
- The RDF Schema (RDFS): A simple ontology language based on RDF to define class structures and properties.
- The Web Ontology Language (OWL): A more powerful ontology language, which adds more concepts to RDFS.
- The declarative query language SPARQL: Used for querying RDF graphs [Sch+14], allowing for the retrieval of specific information from ontology-based knowledge bases.

RDF is a standard in the Semantic Web domain and serves as a crucial foundation for various ontology languages. In essence, RDF describes information using a format called “triples”, each consisting of a subject, a predicate, and an object, and thus describes a relationship between two entities, the subject and the object [Sch+14]. All three are resources, which can be concrete objects in the world, such as things, animals, humans or actions, although the object can also be a literal, such as a string or a number.
An important aspect of RDF is reusability and unambiguity. Every resource has

a unique IRI, so that even two entities with the same name can be distinguished. The scope of these IRIs is global, so that every resource can be identified worldwide [Sch+14], and also in different knowledge bases. A knowledge base constructed from RDF triples can be referred to as a Linked Data graph, where resources serve as nodes and predicates as edges, forming a graph structure where every node can be connected to any other node via predicates [Sch+14].

A simple example of an RDF triple is:

```
    <http://example.org/JohnDoe> <http://example.org/eats> <http
        ://example.org/Banana>
```

This triple represents the knowledge that John Doe (subject) eats (predicate) a banana (object), with each resource having a globally unique IRI.

2.2 Ontology Languages

The description of an ontology as an engineering artifact and its specific concepts is highly dependent on the formal ontology language used. Various languages exist, with the most important ones, which are also interconnected by the used technologies, being RDFS and OWL.

2.2.1 RDF Schema (RDFS)

RDFS is a very basic ontology language, directly based on RDF. It is referred to as a “Semantic Extension” to RDF [HPS14], adding pre-defined entities to RDF as vocabulary and giving them also semantics. By this, certain triples are “forbidden”, as they do not make sense in the context of RDFS, although RDFS does not provide a real foundation for logical reasoning [Cam24].
In essence, RDFS organizes all entities of a knowledge base into the primary types of classes and properties [BG14]. Specific entities of a class are then called instances, and are connected to the classes via the RDF predicate “rdf:type”. With this connection, both the model and the concrete knowledge can both be represented with RDF triples. This is analogous to data modeling in databases, where classes correspond to tables and instances to concrete rows. RDFS also introduces a class hierarchy system, allowing classes to be defined as subclasses of other classes, thereby inheriting properties from their superclasses [BG14].

The properties connect instances to each other. For that, properties can be designed with so-called domains and ranges. These are then classes within the ontology where the property can be applied. For example, a property “eats” can be defined such that only humans can eat food by setting the domain to “Human” and the range to “Food”. There also exists a hierarchy system for properties [BG14].

With these fundamental concepts, an ontology and a knowledge base can already be constructed: Triples describe the classes and properties, providing the formal structure of the ontology, while the concrete data triples adhere to the rules defined by the properties and are organized into the classes. However, as previously mentioned, RDFS does not provide a robust basis for logical reasoning and has a very limited rule system. For more complex ontologies, OWL is therefore used, which will be introduced in the next section.

When RDFS is applied to the simple banana example, it could be represented as follows:

```
<http://example.org/JohnDoe> <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#type> <http://example.org/Human>
<http://example.org/Human> <http://www.w3.org/2000/01/rdf-schema#
    subClassOf> <http://example.org/Animal>
<http://example.org/Banana> <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#type> <http://example.org/Food>
<http://example.org/eats> <http://www.w3.org/2000/01/rdf-schema#
    domain> <http://example.org/Human>
<http://example.org/eats> <http://www.w3.org/2000/01/rdf-schema#
    range> <http://example.org/Food>
<http://example.org/JohnDoe> <http://example.org/eats> <http://
    example.org/Banana>
```

In this example, the concrete information is still present in the last triple, but an ontology structure is also established with the first five triples. The first triple defines that John Doe is an instance of the Human class, and the second triple indicates that humans are animals, and so a subclass of the Animal class. The third triple defines that banana is an instance of the Food class. The fourth and fifth triples establish rules for the “eats” property: the subject must be a human, and the object must be food.

2.2.2 Web Ontology Language (OWL)

The Web Ontology Language (OWL) has been an ontology modeling standard recommended by the W3C since 2004 [Hit+08, p. 111]. In 2011 and 2012, OWL 2 was developed to enhance the original OWL specification, particularly in terms of syntactic sugar, expressiveness, and datatypes [GW12]. Consequently, this project utilizes OWL 2.
OWL builds up on RDF and RDFS, incorporating many RDF and RDFS terms while

introducing additional ones [Hit+08, p. 112]. The fundamental elements of OWL are akin to those in RDFS [Hit+12]:

- Individuals: Concrete objects in the world, such as the person Mary W., analogous to instances in RDFS.
- Classes: Categories, to which individuals belong, e.g. the class Person
- Properties: Relationships between entities, such as a “married” relationship between two persons.

An important difference from RDFS is that a differentiation is made between various kinds of properties: Object properties relate two individuals to each other, while datatype properties assign simple data types, such as strings or numbers, to individuals [Hit+12]. Aside from these two main types of properties there are also annotation properties. These can be associated with every entity in OWL, classes, properties or individuals, and can be designed either as object or datatype properties. Although annotation properties do not contribute to the logical structure of the knowledge base, they provide meta-information about entities, such as alternative names or descriptions [Hit+12].
The class system of RDFS is directly used in OWL, with the “rdfs:subClassOf” property. However, OWL imposes a stricter hierarchy by defining a general superclass named “owl:Thing”, to which every other class must be a subclass [Hit+08, p. 118]. This strict class hierarchy is comparable to the definitions of some object-oriented programming languages, especially Java, where every entity is a class bound to a hierarchy, with the most abstract class being “Object”.

One fundamental feature of OWL is its ability to utilize logical reasoning to infer new statements from an ontology or to verify the logical consistency of the ontology itself. These logical statements are derived from two primary sources within an OWL ontology. Firstly, many statements are implicitly generated by the structure of the classes and properties and their inherent characteristics. Solely by the subclass hierarchy logical implications can be drawn, for instance if one individual is transitively also belonging to another class. Here also RDFS elements are again used, such as domains and ranges, where it is verified, if all data triple adhere to these rules [Bec+04].
In addition to these implicit and straightforward restrictions, OWL supports more complex constraints known as “Class Expressions”, which can be linked with existing classes and properties through “Axioms” to establish logical rules for the ontology [MPSP12]. For simplicity, these terms will be referred to as “Restrictions” in this thesis. These restrictions are applied to class-property combinations, enabling the creation of rules that must be followed when using certain properties with individuals of specific classes [Bec+04]. OWL includes value and cardinality restrictions, where the former specifies whether individuals of a specific class must or must not be

connected to certain property values, and the latter imposes constraints on the minimum and/or maximum number of property connections a class must have. These restrictions are integrated into the ontology by treating each restriction as an anonymous class. Classes that must adhere to a restriction are defined as subclasses to this anonymous class [Sch05].

Due to the possibility that two concepts requiring a property connection may reside in different ontologies, OWL provides an import system. This system allows any ontology to import other OWL ontologies by specifying their URIs, thereby enabling the reuse of classes, properties, and individuals from these external sources. Tools such as the ontology editor Protégé automatically resolve these imports to achieve a complete closure of all necessary concepts.

A significant drawback of OWL is that it is undecidable in its full specification [Hit+08, p. 113]. That means that not every logical query about a complete OWL ontology or knowledge base can be answered, and some queries may be answerable, but could take an impractically long time in the worst-case scenario. To address this issue, the W3C has defined sub-languages of OWL, each containing only a subset of the full OWL specification. These sub-languages are less expressive but offer better computational performance. Limitations in these sub-languages may pertain to how classes can be defined or constructed, or which restrictions can be formulated [Hit+08, p. 140].

Among these sub-languages, OWL Lite is a strongly restricted version that is not widely used [Hit+08, p. 140]. The most commonly used sub-language is OWL DL, which includes multiple restrictions to ensure that its semantics can be described using Description Logics [Hit+12]. The most crucial restriction in OWL DL is the strict separation between classes and individuals [Hit+08, p. 139]. This means that an individual cannot simultaneously be a class in another context, a concept that is permissible in the full OWL specification. The big advantage of OWL DL is that it is fully decidable and enjoys substantial support from various software tools [Hit+08, p. 113].

2.3 Ontology Tools

There are many tools available to work with ontologies and knowledge bases. The following categories of tools are particularly crucial for developing and using ontologies.

File Formats Various file formats and syntaxes exist for storing and exchanging ontologies. While all the presented formats can preserve the same RDFS and OWL data, they differ in their intended purposes, particularly concerning their target audience: humans or machines. For OWL and RDFS ontologies, the most important formats are [Hit+12]:

- RDF/XML and OWL/XML: These are the standard formats for RDF and RDFS ontologies, where the triples are stored in an XML file. Both can be used for OWL: In RDF/XML a special mapping for OWL concepts is needed, while in OWL/XML these are natively supported.
- Functional Style Syntax: A syntax for specifications and for working with APIs.
- Turtle: A more compact and readable format [Bec+14].
- Manchester Syntax: Another human-readable syntax.

Editors Ontology editors are tools designed to create and modify ontologies. They can read ontology/knowledge base files in formats such as XML, Turtle, or others, allowing users to edit the constraints and contents of the ontology. Protégé is one of the most well-known ontology editors, but other editors, such as webOnto and ONTOLIS, are also available[3].

Reasoners Reasoners are essential tools that serve two primary functions. First, they can infer indirect knowledge that is not directly stated in an ontology, such as indirect class memberships. To achieve this, reasoners construct internal data structures [Gli+14]. Without a reasoner, one could only utilize the direct class and instance declarations specified in an OWL file.
Second, reasoners can be used to verify the consistency of an ontology from a logical point of view, ensuring there are no contradictions. The construction of data structures by the reasoner can only occur if the ontology is logically consistent. If inconsistencies are present, the reasoner can provide the triples causing the inconsistency as an "Explanation". The Pellet reasoner is the most well-known, but other reasoners, such as HermiT and FaCT++, are also available.

Frameworks and APIs For programmatic access to ontologies, there are also some frameworks and APIs available. Enumerating all of them across different programming languages would be challenging.

[3]These editors are examined more thoroughly, particularly in comparison to ontology-driven form generators, in chapter 3.

For this thesis, the Java-based solutions OWL API and Jena are particularly significant. OWL API is a powerful API to work with OWL ontologies, while Jena is a more general framework for working with RDF. This means that Jena can be used to construct simple RDF graphs or RDFS ontologies, while the OWL API is specialized for OWL ontologies [The24b].
Additionally, Jena offers more features than the OWL API, including its own SPARQL engine and a built-in triple store. Further details on these frameworks will be provided in chapter 5.

2.4 Ontology Engineering and Evaluation

Before the aforementioned techniques for reasoning on ontologies can be applied, the ontologies must first be created. This discipline of creating high-quality ontologies is called Ontology Engineering, and is more complex than it initially appears. It is comparable to the process of Software Engineering [Hit+08], as both involve the creation of complex products (software or ontologies) and face challenges in evaluation, given that there is no single perfect solution. Consequently, the first step in Ontology Engineering, similar to Software Engineering, is a thorough requirements analysis [Hit+08]. These requirements in ontologies are referred to as "competency questions". These are natural language questions that encapsulate the knowledge to be represented by an ontology [Wis+19]. They function like queries that should be executable within the ontology [GU96]. The more given competency questions an ontology can answer, the better it fulfills its purpose and represents the domain knowledge. These competency questions can also help determine if extensions to the ontology are necessary.

However, as Gruninger et al. describe, formulating effective competency questions is not straightforward, as simple "lookup" queries are not sufficient, and a set of also more complex questions, which build upon the results of simpler competency questions, is necessary [GU96]. This process naturally requires individuals who are experts in the relevant knowledge domain, often referred to as "domain experts" in various studies [Sch21][Law19]. Once these informal, natural language questions are articulated, the next step involves formalizing them so they can be executed by a computer [GU96].

After these competency questions are established, the actual ontology can be created. The techniques employed vary significantly depending on the basis from which the ontology is constructed [Hit+08]: This could range from undocumented human knowledge to unstructured or even structured text from which concepts are extracted. For example, in a project where a huge medical thesaurus was converted into an ontology, a 4-step ontology construction, including automatic generation of definitions, automatic validation for consistency and also manual revisions, was applied [HS04]. For this thesis, the required knowledge is provided as human knowledge. However, the focus is not on building a complete ontology from scratch but rather on extending an existing ontology with a few additional concepts.

Evaluating an engineered ontology besides using competency questions is also a complex task. There are some suggested methods [Hit+08]: One straightforward approach is to logically check if the ontology is consistent and free of direct contradictions. Also, there are some methods to structurally validate ontologies, like the manual validation method OntoClean. For this, classes and properties are assigned special metaproperties, which are then used to impose constraints. These constraints help restructure the concepts with different dependencies [GW09]. All these steps require human interaction to assign properties and resolve any inconsistencies identified by the assignment.

It shows that this field of Ontology Engineering shares a fundamental challenge with Software Engineering: the technical aspects of building ontologies or software are intertwined with a highly specialized domain. So, experts in ontology languages and logic, which can be quite complex as demonstrated by OWL, are needed alongside domain experts who understand the domain well enough to formulate competency questions, establish general rules, and evaluate the engineering process's outcomes. This division between domain experts and ontology experts is often described as a bridge that must be crossed in ontology engineering [Sch21]. This is the most important premise of ontology engineering, this thesis will build up on: That the user group of ontologies is divided into two distinct groups that require tools to collaborate more efficiently.

It is also worth noting that, in addition to the broad term Ontology Engineering, the term Ontology Population is used in some literature. While in Ontology Engineering the goal is to build the complete ontology structure with all concepts and rules, Ontology Population is simpler, focusing on adding new instances of classes and properties without updating the ontology's structure [BT+21]. So in database terms, while Ontology Engineering is about defining the data structures for a new database, Ontology Population is about inserting new concrete datasets into an existing structure.

Chapter 3

Ontology Driven Form Generators

In this chapter the current state of form generation based on ontologies is examined. To achieve this a literature survey is conducted to explore the existing research and analyze the results.

3.1 Survey Questions and Goals

The primary question of this survey is, which ontology-driven form generation tools currently exist. The subject of this survey is therefore on already existing research papers, which discuss and implement such tools.
Before delving into the concrete survey, this term "Ontology-driven form generator" should be defined clearly: In this survey it is defined as a tool which can produce forms, especially HTML/web forms, based on structures of any given ontology. These forms can then be used to insert research data. The critical aspect is that these forms should cover the whole work lifecycle with ontology-based research data: That means that, first, the creation of forms should depend on an ontology with form fields based on ontology elements. However, it is not necessary that the complete form is generated automatically by the tool itself. Instead, also manually created forms are allowed, but then the tool should still connect the created forms to the ontology in some way. Second, filled-out forms should be included in the knowledge base constructed by the ontology. Both are important requirements for any ontology-driven form generation tool, as defined for this thesis.

The main goal of this survey is to identify existing ontology-driven form generators. After providing a general overview of the work in this field, it is then examined, which functions are implemented by these tools, in addition to the architecture and implementation of the basic form generation. The main set of interesting additional features is, as stated in the motivation of this thesis, about the support of developing the underlying basic domain ontology into a more refined application ontology. To support this, features about extending the given ontology in the tools are important,

for instance, the addition of new ontology elements like new subclasses and properties, but also the ad-hoc creation of new individuals. It also examined, if other types of ontology modification are supported by the tools, such as deletion and modification of ontology elements.
Aside from this set of features it is also examined if other features regarding flexibility, usability and validation of the knowledge base are implemented. The specific features are detailed in the next section.

3.2 Methodology

The survey follows these steps, which are explained in detail below:

- Collection of survey candidates
- Filtering the candidates for relevant papers
- Investigation of the relevant papers for workflow and additional features

3.2.1 Collection of Papers

Papers are collected using keyword-based search requests in specialized research search engines. In particular, the following engines are used:

- Google Scholar
- DBLP
- SCOPUS advanced search

Using multiple search engines is crucial for two main reasons: First, they all have different indexes, and therefore can search in different paper collections. Not every search engine has every paper, as various studies show [Fat+20]. So, this mixed usage of search engines is very important, especially in this relatively small research topic of ontologies, compared with other, more popular, computer science topics, for example ML research, regarding the number of developed papers.
Second, all engines use different algorithms to rank papers for relevance.

Google Scholar is used as a baseline due of Google's extensive index of electronic resources, and the company's experience with search engines. However, there are disadvantages, for instance the low index update frequency of only once a month [Fal+08]. To also obtain very current research, the search engine SCOPUS is used.

Also, this search engine has a high number of journals and in general a broad scope of papers up to 1966 [Fal+08].
Finally, the search engine DBLP is used as a German search engine specifically for computer science literature. Another advantage of DBLP is its public funding by the Leibniz Center for Informatics, in comparison to Scholar and SCOPUS, which are both owned by corporations [dbl24]. Additional search engines, especially for old literature, are unnecessary, because, as detailed later, only RDFS- and OWL-based tools are researched, and so only papers from 2004 onwards are relevant for the survey.

For each search engine, the following keyword combinations are used:

- Ontology based data acquisition
- Ontology driven form generator
- Web-based Data Acquisition ontology
- Ontology population with web forms
- Ontology form acquisition
- Ontology driven forms

Only the first 50 search results per combination are considered due to diminishing relevance with each result.

To obtain more results, a relative search around each relevant result is performed. This means, all papers which cite this result and are cited by this result are included as survey candidates. With this tactic a forward, as well as a backward search is performed, so that, as far as possible, all relevant research is found. The forward search is performed with the specific feature in Google Scholar.

Despite their popularity, Large Language Models (like ChatGPT or Google Gemini) are not used to obtain research papers, although defining prompts for this task is theoretically possible. This is due to the danger of hallucinations, where LLMs could possibly "invent" research papers [AM23], and the general obscurity of how the LLM generate result lists. Additionally, in general LLMs are not up-to-date, as their input data is usually several months to years old, lacking the most recent research.

3.2.2 Filtering for Relevant Papers

After collection, the found papers are filtered to retain only those relevant to this survey. The imposed requirements for relevant papers are set up in two categories: Must-requirements, which have to be fulfilled by each paper to be considered in this survey. And should-requirements, which designate more relevant papers. Papers meeting both must- and should-requirements are considered “primary findings” and analyzed in detail, while those meeting only the must-requirements are considered “secondary findings” and analyzed more broadly.

There are three must-requirements: First, each paper must describe a tool, which represents an ontology-driven form generator. That means that forms in any technology, either web based or others, with different form fields consisting of text fields, number fields, date entries, etc. should be the main user interaction, in contrast to, for example, free text editors. These forms must in any way be connected to the ontology, either automatically generated or in other ways based on components of the ontology. And, very importantly, filled forms must then directly contribute to a semantic knowledge base set up by the given ontology, so the ontology is “populated”. In this way, the workflows of the presented tools, must completely depend on ontologies.
The second must-requirement is that only tools which cover RDFS or OWL ontologies are included. As already explained, OWL, and in less complicated use cases also only RDFS, are the standard languages in the Semantic Web context and have a high relevance in this area. But, because the prototype will be developed with OWL, as detailed later, and OWL introduces even more structure elements for ontologies, the focus is set on OWL tools, and therefore supporting OWL ontologies is a should-requirement in this survey. Tools which aim for other ontology languages, especially not defined or specifically created ones, are not considered.
The third must-requirement is that the contribution of the form results to the knowledge base should mainly be populating the ontology. This is to differ against generic ontology editor tools, as here populating the ontology is the main goal.

Besides the already mentioned OWL requirement there is one other should-requirement: The ontology, on which the forms are based, and which is populated by filling them, should be interchangeable, or at least in theory interchangeable, although the tool may have a specific ontology in mind. In this thesis the focus is on generic form generators, which should support any ontology and should not depend on given constructs.

The requirements are summarized in Table 3.1.

No further requirements are set on popularity, relevance, or the timeframe of the

Must-requirements	**Should-requirements**
Tool is an ontology-driven form generator in its basic functionality	OWL ontologies are supported
At least RDFS ontologies are supported	Used ontology can be (theoretically) changed
Population of the knowledge base is the main functionality of the tool	

Table 3.1: Overview of the survey requirements

research. As detailed in the upcoming section, the research topic around ontologies is relatively small compared to other computer science topics, making additional restrictions unnecessary. Measuring a bottom line of necessary relevance in this area is also challenging.
Although the focus in this work is on web or HTML forms, there are also no requirements on how these forms are implemented in the projects.

3.2.3 Investigation of Found Papers

The relevant papers are then investigated in detail, highlighting the following aspects of each system:

- The main architecture and the goal of the tool
- How the forms are constructed, for example manually or automatically, and how the forms are saved
- The handling of filled-out forms, and how the knowledge base is populated
- Additional implemented features

In detail, the following features are examined:

- Possibilities to expand or modify the ontology, such as adding new classes or properties
- Entity matching and collection from the already existing semantic base, for example, to fill out object properties
- Logic in forms, such as conditional questions that reveal other ones
- Optional fields

- Note fields, or other means to add additional information
- Validation features, so that newly constructed data remains consistent with the ontology and the knowledge base
- Other features that may improve usability and flexibility

The first feature contributes to the topic of ontology extension, while the other ones are more about usability and flexibility.
The survey will investigate the tools feature-wise and their main form-handling architecture. It will not go into detail about the exact technical implementations of the tools, summarizing only the implementation of interesting features and the general tech stack. Any more detailed technical analysis would be beyond the scope of this work, as the survey's main question are about what features are implemented, not how.

3.3 Results

3.3.1 Overview

The general outcome of this survey indicates that the research area of ontology-driven form generators is quite small. Despite using various search methods, only a few relevant papers could be found. In total, only eight papers were found which fit all must- and should-criteria, making them highly relevant for this survey. Besides those there are four other projects, which fulfill the must-criteria but not all should-criteria. These will also be analyzed in this survey, but are not as detailed as the other ones.
One of the primary findings can also be categorized as Semantic Wiki software. However, because the form features of this system fit very good within the given criteria, it will still be considered here. It is explained in specific paragraphs later, why Semantic Wiki systems, and also ontology editors, generally are excluded from this survey except for this one tool.

Overall, this is a very small number of papers, especially in comparison to other research areas in Computer Science, and the broader field of ontologies. Specifically, there is a greater volume of papers about other means of data acquisition for ontologies and semantic knowledge bases, especially using automated methods, but only few papers about a form-based approach.

Also, the citation numbers of these papers are very low, further highlighting the niche nature of this research area. For example, among the primary findings, OWL2MVC

[AA20] has the highest citation count with only 22[1]. In contrast, other papers which deal with data acquisition in ontologies often have higher citation counts, sometimes reaching three digits, such as [LYR09] or [DZP10].

A significant challenge in finding these papers is the terminology. The term "Ontology-driven form generator" is not a widely accepted term, requiring the use of various keywords in search engines to find relevant papers, as the developed tools are all described differently in the research literature. Many search results are also irrelevant to this survey. For instance, a huge research direction which is also covered by these keywords is the automatic population of ontologies using a text corpus, which is not the focus of this survey and thus not considered here.

The following paragraphs will analyze the most relevant form systems in detail.

3.3.2 Primary findings

OBOP by Rutesic et al.

The first research project is OBOP by Rutesic et al. [RRSP21]. This tool servers as a very good example for an ontology-driven form generator, as it features many elements common to other tool, making it worthwhile to explain this tool in detail.

Architecture, input and output, form structure The tool was implemented using JavaScript in the React framework, and therefore creates web forms interpretable with a browser. The basic functionality, which is also similar to some other tools, is as follows: The workflow starts with the import of an OWL ontology, after which forms can then be manually created based on this ontology. Each form targets an ontology class. The form fields are properties, where this class, which is in this thesis from now on referred to as "target class", is the domain of these properties. The properties can be object or datatype properties.
The generated form can then be filled out, resulting in the creation of a new individual of the target class, and the values of the form fields become respective property instances for this new individual.
This generator uses an own ontology named OBOP, in which the forms are constructed and saved. To create forms, the user creates RDF triples in OBOP corresponding to forms and form elements. An architectural disadvantage is that for each target class an already existing individual is needed, which is used as pattern for new

[1]Number was determined via Google Scholar; as of 09/05/2024.

individuals. If the knowledge base does not have an individual for a chosen target class, it has to be created first without the tool.
The output of the tool is then a knowledge base with the created individuals and properties, depending on concepts of the domain ontology.

As this all may seem a bit abstract, an example, which is also given in the paper [RRSP21], is very helpful: In the example a form for restaurants is created with only one field: the name of the restaurant. This field, which can be found as an instance of “obop:field” in the OBOP knowledge base, has the property “obop:containsDatatype”, which references the class “gr:legalName” of the imported domain ontology. So, as the OBOP ontology contains this property, the tool renders the form with this input field. But, it seems so, that currently only text fields are supported, as no other datatypes and fitting form fields are mentioned in the paper. The form itself is connected to the pattern instance with an object property “obop:modelBelongsTo”. The pattern itself is of class “gr:Restaurant”. When the form is filled out, a new individual of the class “gr:Restaurant” is created, and the property “gr:legalName” is filled out with the given name.

OBOP concepts do not appear in the output knowledge graph, only the connected domain ontology concepts are used.

Additional features The paper does not explicitly state whether OBOP includes an entity matching or searching mechanism: For object properties the label/name of the target individual has to be entered, and if it does not exist yet, the tool creates a new form [RRSP21]. But it looks like there is no mechanism to help the user with searching usable individuals. That means, existing individuals can be used, but the tool offers no support in finding them. So a user must have the ontology and its knowledge base in mind.
A significant focus of this project is on complex form elements. The tool, for example, supports conditional questions, depending on given answers, and also loops parts of the forms for questions with multiple answers. As mentioned, the forms are all saved as ontology knowledge bases themselves, and these complex functions are modeled in OBOP using special classes (e.g. a loop class), and predefined properties that must be filled out correctly, so that the system can interpret them when rendering the form.
The tool cannot modify the domain ontology ad hoc, as it lacks features in this area. So, for changes in the ontology a change outside the tool is needed, and the ontology must be re-imported. Optional fields or note fields also seem to be missing in this tool, as no details or examples of such mechanisms are provided.

FACSIMILE by Gonçalves et al.

FACSIMILE was created by Gonçalves et al. from 2015 to 2017 [Gon+17]. This work presents an ontology-driven data acquisition tool designed for general purposes based on OWL. Unlike some other tools in this survey, FACSIMILE is not limited to a specific use case but is intended to be used with any domain ontology.

Architecture, input and output, form structure The tool is a Java-based web application, and the forms are presented as HTML web forms. One important aspect is its complex architecture. Besides the user-imported domain ontologies, FACSIMILE uses an own "datamodel" ontology, which includes classes such as Question and Section to describe the contents of the forms. However, to describe the structure of the forms, including form fields, question order, and input types, an XML file is used for each form. Thus, not all necessary information is stored as an ontology in this project. The XML file also describes the mapping between the form fields and the domain ontology.
Each form field in a form is connected to a property in the domain ontology, and the filled-out value is then added as an individual to the knowledge base. However, these created individuals do not target the domain ontologies. Instead, individuals for the datamodel ontology, such as the class "Observation", are created, and the filled properties, which belong to the domain ontologies, are added to this Observation individual.
This approach allows the tool to support multiple domain ontologies, provided as standard OWL/RDF files as input, in addition to the aforementioned XML. Strictly speaking, the domain ontologies are not directly populated; rather, the datamodel ontology is. Nevertheless, because the datamodel ontology is connected to the domain ontologies, the filled-out forms can still be considered as data acquisition for the domain ontologies.
The forms themselves are created by users using the XML files, and they support various input types, such as free text, select fields, and others. The output, which is the observation knowledge base, is subsequently delivered as a standard RDF file.

Additional features The basic requirements envisioned for this tool are met, and it also includes some additional features: Notably, it supports the inclusion of multiple domain ontologies simultaneously, as the mapping in the XML file is not constrained to a single ontology. Furthermore, the forms can include conditional questions that are only displayed when another question is answered in a specific way. However, it lacks features for ontology extension: It is not possible to expand the ontology, and it does not include an entity collection or matching feature: While it is possible to link existing entities to question answers, these entities must be explicitly specified in the XML file or the datamodel ontology. Automatic extraction of fitting instances from the ontology does not occur, and validation features based on ontological constraints are also not mentioned in the paper.

OWL2MVC by Aydin and Aydin

Another notable example of an ontology-based form data acquisition tool is OWL2MVC by Aydin and Aydin, released in 2020 [AA20].

Architecture, input and output, form structure This is also a very general tool compatible with any imported ontology, creating web forms based on an ASP tech stack, although the research focus here is on agricultural ontologies. Their research focus specifically is open data and open ontologies. Similar to FACSIMILE, OWL2MVC is based on OWL ontologies.
A disadvantage of OWL2MVC, as with FACSIMILE, is that filled forms are not added directly to the knowledge bases of the domain ontologies. Instead, the forms target a separate knowledge base where the filled forms exist as entities, and the filled attributes are created as instances connected to the forms. This maintains the relationship between the form and the filled value, but the new information is not integrated into the original ontologies.
The forms in this tool are again created manually by users. They can contain various data types, which are represented by different input types, again based on HTML, as this is the form output format, like in OBOP and FACSIMILE.
The input for the tool is an OWL ontology provided by the user as the domain ontology, and, similar to OBOP, a predefined control ontology containing form information. The form structures themselves are stored in a traditional SQL database. Ultimately, an RDF export of the knowledge bases is possible. Here, the knowledge bases also include the forms themselves, similar to FACSIMILE.

Additional Features The tool implements dynamic entity collection. For example, it is possible to use a class as select form element and let the tool collect all instances of this class, which are then offered as options for this form element. Aside from this, the created forms are simpler than those in FACSIMILE: Conditional questions, optional fields, or other additional logic are not possible. It also again lacks any capabilities of expanding the domain ontologies, and validation or the use of ontological constraints for the forms is not mentioned in the paper. Instead, constraints are only derived from the data types themselves. Regarding additional features, it

remains somewhat basic, but the entity collection feature is very useful for good usability.
Further research is not possible, as the provided website is no longer functional, and thus neither the agricultural ontology nor the tool itself can be examined directly.

CARDINAL by Langer et al.

In 2021, Langer et al. published a paper on their ontology-based form generator, CARDINAL [LGG21]. The project's motivation is to create structured metadata organized by an ontology through web forms, rather than static free text. A practical example is a submission form for research, such as conducted surveys. The form generation aims to include special properties depending on the research artifact, such as whether it is a paper or a conducted survey.

Architecture, input and output CARDINAL is also built as web app using the Python-based framework Django. A key difference from other tools is that CARDINAL does not support multiple creatable forms. Instead, it has one basic form with a static part that remains constant and a dynamic part defined by the chosen ontology. Like FACSIMILE, it natively supports OWL ontologies and was not developed with a specific one in mind. However, it is not possible to immediately import and use any ontology; the form part must first be created outside the tool in the so-called OnForm ontology. This is a special ontology created for this research project. Although there is no specific paper on OnForm, some information can be gleaned from the project's code repository [Gö21]: OnForm is comparable to the datamodel ontology of FACSIMILE and consists of form element classes and structure elements. For a specific domain ontology, instances of these classes must be created and connected with properties from the domain ontology via annotation properties in OnForm.
This process allows the dynamic part of the form to be created, with concrete fields filling out properties of the domain ontologies and creating new individuals. Also, ranges are used to create form fields and differentiate data types. User-inserted data can be exported as RDF triples in formats like Turtle or XML. In detail, the OnForm knowledge base, which keeps the questions and forms as individuals themselves, is used for the export. Thus, the workflow is similar to FACSIMILE.

Additional Features A notable feature of CARDINAL is that the tool itself decides which form to use, based on information from the static form part, rather than the user making the selection.
Aside from this, this tool does not have many special features. It includes an entity

lookup system that extracts suitable individuals from the domain ontology. However, it lacks complex form structure elements or logical elements, does not support multiple ontologies simultaneously, and does not have features for ontology expansion. Constraints are also not mentioned.

Vcelak et al.

Another prototype was developed by a research group at the University of West Bohemia in 2017 [Vce+17]. The motivation of this tool is similar to the others: to provide a user-friendly interface for acquiring semantic data without requiring users to have ontology knowledge.

Architecture, input and output, form structure The tool is again a web application that generates HTML forms using the Thymeleaf framework, a Java-based server-side framework for building HTML pages from templates. It follows the typical setup of creating individuals per form, with form elements representing properties of these new individuals. As input the tool can consume multiple ontologies.
This tool does not allow users to create their own forms but instead generates them entirely from the ontology for each given class. Some user modifications are possible through annotation properties in the imported ontologies, which can be used to change data types of fields instead of using the ontology's range value. Certain properties are also directly interpreted as HTML properties, for example to hide some form elements or to make them read-only. This means that no additional configuration for the forms is needed, as the configuration is directly linked to the ontologies themselves.
The output of the forms is directly translated into RDF triples, which can be queried via SPARQL.

Additional Features A notable feature of this tool, compared to others, is the ability to edit already inserted data. Since user-created forms are not supported, this feature is implemented by filling out the corresponding form of the class with the inserted data from the knowledge base, allowing users to make changes. Overall, this tool is relatively simple, lacking features for ontology modification or complex form logic. One other unique aspect is the use of additional constraints besides those needed for the basic form generation: For example, besides domain and range, also cardinality and multivalue constraints are used in the form. However, no other complex constraints seem to be supported.
The implementation of entity collection and matching mechanisms is not clearly

described in the paper: While it is possible to link other individuals as object properties, it is unclear if a search in the knowledge base for existing, fitting individuals is possible.

ActiveRaUL by Butt et al.

ActiveRaUL, developed by Butt et al., is another OWL-based form generator, although it is advertised as a more generic RDF populator [SB+13]. But it generates forms based on an RDFS ontology, and also includes OWL elements, although not many details about this service are given in the two published papers.
The primary motivation is to support users without ontology experience in populating knowledge bases. As the other tools, it is again a web service [HR10]. The tool supports user-created forms but also has a complex logic for automatic form generation based on defined properties. This involves transforming the entire ontology into an internal graph representation, which is then used to generate forms. Input is provided via RDF files containing the ontologies.
Also, it has an entity lookup mechanism for finding fitting existing individuals. However, the focus of this work is less on the forms themselves and more on form generation and information extraction from the ontology. But more details, especially about including restrictions and axioms from the ontology, are not given. Also, again no ontology extension feature is included, the tool only covers the population of knowledge bases [SB+13].

DESIREE by Sadki et al.

Another, rather basic, form generator built on web tools and Jena-Fuseki was created within the DESIREE project [Sad+18]. This tool shares many similarities with the tool proposed by Vcelak et al.: It automatically generates forms through a multi-step process, beginning with the extraction of all relevant properties for the target class. And similar to Vcelak et al., the tool relies on annotation properties within the ontology, thereby mixing form and ontology information. Aside from this, the typical features are supported: the knowledge base can be exported to RDF triples, data ranges are used to create form fields. Also, the tool supports editing of instances. However, it lacks additional features such as ontology extension, validation, or complex forms.
Overall, the generator's details and features are not extensively documented. The lifecycle and the target of the forms (instances of arbitrary classes, as in other tools) remain unclear.

3.3.3 Secondary findings

PRiSMHA by Goy et al.

Another ontology-based form generator is the 2020 system PRiSMHA by Goy et al. [Goy+20]. Similar to OWL2MVC, PRiSMHA is part of a larger project where the form generator is just one component. The primary objective of this project is to crowdsource information about historic events. Volunteers can add contextual information about historic events by filling out forms, including details such as relevant persons, locations, and times. To support this, a dedicated OWL ontology named HERO was created alongside the form generator. The purpose of this generator is that it should enable people with no understanding of the HERO ontology to add information to the knowledge base about historic events.

Architecture, input and output, form structure The system architecture consists of a client-server model with Spring Boot as the server and a JavaScript-driven interface. As data storage, MySQL, a relational database, for forms and other data as well as Jena TDB as triple store are used. The functionality of the tool is similar to the other tools: Each form creates one instance, with fields generating object and datatype properties. This knowledge can be searched, and an RDF export is planned but not yet implemented.
Again the forms are automatically generated. This works here because for the different kinds of historic events, which are modelled as classes, a survey was conducted to find out the most relevant properties, which are then presented first to the user when filling out forms. The relevance is determined by user behavior during form completion. This also explains why this project cannot be used directly with any other ontology, and therefore only is a secondary finding, because this approach is tightly coupled with the HERO ontology and its properties. The crowdsourcing aspect of the project, with a large number of users, was essential for acquiring the necessary statistics for property usage.
The only process comparable to a “form-builder” is a feature that allows administrators to disable certain classes and properties of the HERO ontology, preventing them from appearing in the form.

Additional Features Due to the design of automatically generated forms, they do not include complex logic or conditional questions.
The entity collection and matching functionality for filling out object properties is noteworthy, as it includes a search function to find suitable individuals. However, the search appears to be limited to individual labels, with no sophisticated search functionality also including annotation properties. This information is uncertain, as

neither the paper nor other PRiSMHA resources, including the project website and additional papers [Zin20], provide detailed information on these topics. Unfortunately, the implementation of PRiSMHA is not available.
Other special functions regarding optional fields or ontology extension are not implemented by PRiSMHA. It is also unclear whether the forms support data types other than text input.

K-Forms by Bhagdev et al.

K-Forms, developed by Bhagdev et al., is another interesting tool [Bha+08]. At first glance, it appears to be an ontology-based form generator: It includes functions to create ontology-based forms, and these forms instantiate RDF data. However, a significant difference is that these forms are not based on a domain ontology but instead create one from scratch through form creation. The tool uses form templates, which are groups of forms, representing entire ontologies, with individual forms corresponding to classes. Instances of these forms are then again instances of the classes, resulting in a different workflow: Instead of creating an ontology first and then forms to populate the knowledge base, the ontology and forms are created simultaneously. Therefore, this project is ranked here as secondary finding.
The implication of this architecture is that queries and use cases can only be created after the forms are established: It is not possible to view the ontology and its knowledge base independently of the forms.

RDFS-only Editors

FORMULIS, developed by Maillot et al., is a tool that only uses RDFS and does not include OWL elements [Mai+17a], for instance classes, domain, and range. Also, it shares various similarities with the other tools and includes some advanced features: Each of the forms, created and modifiable by users, corresponds to a new individual of a target class. Individual searching is implemented based on the range value of a field, allowing the tool to search the knowledge base. Additionally, it supports nested forms for individual creation.
A notable feature of this tool is its ability to suggest how to fill remaining form fields once some fields are completed. This is achieved by using SPARQL queries against the knowledge base, with empty fields set as variables to select existing values in the knowledge base [Mai+17b]. Thus, this tool shares many similarities with others, including the primary motivation of user-friendliness [Mai+17b], despite not using OWL.

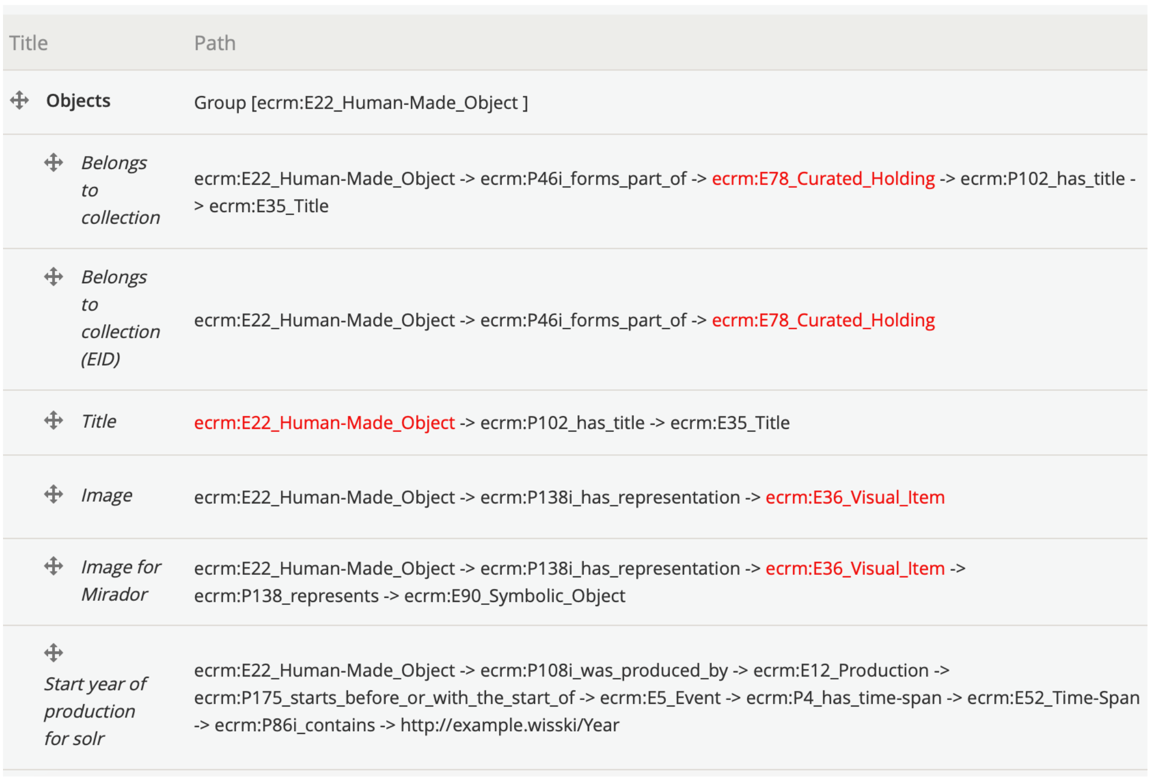
Another RDFS-only editor is RIC, developed by Kalyanpur et al. [Kal+02]. This tool is a very simple instance editor, which allows to directly edit properties of instances. However, it does not use real forms, functioning more as a general editor that represents instance properties in a form-like manner. This simplicity is due to its lack of support for OWL or RDFS subclassing, eliminating the need for complex reasoning.

3.3.4 Ontology-based Wiki software

In the context of editing knowledge bases based on ontologies, there exists a category of software products known as ontology-based wikis. These tools aim to combine the advantages of wiki software with the technologies of the Semantic Web: Just like wikis, ontology-based wikis (or semantic wikis) are collaborative systems, which can be used to create websites. The special feature is that at some points, links to other websites via ontology properties or other metadata can be established enabling automated processing by computer [Sch+07].
Generally, ontology-based wikis do not fit into the category of form generators, as editing in wikis typically occurs via a free text editor or by directly editing individual data without a form [Sch+07].

An example of this is Semantic Wikipedia [KV09]. In this software, content editing is similar to standard Wikipedia, using a free text editor. Additionally, the formal language of Semantic Wikipedia allows connecting properties and other individuals of a knowledge base to each article. By this, a semantic network is created alongside the Wiki articles. However, this approach does not align with the anticipated form-based solutions.
Another example that leans more towards a form-based approach is Ontowiki [Gar+12]: Ontowiki allows for the editing of an RDF knowledge base, where each webpage describes exactly one class instance with all its properties. However, this editing does not occur via pre-created forms, but instead by directly editing properties of instances. This approach is somewhat similar to form-based solutions but does not fully align with them. Especially the described widgets are not comparable to form elements but serve as HTML templates for certain data types, depending on the properties. On the other hand, Ontowiki shares similar goals with many form generators, particularly in terms of simplicity, as technical RDF specifics like IRIs are hidden. It also supports the export of the knowledge base and the import of arbitrary ontologies.

Especially the first example demonstrates that semantic wikis cannot be used synonymously with form-based generators, as their applicability depends on the goals of the individual software products.
A more fitting example, which actually can be categorized as a primary finding, is WissKI. WissKI builds upon the Content Management System Drupal, and enables



Title	Path
✣ **Objects**	Group [ecrm:E22_Human-Made_Object]
✣ *Belongs to collection*	ecrm:E22_Human-Made_Object -> ecrm:P46i_forms_part_of -> ecrm:E78_Curated_Holding -> ecrm:P102_has_title -> ecrm:E35_Title
✣ *Belongs to collection (EID)*	ecrm:E22_Human-Made_Object -> ecrm:P46i_forms_part_of -> ecrm:E78_Curated_Holding
✣ *Title*	ecrm:E22_Human-Made_Object -> ecrm:P102_has_title -> ecrm:E35_Title
✣ *Image*	ecrm:E22_Human-Made_Object -> ecrm:P138i_has_representation -> ecrm:E36_Visual_Item
✣ *Image for Mirador*	ecrm:E22_Human-Made_Object -> ecrm:P138i_has_representation -> ecrm:E36_Visual_Item -> ecrm:P138_represents -> ecrm:E90_Symbolic_Object
✣ *Start year of production for solr*	ecrm:E22_Human-Made_Object -> ecrm:P108i_was_produced_by -> ecrm:E12_Production -> ecrm:P175_starts_before_or_with_the_start_of -> ecrm:E5_Event -> ecrm:P4_has_time-span -> ecrm:E52_Time-Span -> ecrm:P86i_contains -> http://example.wisski/Year

Figure 3.1: Example of the Pathfinder component of WissKI

form-based population of a knowledge base [SG12]. This works in a way that the administrator selects an ontology and creates forms (called bundles) for different classes to create instances of these classes. Separately, the so-called Pathfinder component of WissKI connects each field in a bundle to a property in the ontology. Both components work independently, but if both (the configuration of the Pathfinder, and the correct creation of bundles) are done, for all created instances and their properties, fitting triples in the knowledge base are created. And as the analysis of a current version of WissKI shows, these fields can refer to other entities in the knowledge base, or be single data objects [Ger24b].
A unique feature of the Pathfinder is that it allows for complex property connections, called knowledge paths [FR12]. Unlike the other presented tools, the fields can not only be direct properties, but instead be chains by leading from the target entity to properties of other, connected individuals. Especially in a more complex ontology, it is not possible to get all needed information of an individual by reading all direct properties: For instance, in the given CIDOC ontology specifying a creator for an object requires adding a creation event entity, where the creator, time, and place of creation can be specified.
A concrete example from the current version of WissKI is shown in Figure 3.1. Here

"Objects" as a so-called Group represents an entire form or entity. The other entries below are all form elements with knowledge paths from the base entity class "Human-Made-Object". The advantage of Pathfinder is evident for "Start year of production": As in the knowledge base this information is connected to the object via Production, Event and Time-Span classes, users do not need to create forms for all these intermediate classes manually. Instead, they fill out the form and enter plain text for the start year, and the necessary intermediate objects are created automatically. So this feature is an important feature in WissKI to enable users to add entities via forms, although the underlying ontology may be not straight-forward.

However, WissKI is not an all-purpose, completely general form generator. One WissKI instance can only work with one ontology at a time and not multiple ones, and WissKI is very strongly tied to the CIDOC CRM ontology [Ger24a]. This is because WissKI was created with a specific domain in mind, namely cultural heritage, and not necessarily as an all-purpose tool [SG12]. Additionally, the creation and editing of forms in WissKI are hidden in administrator settings. These bundles should not be frequently modified, and the focus of end-user activities lies in creating instances via the bundles and Pathfinder. So it can be said, that while WissKI does not aim to be an all-purpose form generator, it includes many interesting elements for form generators, particularly with Pathfinder and the knowledge paths.

3.3.5 Ontology editors

Ontology editors should be distinctly separated from form tools. While they may share some similarities and features with ontology-based form generators, their primary goal is to modify the core ontology on which knowledge bases are built, although these can often also be modified. In contrast, ontology-based form generators focus more on ontology population.

WebProtégé

One of the most prominent examples is the web version of the well-known Protégé editor: WebProtégé [Tud+13]. One of the main motivations of this project is, besides releasing the classic Protégé desktop app as web application, lowering the entry barrier for new users who are just starting to work with ontologies. However, as with all ontology editors, it is expected that users have at least a basic understanding of ontologies and their terminology. As the core functionality builds directly upon the desktop client, this tool is primarily an ontology editor.
Interestingly, a simple knowledge acquisition feature with user forms is mentioned, but the paper does not provide much information [Tud+13]. Further research shows

that previous iterations of WebProtégé contained a module for creating forms to edit, but not create, individuals. This feature was very basic [Tud16], supporting both kinds of properties but lacking complex form features and the spontaneous creation of new individuals for property connections. It is also unclear what happens to the form if the ontology is edited in WebProtégé, as no mechanism for preserving the form is described.
In modern versions of WebProtégé, this feature is no longer a direct module but has been separated from the main program[2], indicating that it is not a main priority of the project but rather an additional feature.

IOPE by Baghernezhad-Tabasi et al.

In the case of IOPE, developed by Baghernezhad-Tabasi et al. this differentiation from form generators is not as clear [BT+21]. Here, the motivation is to use web forms to populate as well as evolve ontologies. This is achieved by displaying and accepting concrete instance information as well as possible changes to the ontology per form. The forms are completely automatically generated, with one form per class and per its domain property. The form lists the restrictions on the property, such as the range, allowing users to use a concrete individual or change the range class, thus enabling population and ontology changes. In the background, an internal ontology is used to render the forms, with mapping and binding rules to generate the forms from the ontology and then generate new RDF graphs from the filled forms, containing new individuals and changed OWL axioms. Any OWL ontology can be used.

Although this tool leans heavily towards general ontology editing, all changes are based on forms containing free text boxes, lists, and other form elements, similar to other tools. Thus, it is a mix of an ontology editor and ontology-driven form generator.

3.3.6 Unrelated Work

Aside from the presented tools, some tools found in the search criteria do not fit into this survey, although they initially seem like ontology-driven form generators. In other cases, the descriptions were too unclear to determine if they fulfill the requirements, and the source code was often unavailable for further research. For example, KME by Horridge et al. initially seemed like a fitting example. However, it showed that the output of this tool actually are ontologies representing forms which then have to be rendered into real forms by other tools [Hor+14]. Other tools use

[2]This was tested in WebProtégé 4.0.2 of August 1, 2020.

forms not for populating or evolving ontologies but only for selecting already entered data [HKB09].

Additionally, tools that only use RDF without an ontology context are not included in the survey. An example is Annalist, which does not use OWL or RDFS but only RDF for generating general RDF triples [KWP16]. Other examples include RDFedit [Poh14].

3.4 Discussion

The survey indicates that there are existing tools designed to generate forms from ontologies and to populate ontologies using forms. However, the number of such tools is limited, and some lack available source code or are built on outdated technology stacks, making them difficult to evaluate. Nonetheless, several noteworthy approaches have been identified.

Similarities between the Tools

The tools differ in many aspects, but also share some common features. The core architecture and lifecycle of these tools are quite similar: they typically start with the import of one or multiple ontologies, optionally accompanied by configuration artifacts. The output of these tools generally contributes directly to the ontology's knowledge base or to a special form knowledge base by generating RDF triples.

A common principle observed is the main architecture of the forms: each class in the imported ontology corresponds to a form, and the form elements represent the properties of that class. This is, for example, the case with OBOP, PRiSMHA, DESIREE and in the work of Vcelak et al. This form structure is logical, as data properties can be directly translated into form fields. For instance, string properties become free text fields, and datetime properties can use date selectors. Tools like FACSIMILE and OWL2MVC utilize this structure.
Translating object properties is more complex, as these properties target other individuals in the knowledge base. A common solution among form generator tools is to use a drop-down box offering suitable individuals, where "suitable" means that the property's range matches the offered individuals. Here the exact implementation varies from tool to tool: One issue besides selecting existing individuals is that it could also be chosen to create a new individual, which typically involves opening another form. This raises edge cases, such as the absence of a form for a required class. Most tools do not address this issue, suggesting that ad-hoc creation of individuals

is generally unsupported. Only for some tools, like OBOP, a solution is explicitly stated.

Another significant aspect is the general lack of consideration for additional restrictions and constraints of the ontology. As already explained, an ontology can describe constraints in a very detailed way with axioms. OWL already gives implicit constraints by the concept of properties, subclasses and so on. But there are also more explicit constraints possible, like cardinality restrictions, or also more complex restrictions like disjointness. Among the reviewed works, only Vcelak et al. consider more complex constraints, such as cardinality and multivalue constraints from the ontology. However, value constraints like "owl:allValuesFrom" or "owl:someValuesFrom", as described in the OWL 1 specification [Bec+04], are not addressed by any of the tools. This appears to be an open research question, as cardinality and value restrictions could easily be violated even with simple ontology population.

Differences

Besides those similarities there also is a large amount of differences between the particular tools. These differences often stem from the motivations behind the research projects. For instance, in PRiSMHA, the form generator is part of a larger project focused on crowdsourcing information, whereas in OWL2MVC, the tool itself is the primary focus, with an emphasis on open data. Additionally, differences can be observed in the technologies used to implement the tools, although web forms are a common focus. A frequently seen technology stack includes Java and HTML/JavaScript, as in FACSIMILE or PRiSMHA. However, more unique stacks are also employed, such as ASP.NET in OWL2MVC or a React/JavaScript-only stack in OBOP. While implementation details might seem irrelevant to the design of the applications, these examples demonstrate that the chosen technology stack directly influences design decisions, particularly regarding the forms and available input elements. This can be seen in OWL2MVC, where forms are entirely built using ASP control elements.

A major design difference is how the forms are compiled: Not in all tools, like for instance in OBOP, FACSIMILE or OWL2MVC, the forms are manually created by users, but instead in PRiSMHA or DESIREE forms are at least partially automatically generated. In this survey actually half of the primary findings are automatically generated, as shown in Table 3.2. This design decision impacts the complexity of the forms: user-crafted forms can include special conditional fields, as seen in OBOP or FACSIMILE, whereas tools with automatic form generation generally lack such features.

Tool	Ontology evolution	Entity collection	Form logic	Optional fields	Constraint validation	Autom. form gen.	Other/ Notes
OBOP	-	-	conditional fields, loops	-	-	-	-
FACSIMILE	-	-	conditional fields	?	-	-	multiple onto-logies at once
OWL2MVC	-	+	-	?	-	-	-
CARDINAL	-	+	-	?	-	+	comb. of static and dynamic form elements
ActiveRaUL	-	+	-	-	?	+	-
Vcelak et al.	-	?	-	?	+	+	indivs. editable
DESIREE	-	-	-	-	-	+	indivs. editable

Table 3.2: Implementation of additional features in the primary findings

Another significant difference lies in the storage and representation of the forms themselves. The tools employ various strategies for this. One approach is to save the form information also within ontology knowledge bases, for instance in OBOP and CARDINAL. In CARDINAL, for example, a dedicated "form ontology" is used, containing classes and properties for building and storing forms. FACSIMILE uses another approach: Here the form structure is saved separately in XML files, which references the imported ontologies at specific points. In OWL2MVC, the developed forms are stored in an SQL database.

Finally, there are numerous differences in the additional features offered by the tools. The features implemented in each tool have been discussed in previous sections. An overview is provided in Table 3.2, which illustrates the variation in additional feature implementation across the tools. Each tool emphasizes different aspects, such as editable individuals, constraint validation, or advanced form logic. The most common feature besides the automatic form generation is the entity collection and matching mechanism, found in ActiveRaUL, CARDINAL, and OWL2MVC.
A particularly interesting observation is the absence of ontology extension or specialization capabilities in any of the tools. Additionally, constraint validation is generally not implemented, except for the project of Vcelak et al., where some simpler constraints are considered.

Open Research Questions

The feature comparison in the previous section highlights a key open research question: the implementation of ontology extension during form filling. None of the surveyed tools include this feature, making it a challenging topic with no prior work or experiences to draw upon. It remains unclear whether this is a feasible feature for form generators.

Furthermore, general usability has not been a primary focus in any of the tools, often being only briefly mentioned. This is partly due to the lack of discussion on the separation of user groups, particularly those without extensive ontology knowledge. Most papers consider only one user group, typically ontology experts, and do not address the communication between different user groups.

Also, it was shown that this topic of ontology-based form generators is still an active research field. Many presented tools are quite recent, and there is also active research at the moment, for instance a PhD project about applying form generators in the geological domain [Qu22].

Chapter 4

Conception and Requirements of a Form Generator Prototype

In this chapter, the requirements and the concept of the prototype to be programmed are presented. As discussed in the survey, there are already some form generators available, but none of them address the topic of ontology extension, or rather specialization. Therefore, this will be one of the main focuses of this work. Additionally, other basic functional and non-functional requirements are outlined.

In summary, based on the details in the introduction and the results of the survey, the goals of this implementation are as follows:

- Developing a prototype of a form generator on a modern code basis with a focus on code quality and extensibility for the possibility of further research.
- Thereby also including features to improve usability, aiming to find approaches for a better general user acceptance.
- Researching the possibilities of ontology extension, in the sense of specializing domain ontologies into application ontologies when using the form-filling features.

The forms are structured similarly to the examples provided in the survey, such as OBOP, PRiSMHA, and the work by Vcelak et al. Each form is associated with a target class, and the completed form instances are subsequently mapped to individuals of that class. Each field is a direct property of the individual, implying that no shortcuts, such as those used in WissKI, are employed. For this prototype, it was decided to integrate the new individuals with the pre-existing ones into a single knowledge base, allowing for references to other pre-existing individuals, and for searching in a combined knowledge base of all individuals.

4.1 Basic Functional Requirements

The form generator tool is built on the assumption that there are two roles using this software: ontology experts and domain experts, as defined in chapter 2. The suggested basic workflow, similar to the tools presented in the survey, is illustrated in Figure 4.1. The ontology expert is responsible for importing the ontology and creating the forms. This role is experienced in OWL and has a basic understanding of the imported ontologies, their properties, and classes. Therefore, the expert can create and modify the ontology, import it, and create basic forms. However, due to the ontology expert's limited domain knowledge, mistakes and incomplete forms may occur, potentially missing some properties.
This is where the domain expert comes into play. The domain expert is responsible for filling out the forms. This expert has a deep knowledge about the domain, and is able to interpret and complete the forms. If the domain expert cannot enter all the information in the forms, they would need to consult the ontology expert, who might then even change the ontology by adding new concepts, and then add new fields to the forms. Without additional functions, the ontology must be re-imported, restarting the main workflow. Over time, this iterative process evolves the initial coarse domain ontology into a finer, concrete domain ontology that meets all the domain expert's needs.

With this workflow, the basic use cases of the tool can be defined. The main functions of the tool are:

- Ontology import
- Form creation
- Form filling
- Ontology/knowledge base export

The ontology import should enable the ontology expert to import an ontology from an RDF/XML file. The ontology should then be visible as imported on the homepage. It is important that this tool remains as general as possible, and therefore, unlike some other tools, does not make any suggestions about a basic ontology, or knowledge area where this tool will be used. Also, the tool must support OWL ontologies in particular, and not only RDFS ontologies, because of the already mentioned higher expressiveness of OWL, more features, and the status as de facto standard for Semantic Web applications.
Next, after the import, the expert should be able to create forms based on the imported ontologies. Other than in some of the tools from the survey like OBOP, the form creation should be actually a part of the tool, and not be done separately and then imported.

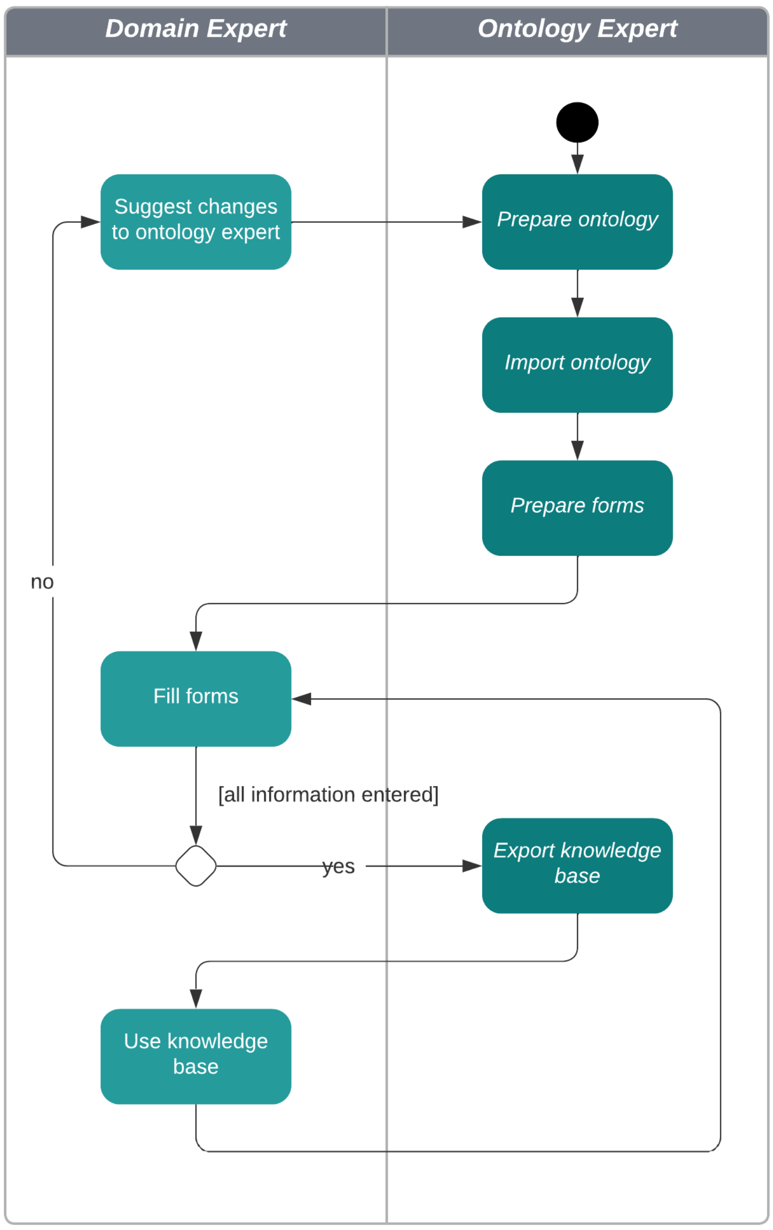


Figure 4.1: Basic form generator workflow between domain experts and ontology experts

Although it could be possible to create forms based on multiple ontologies, the prototype should support only one ontology per form due to complexity reasons. The domain expert should then be able to fill out the forms. The filled form instances should then directly be converted to individuals of the knowledge base. In this tool no differentiation is done between form instances and individuals, as both are the same. This is a simplification, which is debatable: Strictly speaking, a form instance is created at the initial filling of a form, while the resulting individual could be changed later on. Then those both entities are not identical anymore. But for this thesis this topic is not further relevant.
Another important aspect is that forms should remain editable even after instances have been created, allowing the ontology expert to add missing properties as needed. Finally, the new knowledge base containing the newly inserted data should be exportable to an RDF/XML file. This file should directly extend the imported knowledge base and thus also still contain individuals and concepts which were are part of the base before.
However, form terms should not be part of the knowledge base, unlike in some other form generators like FACSIMILE. This decision was done because of the focus of the tool in the original ontologies and knowledge bases themselves, and because of the focus on a later research data acquisition: The forms are only means to collect research data, and are not important themselves, like questionnaires. Therefore, they should not be part of the knowledge bases.

This concludes the basic main features of the prototype that together form the basic tool construct. Based on this foundation, additional concepts regarding ontology specialization and other usability measures are introduced.

4.2 Supporting the Creation of an Application Ontology from a Domain Ontology

A novelty of this tool, compared to other form generators, is its ability to extend the ontology, refining a coarse domain ontology into a more detailed application ontology. However, the tool should remain primarily a form generator rather than an ontology editor. Therefore, the extension features must be seamlessly integrated into the workflow of the domain expert. As illustrated in Figure 4.1, the domain expert's primary task is to fill out forms. This involves populating fields generated from properties and selecting individuals for object properties. This defines three main points for extensions:

- Adding new individuals when selecting one for an object property.

- Adding new fields to forms if additional information for an individual is required.
- Creating new properties, either datatype or object properties, as described in the OWL specification. In some cases, it may also be sensible to allow the creation of new classes as range classes, especially if the domain expert anticipates future utility.

Additional form fields will be implemented as temporary elements for the current form, while new ontology elements are permanently added to the knowledge base. This ensures that forms remain usable by other domain experts in their standard form. If a change is so important that it should be permanent in the form, then it would be the ontology expert's task to permanently include it via the form editor. Meanwhile, new ontology concepts can also be ignored once created, and in general cannot be made temporary, once an individual using these classes and properties is added to the knowledge base. The modified workflow is visualized in Figure 4.2. The workflow changes are shown in orange. The main difference is that the domain expert now has the ability for own changes to a certain extent. But it also shows that the ontology expert is still needed as a control instance, and to revert or edit proposed changes.

The exact features are detailed in the following sections. They are each introduced with a short user story, to make the feature more understandable from a user perspective.

4.2.1 Adding New Individuals

```
As a domain expert,
I want to add new individuals to the respective fields
    in the form while filling them out,
so that I can spontanously add needed entities not yet in the
    knowledge base without having to interrupt the current
    form fill process.
```

This feature is straightforward: while filling out object properties, the required individual for a form instance may not yet exist. The question is, how this situation can be addressed without interrupting the form-filling process to create the individual in another form, or even create one when no form for a specific class yet exists.

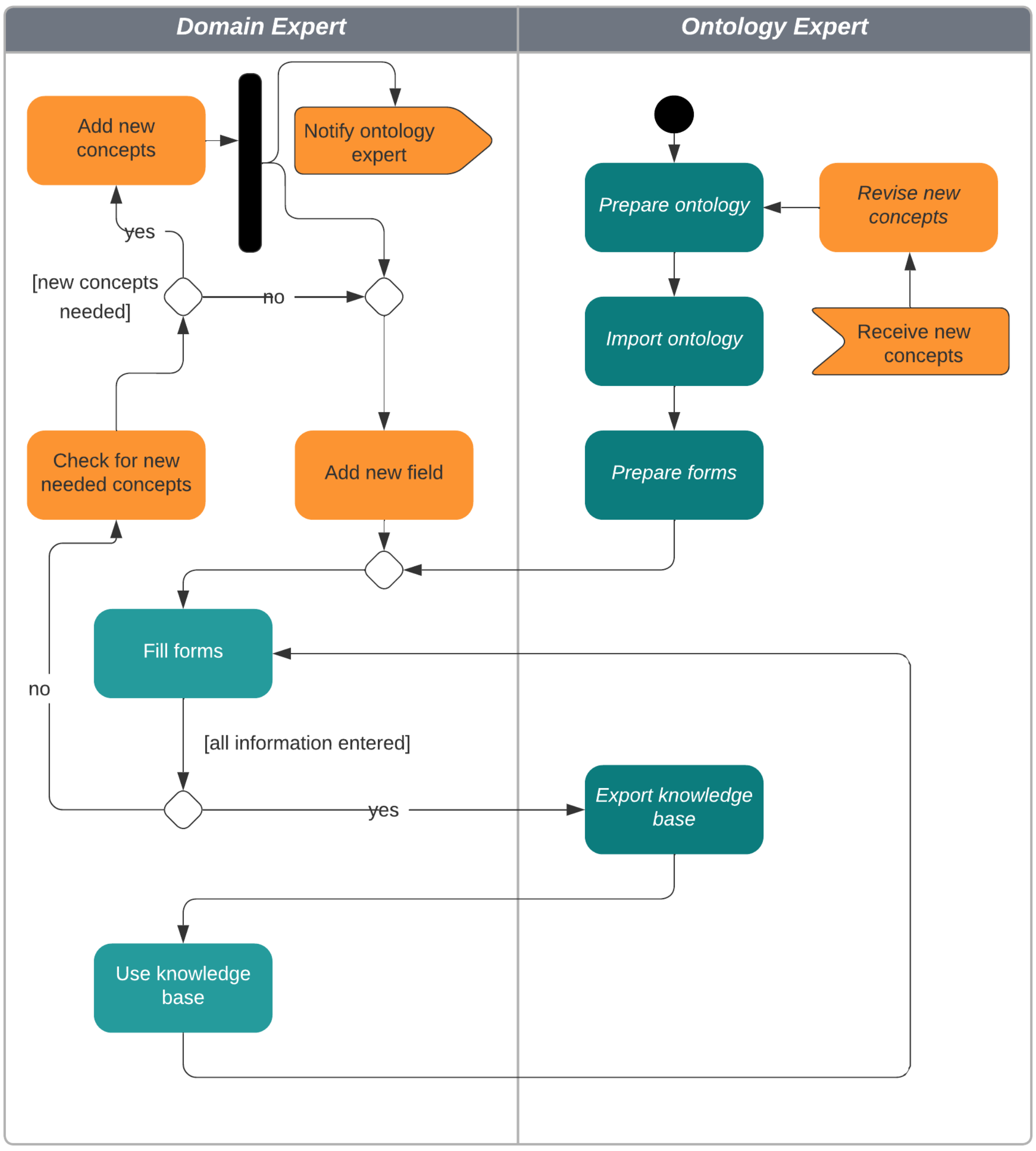


Figure 4.2: Revised form generator workflow with ontology specialization elements

4.2.2 Adding New Fields and Ontology Extension

```
As a domain expert,
I want to add new temporary fields to the form while filling out,
so that I can spontanously add new data to the knowledge base
    which is not yet part of the form, but should still
    be saved.
```

As already mentioned, the domain expert should be able to add new temporary fields to the form while filling it out. These fields should function just like normal form fields but should not be permanently added to the form, as they are intended for ad-hoc use. For significant changes in the requested data, a permanent change by the editor is needed.
That feature presents a challenge to the form generator concept: Typically, the line between the domain expert and the ontology expert is clear, as the domain expert should not interact with the ontology, only with the forms. For the sake of usability, this goal is relaxed, allowing the domain expert to interact directly with properties and classes of the ontology. This raises a whole new design problem, with the following main question: How to make these ontology elements properly visible to the domain expert so that they actually modify forms and add form fields without having prior knowledge?

The other part of this feature involves adding new concepts to the ontology.

```
As a domain expert,
I want to create new properties and classes in the ontology while
    filling out the form,
so that I can add new data to the knowledge base
    which is not yet part of the ontology.
```

This feature supports the specialization of the coarse domain ontology to a finer application ontology, allowing domain experts to create new properties and classes. The challenge is to make this feature usable for non-ontology experts while maintaining usability. Key questions include:

- How can this feature be implemented to be usable and understandable for non-ontology experts? This is especially relevant, as the domain expert normally does not have access to properties but only fields with potentially different names.
- What constraints are necessary to prevent the creation of nonsensical or incorrect properties?

- How can the creation of properties, that already exist in the ontology, be prevented?
- As creating properties is permanent, in contrary to the temporary form fields, how can be prevented that inconsistent properties are created?
- The existing ontology should not directly be changed by the new features, but it should be extended, while the original one also remains valid. How can this be achieved from a technical side, especially if the ontology is saved in a foreign, unchangeable namespace?

These questions highlight the differences between this tool and a generic ontology editor, as the restrictions necessary for ontology extension are not present in a generic editor. While ontology editors, as explained, are very general tools containing all possible functions for ontology and knowledge base editing, this tool should only allow for some very limited editing. The planned ontology extension impacts other features of the tool, as will be shown later.

In summary, the key difference between the two features, adding fields and adding new concepts, is that the former is temporary and form-bound, while the latter is permanent and ontology-bound. Because of the workflow definition and the stated entry point for change from the domain expert, adding new concepts only makes sense in the context of new fields in the form in this concept, as this is the only component of the form generator, which the domain expert is working with.

Especially the aspect, that the current quality of the domain ontology has to be kept up, and that the domain expert should not be able to create nonsense properties, is very important. Therefore, this should be considered a separate research point not covered in ontology-driven form generators yet. Therefore, this point will be the focus of the next section.

4.2.3 Consistency of the Knowledge Base

```
As an ontology or domain expert,
I want to ensure consistency of the knowledge base
    when adding new data,
so that it always remains compatible with the domain
    ontology and illogical statements are avoided.
```

Ensuring the consistency of the knowledge base is a critical aspect, especially in the context of ontology extension or specialization with new properties and classes. But also aside from that, ensuring consistency is not a trivial task, as adding new individuals can lead to inconsistencies too. For example, property restrictions and

domain and range specifications can be complex, as they may be inferred by subclass relationships. As a concrete example, consider the following: In the Wines & Foods ontology, which is later introduced, the class “DryWine” is defined as a subclass of “Wine” with the restriction that the property “hasSugar” must be “Dry”. If a form creating DryWines is created, it must be ensured that the domain expert cannot set this specific property to any other value, either in pre-defined fields or in added fields.
So, a more sophisticated consistency check on multiple levels is needed. Of course, as this tool should be used by non-ontology experts, this validation has to be automatic, and no human support from ontology experts should be needed.

4.3 Non-functional Requirements

Derived from the motivation of this tool of allowing further development in the future, it should be developed using a modern tech stack with actively developed frameworks and libraries, and state-of-the-art platforms. As many of the presented form generators are built on outdated technologies and have not been migrated to modern solutions, this is an important aspect of this prototype.
Additionally, the tool should be as simple as possible in both user interface and software design to enhance usability and facilitate further development. Finally, the tool as a larger software product should adhere to typical software quality concepts following ISO/IEC 9126–1 or ISO/IEC 25010:2011. Some relevant concepts are [Bal09, pp. 468 sqq.]:

- Fault tolerance: An error during runtime should not crash the entire program.
- Modularity: Individual components should be interchangeable.
- Comprehensive documentation: The code and APIs should be well-documented to improve the comprehensibility of the tool.
- Understandability and operability, especially for domain experts.

A special focus is laid on usability. Several features aimed at enhancing usability should be implemented. The most significant among these is a sophisticated system for individual collection and selection, also already implemented, for instance, in OWL2MVC or CARDINAL:

```
As an domain expert,
I want to receive a searchable list of possible selectable
    existing individuals when filling out object form fields,
so that I always know which entites already exist
```

```
    and I can directly search for them.
```

When working with forms, domain experts must be able to use existing individuals of certain OWL classes in the knowledge base, when filling out object properties. Therefore, a function that retrieves relevant individuals and allows the user to search them is required at designated locations in the program. This function should differentiate based on a given class parameter and only show individuals belonging to that class. That means that OWL reasoning in the backend is needed. Newly created individuals should also be included, so the collection functionality must work dynamically at runtime.
Additionally, the usability factor again is crucial, because especially in a huge knowledge base there could exist a large amount of individuals. A simple drop-down menu for selecting an individual would be impractical in this case.

Apart from this, other usability features inspired by form generator tools and general usability principles for electronic forms should also be implemented:

- Fields with multiple values, eliminating the need for multiple fields with different names. This feature is common in digital forms but can be challenging to communicate to users [JG09, p. 93].
- Optional fillable fields, which are clearly distinguishable from required fields. This was suggested by Bargas-Avila et al. in a study about designing good web forms [BA+10].

4.4 Limitations

At this point it makes sense to shortly summarize what this tool should not be: a generic ontology editor. The survey shows that ontology editors fulfill other tasks then the presented form generators, and are general toolsets, where the main user group remains ontology experts. As this tool aims to bridge the gap between domain and ontology experts, no generic edit functions are planned, but only the mentioned extension functions, as well as some light editing functions for individuals.

Also, very importantly, this tool is developed as prototype and is therefore no complete final product. Several important aspects will not be implemented or researched and must be addressed before considering an actual release:

- Security, including an account system, protection against injection attacks, etc.

- A real multi-user system with different roles and permissions for different kinds of experts and administrators, and in general, users with shareable ontologies and forms.
- An improved frontend

The main value of this tool lies in continuing the research efforts in ontology-based form generators. This tool aims to combine the functions of multiple research projects into one general tool and advance the research by including features of ontology extension, flexibility, and usability.

4.5 Example Use Case

To test the form generator, an existing ontology has been chosen. This ontology will be used to create forms and demonstrate additional features, showcasing their practical relevance and ensuring correct implementation.

4.5.1 Used Ontology

As example ontology the “Wines & Foods ontology” created by the W3C itself is used [SWM04]. This ontology consists of two separate ontologies that import each other, and therefore form a common knowledge base. It serves as a tutorial ontology for the W3C to explain key OWL concepts. This ontology was chosen for several reasons:

- It contains a substantial number of classes (138) and individuals (206), making it suitable for usability tests involving large amounts of selectable classes and individuals.
  On the other hand, the numbers are also not excessively high, and therefore the ontology remains understandable.
- The ontology is not only a taxonomy, but also contains various axioms, like subclass restrictions. In total there are 889 logical axioms, and thus validation tests are feasible.
- The domain of the ontology is simple and easily understandable, covering information about foods and types of wine. This makes it suitable for examples, unlike domain-specific ontologies that require expert knowledge.

As this ontology will also be specialized through ontology extension, a brief introduction to its domain is given: The ontology covers two main areas: foods and wines. The food classes form a traditional taxonomy, with the class "EdibleThing" divided into subclasses like "Dessert", "Meat", and "Pasta". These subclasses, such as "RedMeat", do not have specific restrictions, making them suitable for simpler use cases. In contrast, the wine classes, such as "DryWine" and "Chardonnay", are defined by restrictions that specify when a concrete instance qualifies as a particular wine type. These restrictions are based on object properties like "hasColor", "hasFlavor", and "madeFromGrape". These classes are ideal for validating instances of wine against more complex restrictions.
Both areas are combined by "Meals", which consist of courses that include wines and foods. All together, the following competency questions, among others, can be answered by this knowledge base:

- Which wine fits best with food X?
- Where does the concrete wine X come from?
- Does wine X contain sugar, and what does it taste like?
- Are there different wines from adjacent regions?

4.5.2 Concrete Use Case and Competency Questions

To demonstrate the capabilities of the evolved form generator, an example use case based on the Wines & Foods ontology is built. For this thesis it is assumed that the ontology, which has a solid foundation of existing drinks and meals, should be used in a restaurant context. Here it should be used to record customer orders in the restaurant, either directly by waiters using mobile devices or afterwards.
Each order includes multiple wines and foods, and also has some meta information like a table number. By building this more concrete application ontology on top of the general Wines & Foods ontology, the restaurant can combine the advantages of both. By the general ontology it can get recommendations for good meals and knowledge about the wines, and with the finer application ontology also the following competency questions can be answered:

- Which is the most popular product of the restaurant?
- How often was food X sold?
- Which foods and drinks are frequently sold together?

Of course, any changes to the basic domain ontology must ensure that all competency questions can still be answered.
This is the basic application extension which is already implemented by the ontology expert from the beginning. After that some other extensions to these competency questions are planned. These are detailed later.

Chapter 5

Design and Implementation

In the following chapter, the developed prototype "OntoFormGenerator" is introduced.

The decision to develop a new form generator from scratch, rather than building upon existing tools from the survey, was driven by several factors. Many of the surveyed tools used older technologies unsuitable for modern web applications, as discussed in chapter 3. Also, these tools were created with specific constraints and research focuses, making an expansion for ontology specialization and other requirements challenging. Furthermore, the source code for some tools was not available. Therefore, a complete own implementation was undertaken.

5.1 Architecture

5.1.1 Overview

OntoFormGenerator is designed as a web tool for online usage, accessible via a standard web browser. The prototype is built within a web architecture powered by the Java server solution Spring Boot. That means the frontend consists of HTML, CSS and JavaScript files which are delivered to the web browser for rendering, and a backend on the server side which handles API requests when the user interacts with the tools. The tool then executes Java-written functions to modify the data model, and updates the information displayed to the user. This architecture was influenced by other form generator tools like OBOP and Facsimile and decided upon early in the development process. A significant advantage of this approach is the use of HTML forms, which include various form fields based on specified data types, automatic web browser validation, and easy form submission using POST requests [WHA24].

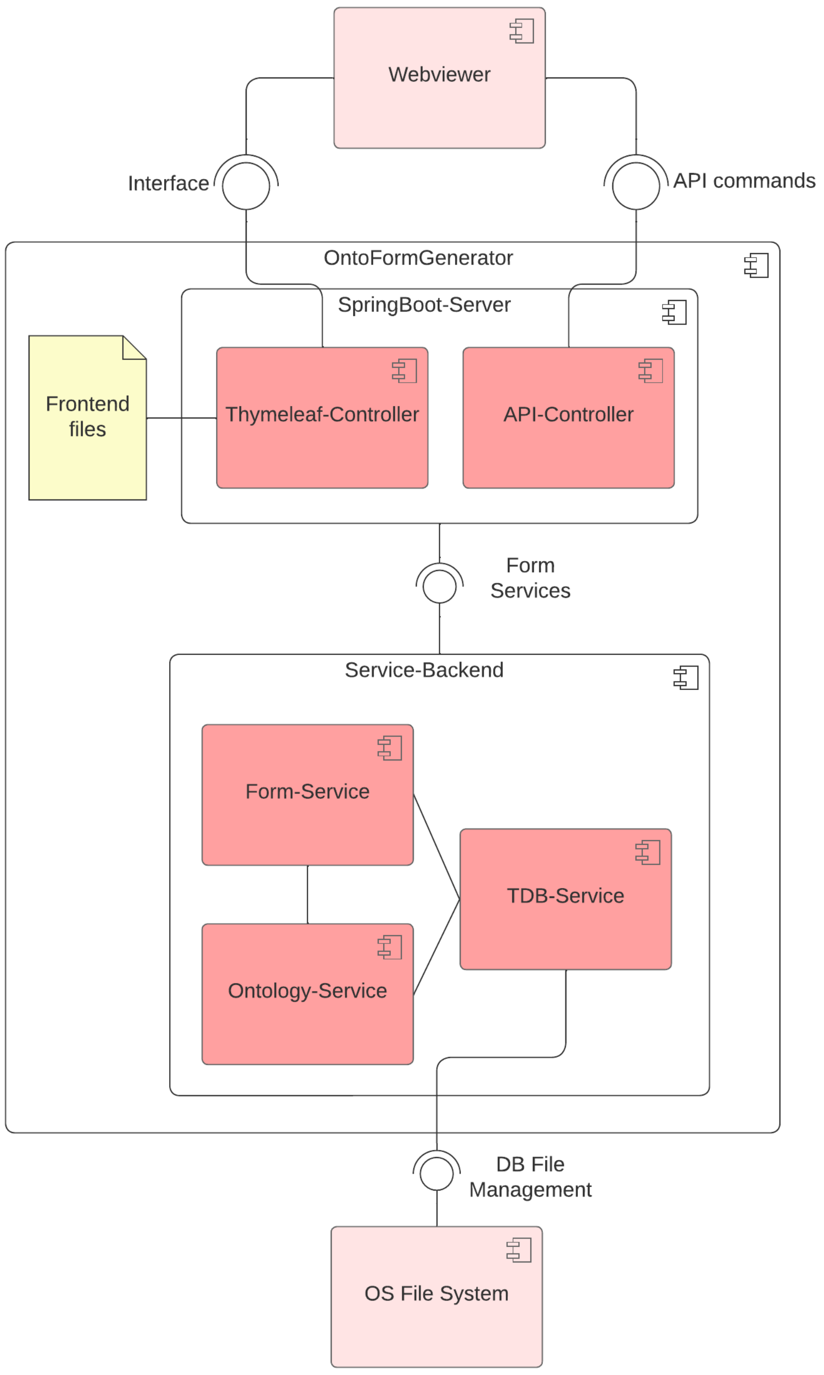


Figure 5.1: Component diagram of OntoFormGenerator

The component diagram in Figure 5.1 shows the main architecture of the system. The system can be broadly divided into two main components: the service backend and a server component delivering HTML pages to the web browser and redirecting API calls to the service backend. These HTML pages then represent the frontend of the software.
The service backend consists of three main services for form and ontology editing, as well as a database service. In total, the backend follows a classical web architecture, with a Spring Boot server providing the interface for connected web clients via a "Thymeleaf controller" and offering a REST API as an "API controller" for executing functions, which are then directly transferred to the service backend. The service backend represents the "Model", the server component including the controller classes is the "Controller" component, and the HTML pages are the "View" [Bal11, p. 197]. The application itself also contains the database functionality, using the file system of the operating system for physical data storage directly, eliminating the need for an external database system. Thus, the persistence layer, typically present in a web architecture [Bal11, p. 197], is ensured by the software itself. More architecture details about each component are given in the next section.

Deployment of the system is straightforward, as it consists of a single deployment unit. The program is packaged as a single executable, which, thanks to the integrated web server in Spring Boot, can be run using a standard Java runtime to make the web server available.

5.1.2 Backend

The decision to use Java as the programming language was made early in the development process, just like the general web architecture, for several reasons. First, Java remains a popular language, consistently ranking high in language popularity indices, such as the TIOBE index, where it was ranked third as of October 2024 [Jan24]. But more importantly, Java is also well-suited for this project due to its simplicity, robustness, and extensive library ecosystem, which facilitates easy development and extension [IBM24]. Additionally, Java has long been suitable for web development, initially with Java EE and today also with other modern web frameworks [Tha22].

One of these is Spring Boot, which simplifies the creation of web server applications in Java, offering features like easy API endpoint definitions via decorators and dependency injection for simpler programming and better architecture [Bro24a][Bro24b]. The deployment of Spring Boot programs is also very easy, as a web server is integrated in the output JAR artifact, so that no installation of Java application servers like Tomcat is needed. Furthermore, Spring Boot is actively developed with a regular release cycle for its components[1], so that it is unlikely that this technology stack will get outdated very soon, and longevity of OntoFormGenerator is ensured.

Another main architectural decision for the backend was the choice of library for editing RDF data. Implementing a custom solution was deemed impractical, especially because there are multiple Open Source APIs and libraries available.
The two candidates were Apache Jena and the reference implementation of the OWL API. Both support OWL2 as of October 2024, are actively developed, and are open-source frameworks. Jena is the bigger library with more functionality, as it includes a SPARQL API and has its own integrated triple store. Also, Jena is better documented, while the OWL API web presence is deprecated, and there is only little documentation directly at the project's GitHub page left.
However, OWL API is simpler to use, and while Jena is a general tool for all kinds of RDF data, OWL API completely focuses on OWL.

The decision fell for Jena as main library because of the integrated triple store and the better documentation. However, the OWL API is also used as a reasoner interface because some reasoners, like HermiT or FaCT++, only support the OWL API [Gli+14][Inf13].

5.1.3 Frontend

The web frontend uses a classic web stack of static HTML, CSS, and JavaScript files, included in the Java backend project and delivered by the Spring Boot server. This approach simplifies the architecture by requiring only one web server for both frontend and API requests. Complex JavaScript frameworks like Svelte, React, or Angular were avoided, as the frontend only requires simple output and input functionality with some validation. The RDF logic is handled in the backend, and the app does not involve account management or complex backend connections, making simple HTML files sufficient.

The HTML files use the Thymeleaf framework, which treats them as templates with placeholders filled by the Spring Boot server via function calls before delivery. This reduces the number of JavaScript API calls needed to fetch information from the backend, improving data economy. However, there are still API calls needed for information that is not necessarily used by the user, ensuring that HTML files do not become too large with many ontology elements. For example, when a form is loaded, the needed individuals for the object properties are directly loaded via Thymeleaf, as the user wants to search the complete knowledge base when he fills out the form, and does not want to wait for server calls to fill in the individual

[1]See: `https://spring.io/blog/category/releases`. Accessed: 11/24/2024.

	Data format	**rel. Size**	**Read**	**Write**
Imported ontologies	Triple data	large	⊗	⊗
Meta information about ontologies	unclear	small	⊗	⊗
Form data	HTML forms/XML	medium	⊗	⊗
Filled form data	Triple data	large	⊗	⊗
Configuration data	Key/Value data	very small	⊗	

Table 5.1: Overview over the data used by OntoFormGenerator

lists. On the other hand, the graph of all classes in an ontology, which might be needed when creating a new property, is only loaded if the user specifically calls this function, as there are many cases where the user does not need this feature. This balance between Thymeleaf loading and dynamic API calls optimizes network bandwidth, user experience, and processing expense on the client, while allowing for future scalability [Hal12, pp. 64 sq.].

The CSS framework Google Material Lite was chosen for its modern, appealing look and ease of use. The Lite variant consists of additional CSS and JS files that can be installed on the server, ensuring data protection by avoiding calls to Google servers [Goo24]. Google Material Lite is open-source under the Apache License, making it suitable for this project.

5.1.4 Data Storage

To determine the necessary data storage, the types of data used in the application were analyzed. An overview of the read and written data objects in OntoFormGenerator is given in Table 5.1, the respective sizes were only roughly estimated for comparison.

The main data to be stored is triple data, including imported ontologies and filled forms, i.e. individuals. Given the expected large volume of concepts and individuals, this data will constitute the bulk of the application's storage requirements. Both must be written and read: The ontology data is written at the import stage, and is read when creating forms, while the new individuals are written when the form filling is completed, and are read later, for example when referencing to them in another form. Apart from that, the forms themselves also need to be saved potentially taking up significant space as a large number of forms may be created. If they are directly saved in their target format, they should be available as HTML files, and therefore in an XML format. Other data types, such as meta information about the imported ontologies and created individuals and configuration variables as key-value data, which are only read by the tool at the program startup, are relatively small.

The main requirement for data storage is thus a triple store capable of handling large amounts of RDF data. Given the application's focus on importing, exporting, and editing RDF-based knowledge bases, storing ontology data directly as triples is more efficient than converting it from a relational or other database format. As Jena is already in use for the backend ontology functionality, it was decided to use its integrated triple store TDB. The main reasons are its seamless integration with Jena, robust features like MVCC transactions, and suitability for production use [The24a]. The transaction feature is particularly interesting for future multi-user scenarios. Also, it was shown that TDB is able to also handle extremely large datasets like the complete Wikidata repository of 2021 [Wor21], which further supports this choice. Additionally, in comparison to other popular triple stores, like MarkLogic, GraphDB and Virtuoso, TDB is completely open-source.

For the remaining data types, it was decided to keep the architecture simple and save all data in TDB, as the only remaining significant data volumes are the forms. For that, a custom ontology representation, named "forms" is used, allowing for easier editing and modeling directly in the triple store. That was inspired by some of the tools from the survey, like OBOP, which also use custom ontologies for saving internal data and form representations. This approach also decouples form representation from display, making future changes to the rendering easier. The backend converts the triple format into the final HTML artifacts for display, ensuring flexibility and maintainability.
The other data types, the meta information and configuration variables, are also saved in this "forms" ontology. The exact design of this ontology is explained in the next section.

5.2 Design

5.2.1 Data Storage

As explained, all information about the forms is stored in a dedicated ontology knowledge base referred to as "forms". The classes and properties are shown in Figure 5.2. Classes are represented as rectangles, object properties as red arrows connecting domain and range classes, and data properties as green arrows linked to their respective datatypes. Cardinalities of the object properties are also indicated.
The central entity in this ontology is the Form class. A form consists of multiple, but at least one, form elements, which are the user inputs. These form elements can

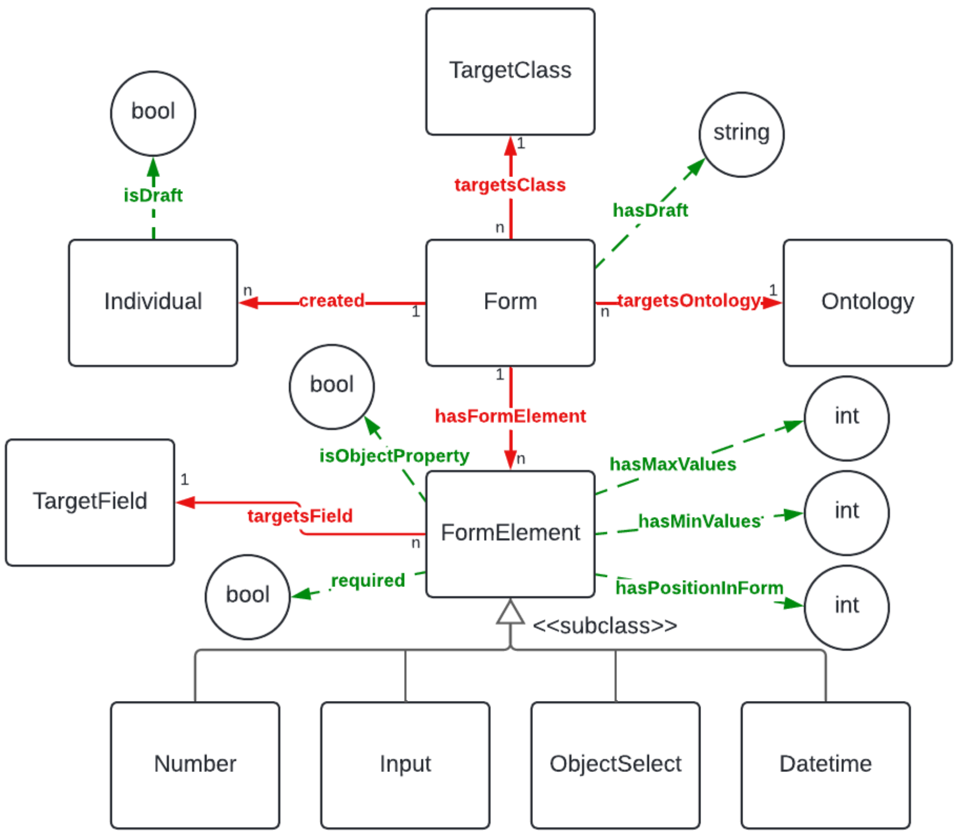


Figure 5.2: Structure of the forms ontology

have various data types, modeled as subclasses of FormElement, such as text input for strings, integer-only input for numbers, or ObjectSelect for object property fields. Also, the FormElement has a reference to a TargetField, which is a property of the domain ontology, and various data properties, like minimum and maximum possible values, the position in the form or a “required” flag, if this field is optional.
Any form is connected to exactly one class, the TargetClass, of the ontology, for which instances are created. Instances created by the form are stored in both the form ontology and the knowledge base of the imported ontology, allowing for later analysis of which individuals were generated by the form generator. Also, the data model distinguishes between completed individuals and drafts, which are initialized but uncompleted forms[2]. Drafts are not included in the knowledge base of the imported ontologies but are marked in the forms ontology with “isDraft”, and their content is saved as a JSON string with “hasDraft”.
The property “targetsOntology” is used in a denormalization sense. Although each TargetClass belongs to exactly one Ontology, making this property theoretically unnecessary, selecting the correct ontology for a given form is often needed in the tool, and so this property is introduced additionally. Also, the transactional nature of TDB ensures consistency even with this potential source for contradictions in the data model, and this relationship between form and ontology remains unchanged once established.

The instances of Individual, TargetClass, TargetField and Ontology are all individuals from other ontology knowledge bases. Initially, the plan was to use concrete individuals from other ontologies via OWL imports. This would have been theoretically possible, although classes and properties are here used as instances, breaking the clear separation of classes and individuals in OWL DL [Bec+04]. But OWL 2 includes the option of Punning, in which the class and individual would be different objects, but could share the same IRI [GW12]. However, in the end that idea was discarded, as this would have caused a very large number of imports in the forms ontology for every imported domain ontology. Instead, the same IRIs as the original ontologies are used without importing them, maintaining the relationship to the original objects as these IRIs are globally unique, as already explained.

The initial forms knowledge base is automatically set up by the application at startup, if it does not already exist.

5.2.2 Server Design

A basic class diagram of the backend is shown in Figure 5.3. The main classes are the controllers, which are the entry points for the API calls and do not yet contain

[2]More details about drafts are given in subsection 5.3.3.

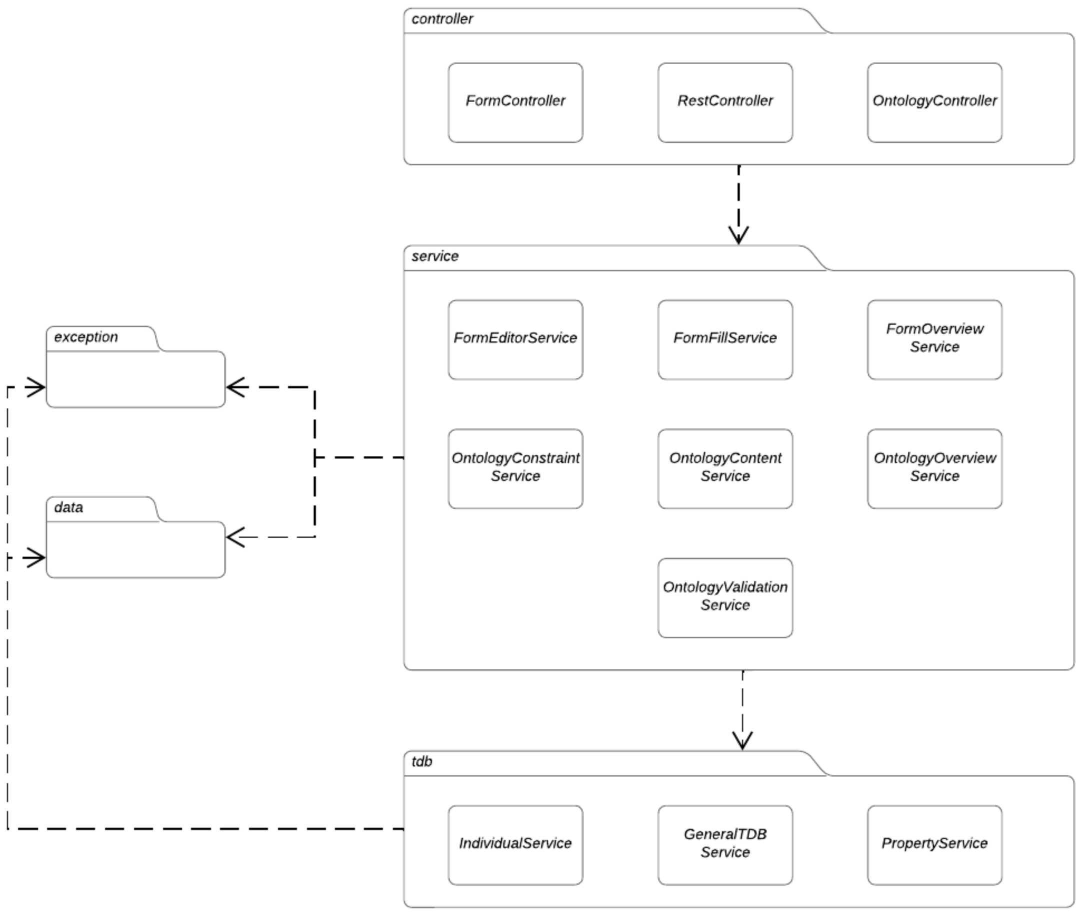


Figure 5.3: Class diagram of OntoFormGenerator

any main program logic, and the services, which contain the application's core logic. These services are again divided into higher-level logic services and low-level TDB services, which directly interact with the TDB database. Therefore, this software is built in a layered architecture, where the controllers use the high-level services, and these use the TDB services for data access, while the particular service and controller classes in one package do not have to use each other and remain self-contained. This architecture is also compliant with a typical Spring Boot architecture [Orn24], allowing for the use of Spring Boot patterns like dependency injection. For instance, the TDB service is injected into high-level services, with object management handled by the Spring Boot framework. This all works by the simple use of Spring Boot decorators, so that the main code remains tidy and does not have any web server lines, but is only focused on the main ontology and form logic.

The server's data model, is simple and consists mainly of data transfer objects, a largely used Software pattern [Ora19], for transferring form and ontology data to the frontend, like Individual, OntologyClass or ValidationResult. The exception package contains specific exception types for validation, demonstrating the advantages of object-oriented programming by extending exceptions for readability: There is an abstract "OntologyValidationException", and from that some subclasses for different validation errors, like "NamingSchemaDifferentException".
The controller and service packages are organized by specific tasks: Functions for getting or editing ontology and form information are divided into separate services for readability and maintainability. For example, the FormFillService handles form filling functions, while the OntologyContentService manages detailed ontology information. Similarly, the RestController handles direct page calls in the frontend, while other controllers manage internal API calls.

The dynamic design of the application is simplified by this class design, as shown in a sequence diagram in Figure 5.4. The user, or rather the web browser, communicates directly with a Controller, which uses high-level services to execute tasks. As not much data is transferred per call, and the runtimes for changes are short, the program only uses simple synchronous function calls, and no complicated messaging architecture or asynchronous elements are used.
TDBDataset objects, representing direct datasets from the database, are created and disposed of for each API call. Dataset handling is transactional, using read and write locks as needed. To prevent early exit issues, such as those caused by exceptions, every dataset access is secured by the "Dispose" pattern. This is a software pattern which hides such resource usages of databases in special language construction, so that in every case the resources are released correctly [CA08, pp. 319 sqq.]. In Java that can be implemented with try-finally-blocks, and special disposable objects implementing the Disposable interface, which was done for the TDBDataset.

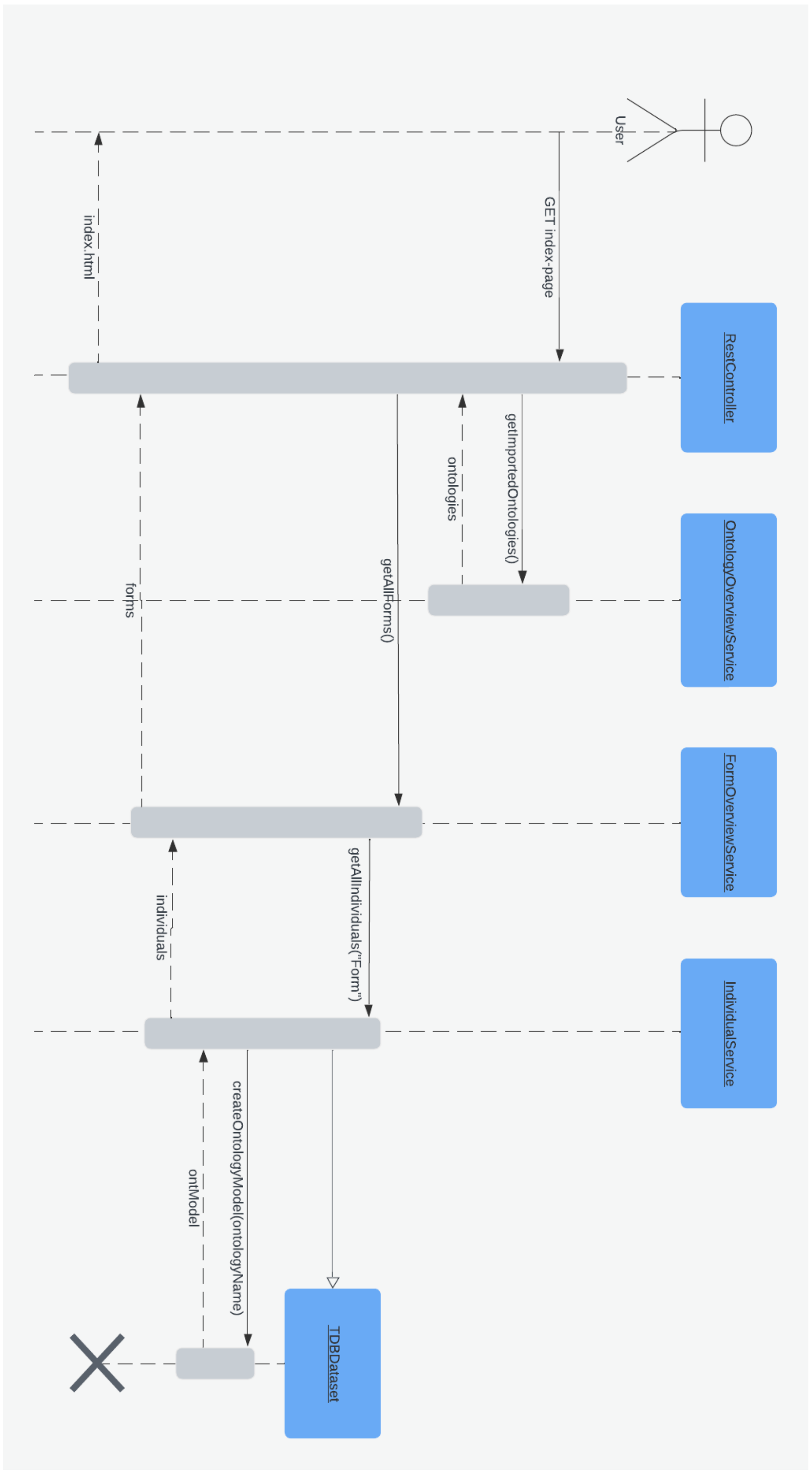


Figure 5.4: Sequence diagram of a basic OntoFormGenerator call

The API design follows established rules [Mas12], directing API paths to resources or controllers instead of functions, using correct naming conventions, and appropriate HTML request methods like GET or POST. An example is given in the following listing:

```
GET /api/ontologies/{ontologyName}/classes/{className}/
    individuals
DELETE /api/ontologies/{ontologyName}/individuals?iri=...
```

Here a GET command retrieving all individuals from a specific class in an ontology is given. Because this concept is hierarchical it was designed as REST path. Also, for identifying ontology resources IRIs are used instead of names or IDs in query parameters, so that the resources can be uniquely identified, and no mapping mechanism is needed, although they make the paths longer.

Logging is handled with the standard Spring Boot Logger implementation, differentiating between critical errors, non-critical warnings, and informational messages.

5.2.3 Used Libraries

In addition to Jena, OWL API, and Google Material, the tool utilizes several other libraries for ready-to-use, reusable solutions to standard problems:

- Selectize: A library for providing drop-down lists with search functions[3]
- Jquery: A standard JavaScript library for easier DOM manipulation, for example when data from an API call has to be integrated[4]
- Graphology: A library for graph objects in JavaScript. Used for creating OWL class graphs[5]
- Sigma: A library for displaying graphs on web pages, using Graphology for the graph objects[6]

[3]`https://selectize.dev/`. Accessed: 11/24/2024.
[4]`https://jquery.com/`. Accessed: 11/24/2024.
[5]`https://graphology.github.io/`. Accessed: 11/24/2024.
[6]`https://www.sigmajs.org/`. Accessed: 11/24/2024.

5.3 Features and Usage of the OntoFormGenerator Tool

This section outlines the main features and general usability of the OntoFormGenerator tool, focusing on its three main views. Only the essential features, as detailed in the previous chapter, are discussed here. Additional features are covered in the subsequent section after a comprehensive overview of the application.

OntoFormGenerator is a fully functional form generator comparable to tools presented in the survey. It includes the most important key features which were also described in the requirements analysis: The import of ontologies, as well as the export of knowledge bases augmented with new individuals, form creation and form filling, which creates new individuals. The tool also implements some additional features, which improve the usability, like optional and required fields and multiple possible values. Field appearance is determined by the property datatype, using standard HTML features for number-only input fields or date selectors.
Noteworthy additional features include individual editing, ontology extension features, and enhanced validation features.

5.3.1 Main Page

A screenshot of the main page is shown in Figure 5.5. The main page consists of three subpages selectable via a menu band, and provides basic functions for importing and exporting ontologies, creating and deleting forms, and editing individuals. From the main page the form editor and filler components can be accessed by selecting a created form. Especially the main page shows the advantages of Material design: With it common design elements from other web pages and mobile apps like action buttons in the right bottom corner, a menu band, and various consistent icons, as well as pre-designed components like boxes and buttons can be easily implemented in the frontend code. For better usability and accessibility also tooltips were implemented for icons. With Material the frontend has a modern look, but also is not overloaded with too many distracting effects.

The main page incorporates design principles used throughout the application. Loading circles are employed to block the application during complex queries, indicating to the user that the application is processing data and is still working. This improves usability and user satisfaction, as supported by User Experience research [BKB00]. In the main page, it is concretely used for loading individuals of a certain class, as, depending on the amount of created individuals, this request can take some time.

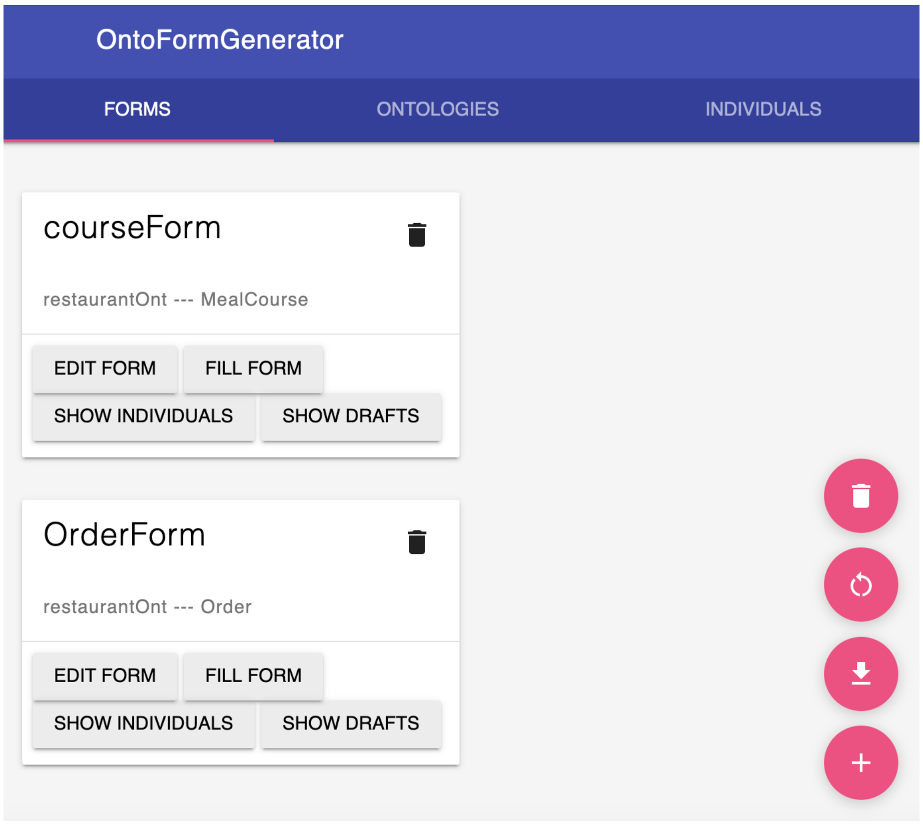


Figure 5.5: View of the OntoFormGenerator index page

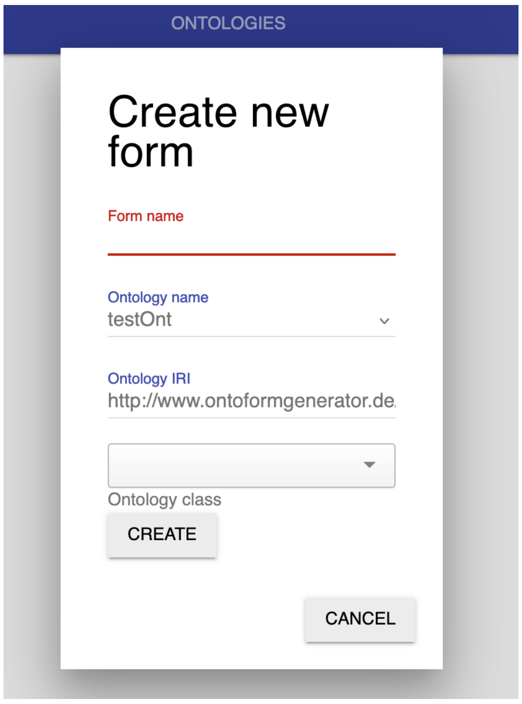


Figure 5.6: View of the form creation dialog

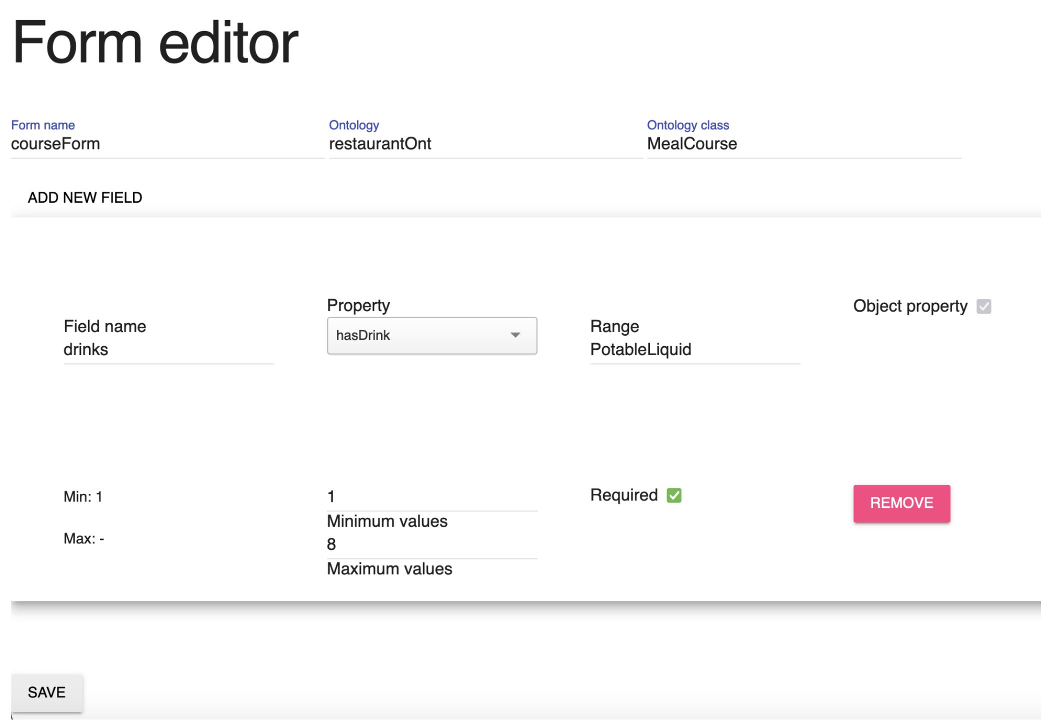
Another important principle in the whole software is the usage of dialogs. One example is shown in Figure 5.6. These are also designed with Material elements and therefore fit to the rest of the application. They are used to streamline the web page by moving contextual parts needed for specific actions to dialog windows. On the main page, dialogs are used to create new forms, including selecting a form name, ontology, and target class from the ontology.

5.3.2 Form Editor

The form editor, shown in Figure 5.7, allows ontology experts to create forms by adding fields. Other than, for instance, in OBOP no sample individual is needed to create a form, but forms can also be created in empty knowledge bases.
In this view the ontology expert can create the first form drafts by adding fields. For better usability, the form editor includes an entity collection feature similar to OWL2MVC that automatically collects suitable properties for fields from the ontology, considering the domain of the form's target class.

For each field the editor must specify the following properties:

- Field name: The designation given to the form field during form filling. An own name is used instead of the property name to abstract form filling from


Form editor

Form name: courseForm
Ontology: restaurantOnt
Ontology class: MealCourse

Field name: drinks
Property: hasDrink
Range: PotableLiquid
Object property ✓

Min: 1
Max: -
Minimum values: 1
Maximum values: 8
Required ✓

Figure 5.7: View of the OntoFormGenerator form editor

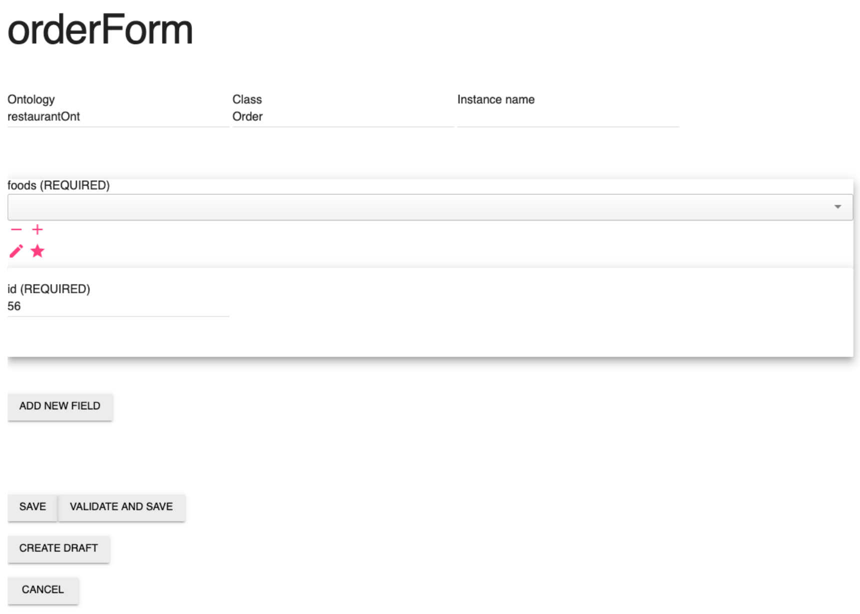
the ontology, simplifying it for the domain expert.

- Property: The property referenced by the field, also determining the datatype. Possible properties are pre-assigned via Thymeleaf and selectable through a searchable drop-down list using Selectize, which only shows reasonable properties based on the domain of the target class. This pre-selection aids validation by allowing only valid properties.
  A search function in the drop-down list helps find properties in large domain ontologies.
  Selecting a property automatically fills range, an object property flag and existing cardinality constraints in the ontology, if available.

- Own Min-/Max-values: Allows the user to specify or force multiple values for a field.

- Required: Indicates that the field is optional.

A form can contain an arbitrary number of fields, which can be modified at any time. Optional and multi-value fields enhance usability and user acceptance, allowing more use cases to be implemented with a single form. This approach avoids creating numerous fields for a single property.


orderForm

Ontology
restaurantOnt

Class
Order

Instance name

foods (REQUIRED)

id (REQUIRED)
56

Figure 5.8: View of the OntoFormGenerator form filler

The searchable drop-down list from Selectize is also used in other parts of the application, like for selecting individuals in the form filler, as it is a very user-friendly way to search for elements in a large list, and therefore improves the usability of the tool.

5.3.3 Form Filler

Figure 5.8 shows the form fill view of OntoFormGenerator. The fields added in the form editor are rendered with their respective data types. Technically, prepared Thymeleaf templates for all possible datatypes are inserted into the core HTML file based on the fields specified in the editor. Also, the "required"-property is shown and forced, before a saving of the individual to the knowledge base is possible.
Object properties are filled by retrieving fitting individuals from the ontology, added to a searchable drop-down list for fast knowledge base search. Other datatype properties are rendered with appropriate inputs, such as text fields, number-only fields, and date selectors. An important edge case can happen here, if range values are not set in the ontology, as the tool cannot know which datatype is expected. Because of the nature of OWL ontologies, this can regularly happen, as a valid ontology does not have to be complete. As solution the tool uses default ranges: For object properties, the default range is owl:Thing, for datatype properties it is xsd:string, as the most general alternatives.

If a field has multiple possible values, adding new values is possible with little "-" or "+" buttons, with browser alerts appearing as error messages, if the maximum amount of values has been reached.
In the standard workflow, a domain expert fills out the form and saves it, adding a reference to the "forms" knowledge base to track user-created individuals. Then a correct individual with the given property values is created and added to the knowledge base of the imported ontology.

Every individual has to be given an instance name before it can be saved, which is transferred to the RDFS label name. The reason is that the domain expert can then later find it easier, for example if it should be used again in another form. Theoretically the user can here also create instances with the same instance name. Thus, it would not make sense to only use the instance name for the IRI. To ensure unique resource identifiers, the instance name is combined with a generated UUID, which is unique, small, and requires no centralized authority [DPL24].

Besides the normal saving mechanism also a draft saving function was implemented. That allows incomplete forms to be saved and edited later from the index page. This is first for a general better usability, as forms do not have to be filled at once. Secondly, this feature is also important for the later described extension features, as this feature allows to interrupt the form filling to create new ontology concepts, and then continue working on the form later.
Technically, drafts are saved as JSON in the forms ontology within the individual with the property "hasDraft", and not added to the domain ontology's knowledge base until completed. An example for a draft JSON is given below:

```
{
    "normalFields": {
        "ontologyName": ["restaurantOnt"],
        "targetClass": ["Order"],
        "instanceName": ["exampleInstance"],
        "hasOrderedDrink": ["CorbansDryWhiteRiesling"],
        "hasSpecialWish": [""]
    },
    "additionalFields": {}
}
```

It can be seen that the general metadata of instance name, ontology name and target class are saved, as well as all already filled fields in list formats for potential multi-value fields. JSON was used as it is a broadly used format with many parsing and writing libraries existing, so that the draft JSONs can easily be generated. It is also highly flexible, allowing for an arbitrary number of fields and values, which is very important for the additional fields introduced in the next section.

5.4 Inclusion of Ontology Extension in the Tool

5.4.1 Extending Forms

The first aspect of the evolution features is about extending forms by the domain expert. As already explained, ontology engineering is a complex field, and such the paradigm for this feature should be to only have minimal changes to the ontology. The focus should be on aiding users in locating existing ontology elements, so that unnecessary changes are avoided. For adding a new field, where a property from the ontology must be selected, inspiration was drawn from the NFDI4Ing terminology service[7]. This service aggregates ontology elements from various engineering ontologies and offers a search engine for finding classes and properties via free text search. For that the search engine utilizes metadata properties of the various classes and properties, including labels, synonyms, descriptions, identifiers and also other own defined annotation properties [Tec24].
In OntoFormGenerator, although the problem is simplified as all terms originate from a single knowledge base, a similar approach is used: A search function which also uses OWL annotation properties. Currently, a basic search function is implemented, which checks if the query text appears, case-insensitively, in the local name, label, and “rdfs:comment” of any property. This implementation has significant potential for improvement, particularly by incorporating additional and custom annotation properties or employing better search methods. The technical implementation is straightforward, involving Jena API calls to retrieve annotation properties and filter properties within the ontology.
Once found, the property is then shown including the label and the “rdfs:comment”, so that for domain experts it is clearly visible, what each property does. Of course, these measures all require the domain ontology to use meaningful values for labels, descriptions and other annotation properties.

5.4.2 Extending the Ontology

Solving the Technical Extension Problem Before any permanent extensions can be made, the main issue of maintaining the original ontology has to be solved. Of course, any exported RDF artifact could coexist with the main domain ontology. The challenge lies with the ontology IRI, which any OWL ontology can have [MPSP12], and should be usable as a download link for the ontology artifact. Any ontology tool would face issues if the downloaded artifact and the exported one from the tool differed.

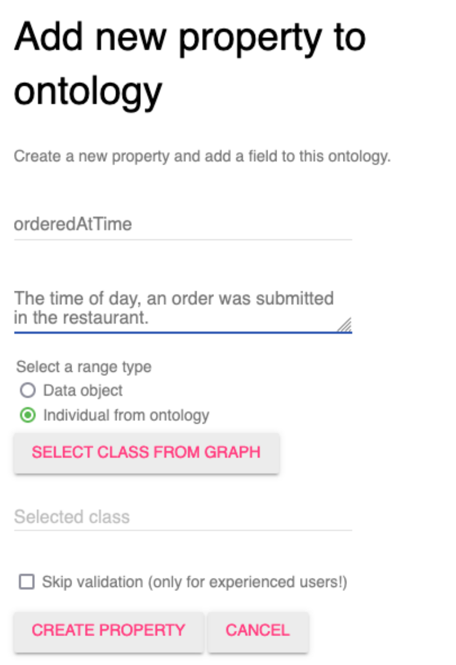
[7]The service is available at `https://terminology.nfdi4ing.de/ts/about` (Accessed: 10/18/2024)

One approach is to use the version IRI of the ontology. This is a standard OWL feature, which allows to give any ontology a unique version with an own IRI, from which this specific version could be downloaded. [MPSP12]. Thus, the technical problem would be solved.
However, semantically, this approach is less meaningful as the exported knowledge base should be viewed as a parallel, more specified ontology for a specific application rather than an improved version of the original ontology. Therefore, it was decided to instead create a new ontology with its own IRI pointing to the web domain of the OntoFormGenerator. This new ontology copies all classes and properties from the original ontology, including their IRIs, and collects all imports from the original ontology. New concepts are then added to this ontology with new IRIs, distinguishing them from the original concepts.

Adding New Individuals The feature for adding new individuals to the knowledge base while filling a form is straightforward: Each object property field in the form filler has a button to add a new individual of exactly this domain class. By this it is also ensured that the new individual is not just an empty one, but it is directly connected to the existing ontology concepts. When using the add button, the user has multiple options to add a new individual: they can call an existing form with the domain as the target class and save the current one as a draft, or they can add an empty individual with only its domain declaration. This is especially useful if the user wants to add a new concept to the ontology, but does not yet know all the properties of this concept. So, the user can also work if they only have incomplete information, enhancing the tool's usability and enabling the creation of imperfect knowledge bases that can be finalized later. When the information is available, the tool includes an individual editor where users can add property values to the individual, accessible from the homepage or the form filler. Only possible properties regarding the domain class are displayed, and only basic values are allowed, making the feature usable by advanced domain experts or in collaboration with ontology experts.

One general concept, which is also seen when creating new classes and properties, is that the tool handles as many tasks as possible in the background, and only lets the user manipulate some things. For the individuals the user only has to give a name, which is used as label name, and also for the IRI, which is generated by the tool. Also, the class value is set just like fitting annotation properties.

Adding New Properties The entry point for this feature is when creating an additional field. As already explained, the tool assists in finding existing fitting properties in the ontology. But if no concept can be found, a new property can be added, which is also done via a dialog window. A screenshot of the dialog is shown in Figure 5.9.


Add new property to ontology

Create a new property and add a field to this ontology.

orderedAtTime

The time of day, an order was submitted in the restaurant.

Select a range type
Data object
[✓ Individual from ontology]

Figure 5.9: Screenshot of the Create-Property-Dialog

The window again only has few relevant fields, like the individual creation dialog. The dialog requires the user to provide a name, domain, range, and description. The range is specified by selecting from given datatypes for a datatype property or by selecting a class for an object property. Everything else, including the IRI, the correct domain, for which also the target class of the form is used, and the correct range IRI by the user selection is done by the tool.
A discussed point in the workflow was the communication of new concepts to the domain expert, so that they can then change the concept to a more fitting one, if needed. For this, it was decided to add an annotation property to the new concepts: “isUserDefined”. These are not shown in the form filler, but are automatically added to the new property or class. On this basis, the ontology expert can later search for all user-defined concepts and properties and decide how to handle them.

Selecting and Adding New Classes The process of selecting and adding new classes for property ranges is crucial due to the potentially large number of classes in an ontology. Therefore, the tool has to support the user in finding the correct class, even more than for properties, as here the properties, shown to the user, are filtered by domain class. Because of this a graphical solution was chosen over a simple search function or drop-down list. This solution involves displaying a graph of all classes in the ontology, accessible from the range value selection in the create-property dialog. In this directed graph the classes are represented as nodes, while subclass connections between classes are shown as edges, pointing from the subclass to the superclass, similar to the subclass visualization in UML class diagrams. For generating the graph a simple SPARQL query is used to get all class-subclass relations of the ontology:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?class ?subclass WHERE {
    ?subclass rdfs:subClassOf ?class .
}
```

This query produces an edge list used to construct the graph. A class that does not have any superclass would be excluded from the graph, as no “rdfs:subClassOf” property would exist. But for OWL this is not relevant, as every class must at least have “owl:Thing” as superclass [SWM04].
The reason that this data structure is built up as graph, and not as tree, is that in OWL circular subclass relationships and multiple superclasses are possible [Bec+04], which can not be covered with a tree.

The graph is displayed in a separate dialog using the Graphology and Sigma libraries, which provide usability features such as zoom, drag functions, and click events. Users can select a class by left-clicking and add a new subclass by right-clicking. Adding classes is simplified for domain experts by limiting the use of OWL class constructors and constraints. For instance, disjointness definitions are not selectable, and only one superclass and the class name can be chosen. A screenshot of this graph is shown in Figure 5.10. The node locations are not assigned randomly, but instead a node layout algorithm was used to calculate fitting positions. For this subclass graph ForceAtlas2 was chosen. This is a continuous algorithm suitable for bigger networks which uses an “energy model”, including attraction and repulsion factors [Jac+14]. By this, classes with the same superclass surround in dense centers, while other subclass connections stretch across the graph. This can be seen in the figure with “NonSweetFruit” as subclass without successor being close to “EdibleThing”, while “Dessert” as a more central node is further away. So, larger subclass structures get more noticeable. The layout algorithm is provided by the Graphology library, and thus does not need to implemented here itself.

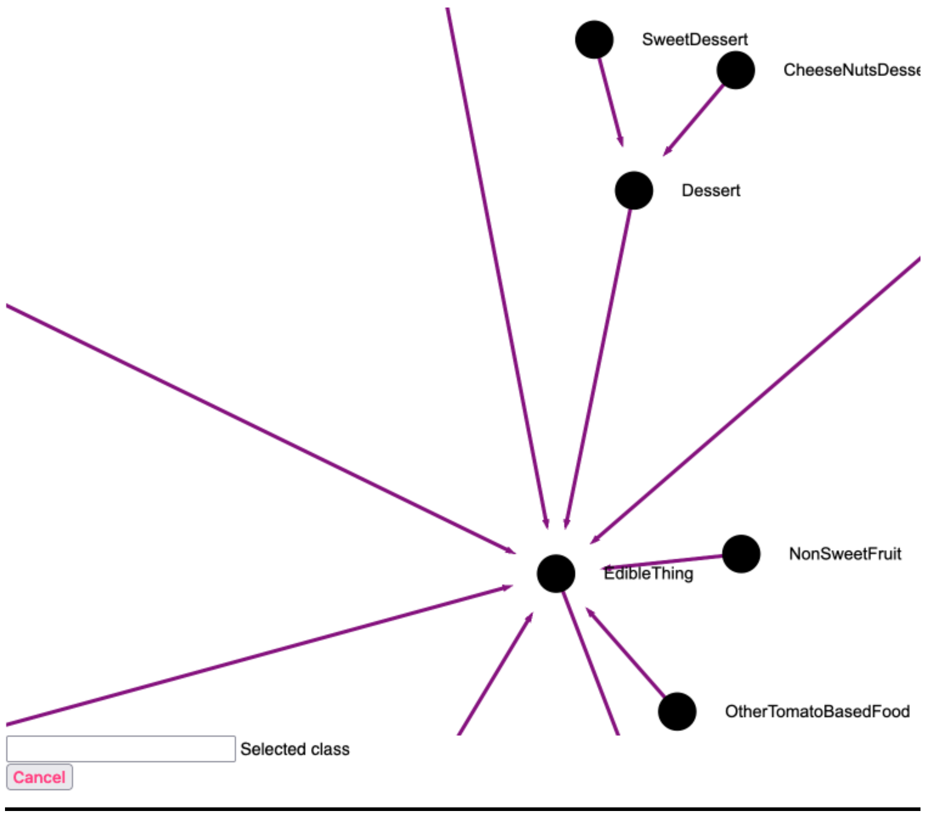
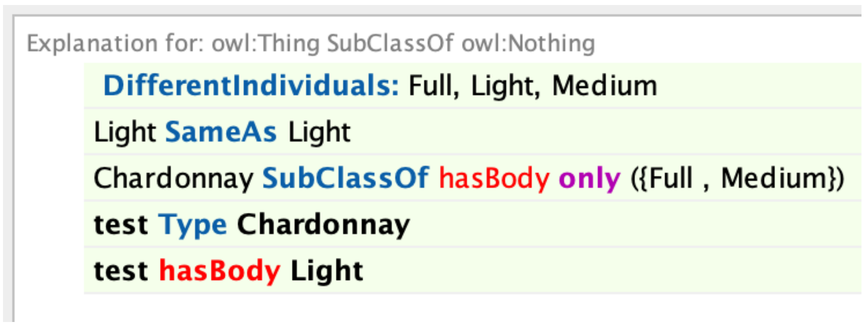


Figure 5.10: Screenshot of the subclass graph



```
Explanation for: owl:Thing SubClassOf owl:Nothing
    DifferentIndividuals: Full, Light, Medium
    Light SameAs Light
    Chardonnay SubClassOf hasBody only ({Full , Medium})
    test Type Chardonnay
    test hasBody Light
```

Figure 5.11: Example of HermiT output on an inconsistent ontology in Protégé

5.5 Validation of the Knowledge Base

5.5.1 Concept for a Validation Framework

The most obvious way for automatic validation of knowledge bases is the usage of reasoners, which check the consistency of the knowledge base and identify inconsistencies. The usage of reasoners has multiple advantages and disadvantages: First, they fit into the required validation process, as they are automatic and can check all logical rules of the knowledge base. Then reasoners can in general be easily integrated into any application. For example, the HermiT reasoner can be used either as command line tool, which is called from the application, or can directly be integrated into any Java application using an API [Mot+17]. They provide specific violated axioms, aiding in identifying the inconsistency.
But on the other hand, there are also some disadvantages: The mentioned result can be very unreadable, especially for non-ontology experts. An example of HermiT output is illustrated in Figure 5.11. Here, only one "allValuesFrom"-restriction has been hurt by a property instance. Still, five axioms are shown. For ontology experts this output is sufficient, but non-experts would have problems understanding the reason for this error.
Other problems with OWL reasoners have already been mentioned in chapter 2: For OWL Full checking consistency can be undecidable, and in general the runtime can be very long depending on the ontology and the amount of axioms, which can be problematic for online tools requiring quick updates.

Because of all these drawbacks, but also because logical reasoners cannot address various other formal problems, like naming or validation of the general ontology design, also some other options for validation were explored. However, it soon became clear that there are not many other options for automatically validating ontologies. One of the most known methods for validating ontology design is OntoClean. But this method is not suitable for automatic validation, as it needs human input to assign certain metaproperties to classes, so that it can then be evaluated if those terms are implemented in the ontology as intended [GW09]. Because of the needed manual work and the dependence on ontology experts, this method will not be considered further.

Another method involves Ontology Design Patterns (ODPs). These are "modeling solution[s] to solve a recurrent ontology design problem"[GP09], and are therefore comparable to design patterns in Software Engineering. Especially the subtype of Content ODPs is interesting, as these are little ontologies which can be used to solve common design problems [BGP09]. So, the idea is to explicitly only allow modifications only with the usage of established Content ODPs.
However, this idea was also dropped because of multiple reasons: Even though a rather big list of Content ODPs exist[8], they all describe very special use cases, and no ODP is fitting for the general addition of concepts as planned here. Most of these patterns are without documentation or literature, and therefore have, supported by the wiki type of website, no quality control. A selection of certified ODPs was planned, but the whole project of collecting and using ODPs is abandoned. This is already seen by the fact that the website is currently [9] unavailable. Because of these reasons it is unsure if a usage of ODPs improves the quality of the specialized ontologies, and therefore they are also not further considered here.

Another idea was inspired by the work of Vcelak et al. that, as depicted in the survey, also checked some constraints like cardinality restrictions [Vce+17] without using a reasoner afterwards, but by integrating them directly into the form editor and filler. By this, the application of constraints is moved way earlier, and those can seamlessly be integrated into the frontend. The problem is that for every type of constraint an own, individual solution has to be found.

Another promising approach is the identification and validation of "ontology pitfalls" as proposed by Poveda-Villalón et al. [PV+14]. This method involves checking for 40 typical errors in ontology development, once the ontology has been changed, including naming conventions and design issues. An implementation of checking against selected pitfalls is therefore a good idea to further improve the modified ontologies.
Combined with the usage of reasoners for securing logical consistency and moving some chosen restrictions to the fill and edit processes, a robust framework to maintain a good ontology quality is provided.

[8]See: `https://web.archive.org/web/20240421061442/http://ontologydesignpatterns.org/wiki/Submissions:ContentOPs`. Accessed: 11/24/2024.
[9]Accessed: 10/03/2024

5.5.2 Implementation of the Validation Framework

The tool implements these validation methods through an abstract class used in the OntologyValidationService, ensuring interchangeability of reasoner algorithms. As concrete reasoner HermiT is used, as it is compatible with the OWL API and could be easily integrated into the program. The reasoner is triggered at different points in the program when the knowledge base of the ontology is edited. These are the following:

- Filling forms and editing individuals, as this can break axioms regarding properties of certain classes
- Adding classes and properties to the ontology

For each validation, first, the Jena ontology is converted to the OWL API format for validation. Then the reasoner is activated, and it either returns conflicting axioms or none, if the ontology is consistent. At this point the transaction functionalities of Jena are used to revert changes if inconsistencies are found. Thus, rollbacks are easily possible, as the changes do not have to be deleted manually, which is a huge advantage of Jena and TDB in comparison to other ontology APIs without these transaction functions.

As already discussed, in case of an inconsistent ontology the returned triples are those that cause the inconsistency. To improve at least the basic readability of the returned axioms, they are transformed into Manchester syntax [Hor12].
The following example, where a new individual cannot be inserted because one property violates an “allValuesFrom” constraint, illustrates this. Here, four output axioms are shown in the standard mode:

```
ClassAssertion(<http://www.w3.org/TR/2003/PR-owl-guide-20031209/
    wine#Chardonnay> <http://www.w3.org/TR/2003/PR-owl-guide
    -20031209/wine#newChardonnay>)

DifferentIndividuals(<http://www.w3.org/TR/2003/PR-owl-guide
    -20031209/wine#Full> <http://www.w3.org/TR/2003/PR-owl-guide
    -20031209/wine#Light> <http://www.w3.org/TR/2003/PR-owl-guide
    -20031209/wine#Medium> )

ObjectPropertyAssertion(<http://www.w3.org/TR/2003/PR-owl-guide
    -20031209/wine#hasBody> <http://www.w3.org/TR/2003/PR-owl-
    guide-20031209/wine#newChardonnay> <http://www.w3.org/TR/2003/
    PR-owl-guide-20031209/wine#Light>)
```

```
SubClassOf(<http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#
    Chardonnay> ObjectAllValuesFrom(<http://www.w3.org/TR/2003/PR-
    owl-guide-20031209/wine#hasBody> ObjectOneOf(<http://www.w3.
    org/TR/2003/PR-owl-guide-20031209/wine#Full> <http://www.w3.
    org/TR/2003/PR-owl-guide-20031209/wine#Medium>)))
```

Next, the four axioms in their Manchester syntax are displayed:

```
newChardonnay Type Chardonnay

DifferentIndividuals: Full, Light, Medium

newChardonnay hasBody Light

Chardonnay SubClassOf hasBody only ({Full , Medium})
```

These axioms are way more readable, because the URI paths are removed, the order of the words are changed and brackets are removed.

As discussed, some restrictions are already included in the form editor and filler. For this prototype the following constraints were implemented:

- Min- and max-values of properties depending on the domain object. If those values are given they will be displayed already in the form editor, and own min- and max-values can only be given by the ontology expert in this frame.
- The "allValuesFrom" restriction is implemented in a way that in the form filler only allowed values are shown for each property. So, after the selection depending on domain and property, also another filtering takes place for these constraints.

These both implementations should be seen as examples as there are way more constraints possible in OWL, as already described, for example with "someValuesFrom".

Additionally, the checking of the 40 pitfalls of ontology design was implemented as another validation measure. For this these 40 pitfalls were analyzed and only those were implemented, which can be checked automatically and which are actually relevant for this tool. The exact analysis of the pitfalls can be found in appendix A1. In summary, only seven out of 32 automatically checkable pitfalls are worth implementing, as the rest is either already ensured by the tool with its ontology design rules, or are too complicated and not really worth the implementation costs and thus remain open. The implemented pitfalls include checks for existing concepts, consistent naming styles, and avoiding miscellaneous classes.

Chapter 6

Evaluation

In this chapter, the implemented tool and the proposed new features are evaluated. The objective is to determine whether the implementation meets its intended purpose by verifying it against the requirements introduced in chapter 4. Additionally, the utility of the features related to ontology expansion, validation, and usability is assessed to ascertain their effectiveness. This evaluation is conducted by implementing the restaurant use case and demonstrating the functionality of the features within this context to ensure the complete use case can be realized.

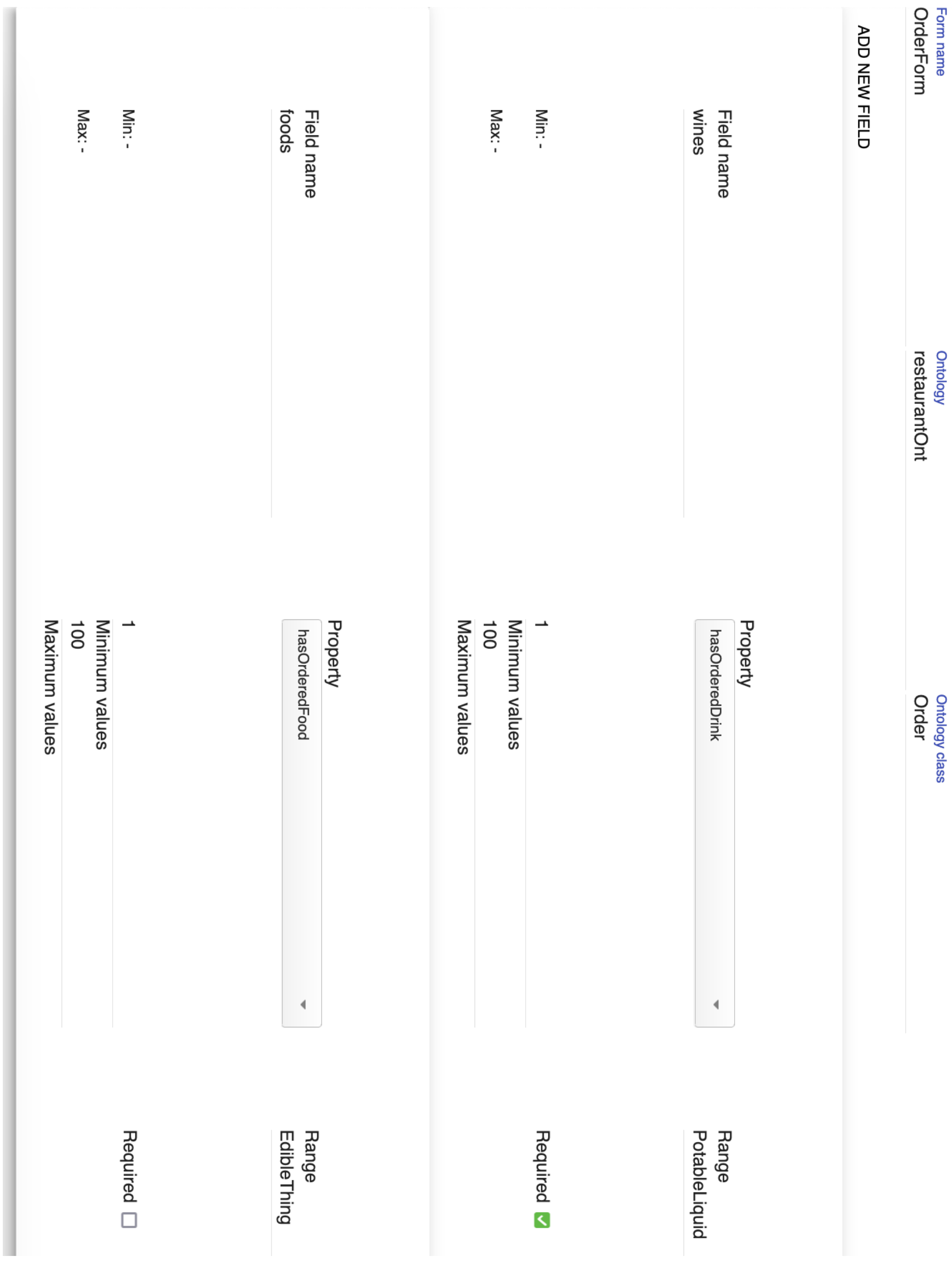
6.1 Implementation of the Food & Wines Use Case

This section details the implementation of the restaurant use case, as described in subsection 4.5.2.

The starting point is a modified version of the Food & Wines ontology[1]. The basic workflow is straightforward. After importing this ontology, a form can be added: In this case, individuals for the Order class should be created: Each order should include at least one Wine and optionally Food/EdibleThing, with the possibility of multiple values for both properties. Additionally, each order must be assigned an integer ID for accounting purposes. This simple form can be easily created using the implemented multi-value and optional field features. The result in the form editor can be seen in Figure 6.1.

A more interesting scenario involves the domain expert further developing this form. Suppose the restaurant using this ontology system wishes to gather new information. For instance, some customers have special requests regarding their orders, such as alternative side dishes. For that a general string property named "hasSpecialWish" is already included in the ontology for every possible special wish in any order, so that they are collected in one property facilitating later research and the creation of

[1]The ontology definition in XML format is given in appendix A2.



Form name
OrderForm

Ontology
restaurantOnt

Ontology class
Order

ADD NEW FIELD

Field name
wines

Property
hasOrderedDrink

Range
PotableLiquid

Min: -

1
Minimum values

Required ✓

Max: -

100
Maximum values

Field name
foods

Property
hasOrderedFood

Range
EdibleThing

Min: -

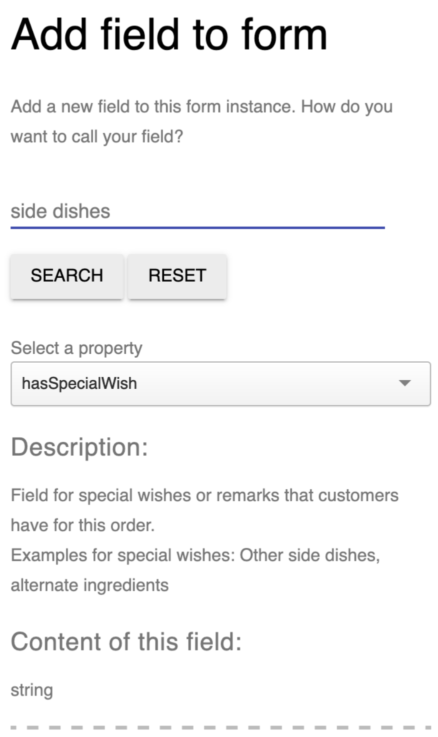
1
Minimum values

Required ☐

Max: -

100
Maximum values

Figure 6.1: The restaurant use case in the form editor


Add field to form

Add a new field to this form instance. How do you want to call your field?

side dishes

Select a property

hasSpecialWish

Description:

Field for special wishes or remarks that customers have for this order.
Examples for special wishes: Other side dishes, alternate ingredients

Content of this field:

string

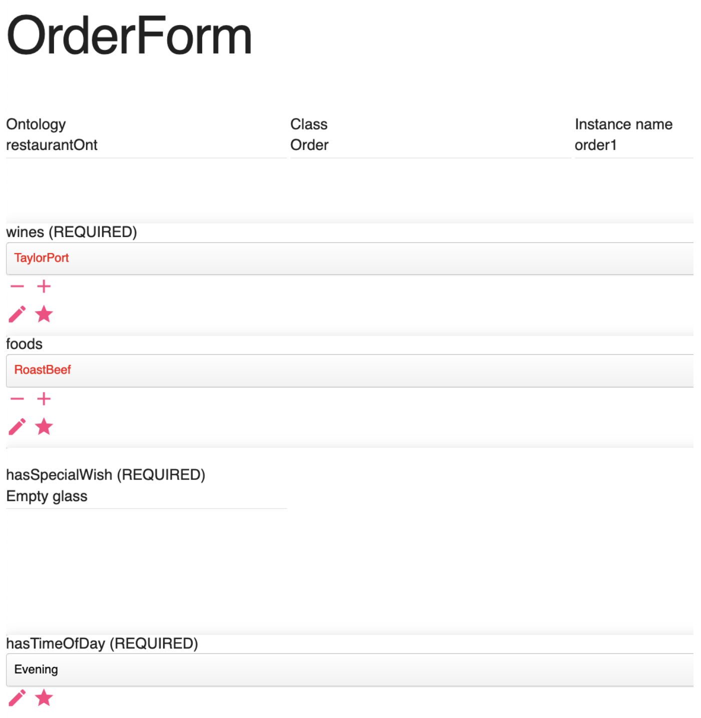
Figure 6.2: The property search function for hasSpecialWish

more specialized properties for particular cases. This property can be spontaneously added to the form. In this case, the property was properly created by the ontology expert with a detailed description:

```
Field for special wishes or remarks that customers have
    for this order.
Examples for special wishes: Other side dishes,
    alternate ingredients
```

This allows the user to easily search for the property, even if they only enter "side dish", and be presented with the correct property. This can be seen in Figure 6.2. Of course, for other cases the description has to be exact enough to capture all possible query inputs by any user.

Subsequently, the restaurant may wish to analyze the times of day when orders are placed to determine which foods are popular at different times. For this case a complete new object property with "times of day" as range, a class that does not yet exist, has to be created, which can be accomplished with OntoFormGenerator. For that a domain expert must complete the following steps:


OrderForm

Ontology	Class	Instance name
restaurantOnt	Order	order1

wines (REQUIRED)
TaylorPort

foods
RoastBeef

hasSpecialWish (REQUIRED)
Empty glass

hasTimeOfDay (REQUIRED)
Evening

Figure 6.3: The complete order form including additional fields

1. Access the Order form and attempt to add a new field.
2. Discover that the desired property does not exist yet and create a new object property.
3. Create a new class with and add the new object property with the new class as the range.
4. Add the new field to the form.
5. Create instances of the new class using the "AddIndividual" functionality, such as "noon" and "evening".

At this stage, the tool's validation mechanisms assist the user in correctly creating the entities: For example, the naming of the property and class is checked if it fits to the other entities for consistency with other entities, and every change in the knowledge base is validated by the reasoner. In this case the reasoner validation is not critical, as Order and the other used classes do not have any restrictions, but for other classes in the ontology, particularly the wine classes, this is crucial.

The complete form including also the two additional fields can be seen in Figure 6.3. Once sufficient forms have been filled, the knowledge base can be exported as an RDF

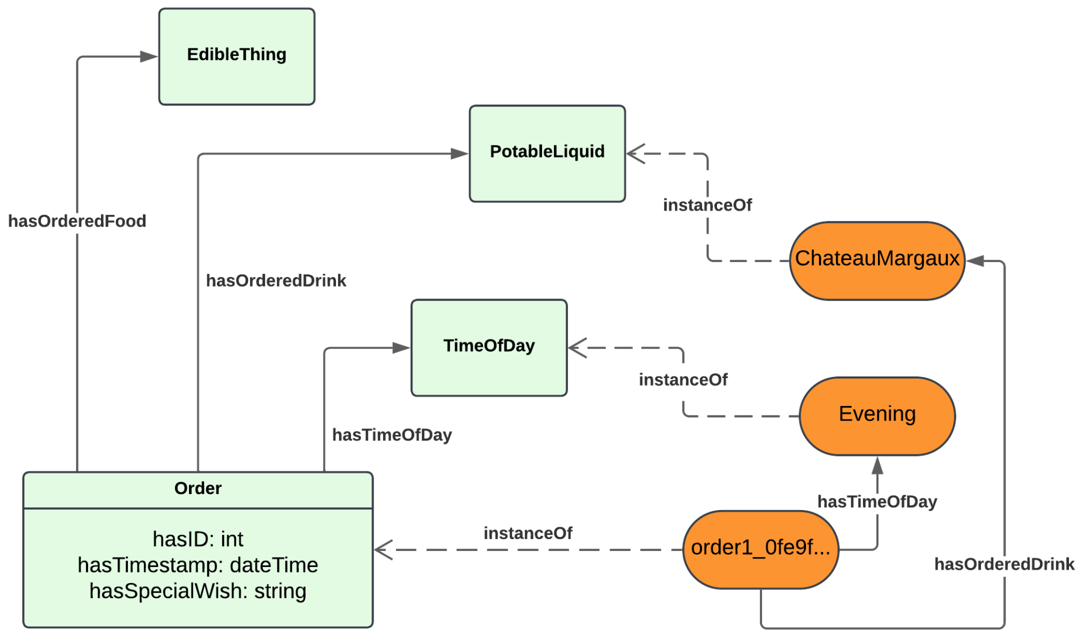


Figure 6.4: The knowledge graph of the restaurant use case after filling a form

file at any time. This now includes the new “TimeOfDay” class, the new “hasTimeOfDay” property, and the new individuals created by filling the form. The resulting knowledge graph of one filled form is shown in Figure 6.4.

It is important to note that, as stated in the requirements, the form itself is not part of the exported knowledge graph, only the filled form instances with the given properties. This allows the restaurant to analyze the data easily, for example, using SPARQL queries. By ensuring that concepts and individuals can only be added, the finer application ontology remains backward compatible with the original ontology, allowing all competency questions, including new ones, as well as the original ones from the Food & Wines ontology, to be answered. For instance, the mentioned competency question “Where does the concrete wine X come from?” can still be answered with the following SPARQL query:

```
SELECT ?region WHERE {
    X rdf:type wine:Wine.
    X wine:locatedIn ?region.
}
```

Even after specializing the ontology, the original concepts and individuals remain available, enabling the original competency questions to be answered. Additionally, the new concepts allow the restaurant to answer new questions, such as "Which orders were made in the evening?":

```
SELECT ?order WHERE {
    ?order wine:hasTimeOfDay wine:Evening.
}
```

One can even go further, and ask for statistics of which wines are ordered how often in the evenings, thereby connecting the two parts of the ontology:

```
SELECT ?wine (count(?wine) as ?count) WHERE {
    ?order wine:hasTimeOfDay wine:Evening.
    ?order wine:hasOrderedDrink ?wine.
}
GROUP BY ?wine
```

As now both queries can be answered, along with the other mentioned competency questions, the tool has successfully implemented the use case.

Moreover, for the ontology expert, the created class and property can be easily identified by the annotation property "isUserDefined", facilitating later refinement of the ontology and modification of user-generated properties.

6.2 Assessment of OntoFormGenerator

This section evaluates the OntoFormGenerator tool based on the findings from the use case implementation and the general descriptions of the features in chapter 5.

6.2.1 Validation of the Basic Requirements

The basic functional requirements have been successfully implemented and are operating as expected, as demonstrated by the use case example. The ontology import functionality is working, creating a new ontology on top of the original one with its own IRI, thereby preserving the original ontology and making the concepts available, including imports. So, the difficulty of not changing the original ontology is already solved at the import step. The advantages and disadvantages of this method compared to using version IRIs have been previously discussed. The export functionality for the new knowledge graph is also operational, with new properties and classes marked with annotation properties.

Next, the form creation and form filling functions are working as specified in the requirements analysis. A key focus was on treating filled forms as individuals of the knowledge base rather than separate entities. This approach ensures that completed individuals are not bound to the forms, allowing forms to be changed or deleted without data loss after already filling it out. Only if drafts are existing, and thus the individuals are not yet formed in the knowledge base, editing of the respective forms is blocked. This could pose problems in a multi-user environment, and a better solution is needed, but it is not relevant for the current implementation. These features make the basic workflow in Figure 4.1 ready for use.

A recurring concept in the tool is the use of implicit restrictions of OWL ontologies to reduce complexity without allowing users to change or circumvent these restrictions. This approach limits user freedom but also reduces the potential for errors. Examples include showing only fitting properties per domain class, simplifying the class hierarchy to a simple graph, allowing selection of concepts only from the current ontology, and automatic generation of IRIs. This approach was intentionally chosen to meet the basic requirement of usability for the two primary user groups, thus also differentiating from generic ontology editors.

In general the implementation of the tool showed that programming a form generator for ontologies sounds simple, but is in fact a more complex task, especially if all rules of OWL and RDFS should be followed. This includes the handling of imports, the creation of actually unique IRIs, and the validation of the knowledge base. These are all topics that are not often addressed in the literature, but are crucial for a working tool, where the resulting knowledge bases, and especially the application ontologies should be consistent and usable for further analysis. This is therefore an important finding of this thesis.

The current programmatic solution for the draft feature, implemented as JSON objects, remains debatable: JSON is fast and flexible but also error-prone and lacks inherent validation mechanisms: For instance, as shown in the example draft of subsection 5.3.3, everything could be entered as values for the fields, including also a wrong number of values. On the other hand, with the later discussed additional form fields, this flexible JSON structure could also be useful.
An alternative could be a draft ontology where drafts are stored as individuals and later transferred to the main ontology.

The non-functional requirements are also fulfilled. Fault tolerance is addressed with various exception handling mechanisms. Faulty entities are hidden, allowing other components to function, and errors are tracked via backend logging for later analysis. Additionally, the web interface allows downloading the current forms ontology structure for analysis with an ontology editor.
Documentation is provided via Javadoc, which can be exported to HTML and is also available in the source code. With that, every important interface function including

the parameters and return values is documented.
Also, various usability features, like non-required fields, multi-value fields and especially the mentioned entity collection feature, using correct OWL reasoning provided by Jena, are implemented.

In summary, the tool operates as expected, fulfilling the basic functional and non-functional requirements.

6.2.2 Evaluation of Ontology Expansion and Usability Features

While the basic use case is implemented and usable by domain and ontology experts, the more interesting aspect is the implementation of ontology expansion and validation features. Various features to help specialize a given domain ontology into a more specific application ontology were proposed and implemented, thus, as shown, the competency questions are still all answerable. Conceptually, the features work as expected, but the tool's usability by domain experts remains in question due to its technical nature. Several main points are discussed for a more thoughtful evaluation:

Adding Temporary Form Fields

As already discussed, the feature to add temporary form fields is the main entry point for other features. The usefulness of making them only temporary is debatable, especially in a multi-user environment where permanent alterations for specific users might be more practical.

The search function for property selection, enhanced by annotation properties and filters, is a good solution for finding suitable properties. Advantages are that it can be made more precise by adding other annotation properties and other filters. Also, just detailed descriptions of properties, like in this use case, can be very helpful, highlighting the importance of the ontology foundation and the need for ontology experts to have basic knowledge of knowledge concepts or collaborate with domain experts.
On the other hand, not only the knowledge base itself is important, but also the implementation of the search logic itself. The current search logic implementation is sufficient for simple ontologies like Food & Wines to find the wished concepts but needs refinement for larger ontologies. Inspiration can be drawn from existing services and information retrieval concepts, treating properties and classes with their annotation properties as document bases for evaluation with search indexes and probabilistic methods [MRS08].

Another idea might be the inclusion of a ranking feature of properties like in PRiSMHA based on user studies. Thus, more important properties can be favored in the search function, and are then proposed first [Goy+20]. The big disadvantage in this paper is that this feature is very domain-specific, and not implemented automatically, as the user study and the initial ranking has to be done manually. Still, this might be an interesting idea for further developing the form generator.

Another point is that this feature implies that the domain expert at least has some basic knowledge of ontology concepts, so that they know what the search function is for, and what even is searched for, and why, for instance, for every property a range type is given. While in this case this is still easy to understand, this problem will get worse with the following evolution features.

Adding New Concepts

The ability to add new concepts to the ontology is a key feature for specializing a domain ontology into a more specific application ontology. However, domain experts must understand the implications of their actions, although they are not ontology experts and may not have a deep understanding of OWL.
The dialog recommends to only create a new property if no fitting existing property via the search function is found. Even if a new concept is needed, the user could still make mistakes, by using a false range type or giving an ambiguous property name or description. The later detailed validation mechanisms and flagging newly created concepts as user-defined help mitigate those errors, although the problem of recreating already existing concepts remains. A possible solution could be similarity score when creating a new concept, based on names and descriptions, to prevent the creation of potentially redundant concepts, but this was not implemented in this tool.

A general challenge is the communication with the user about ontology changes, especially when explaining complicated OWL specifics. An example is the selection of the property range: While name and description should be easy to understand, the range type is more difficult, as it may not be obvious, what the difference between using a data type or an object property is. This is in particular difficult in special cases, like enumerations, which would be rather solved as object property with only a handful of individuals.
In the current implementation that problem might seem trivial because of the easy forms. But this tool was developed with the motive that the main forms could have more complexity in the future, comparable to, for instance, WissKI with its knowledge paths where not only one individual can be directly created, but more individuals can be written to with only one form [SG12]. Especially with such forms,

when the access to the underlying ontology gets much harder, this differentiation of ontology and domain experts makes more sense.

Simplifying the user interface by limiting user control to a few details, such as name, description, and range value, helps to address this issue. Many more complicated concepts are not selectable, and other things are forced. For instance, each property has to be created with a domain - which is set automatically by the tool. Multiple domains values are not possible, just like leaving the domain empty. Furthermore, some more advanced OWL concepts are completely dismissed, like functional or transitive properties, both special kinds of properties, which lead to changes in the reasoning behavior.
Clearly some of these tool restrictions may lead to cumbersome definitions, if for example a property may only be used for one domain. For instance, a property "containsLactose" could be useful for multiple domains, as it may be relevant for drinks and foods. If this property is created in a food form, then however the property can be only used for foods. But, as seen in Figure 4.2, the ontology expert may always just change such properties afterwards and properly embed them into the property, and in this example extend the domain to also drinks. For this tool, created properties and classes have to be sufficiently described to be used immediately, but further details can be always be added later.

Another important topic is the subclass graph. The original idea of this concept is to show the user the class hierarchy, and to make it easy to create subclasses, without having to search them by text, as with the classes a pre-filter is not possible, in contrast to the domain-bound property selection.
But, in its current form this subclass graph has some usability problems: The readability is strongly dependent of the ontology size and structure. The Food & Wines ontology is a good example for this problem, as it has too many classes which are direct subclasses of owl:Thing. The effect of this on the graphical representation is shown in Figure 6.5. This large obfuscated ring occurs because a very large number of classes are directly connected with owl:Thing. Even when using various graph algorithms the clarity of this subclass graph is not really improved.

A more clear tree structure also cannot be used, because the OWL standard allows circles and multiple inheritance, which break the tree structure. Because of this, the question arises if the subclass graph is really the right way to show the class hierarchy, or if another method is needed. A dual view with a graph structure and a textual search function could be a solution.
Also, another problem was already mentioned at the property-creation-functionality: The concept of subclassing as a more complicated OWL/RDFS feature may not be clear to users. Thus, it raises questions about the presumed knowledge level of domain experts, and if it is realistic if a domain expert can actually use such features without at least some basic knowledge of ontology concepts.

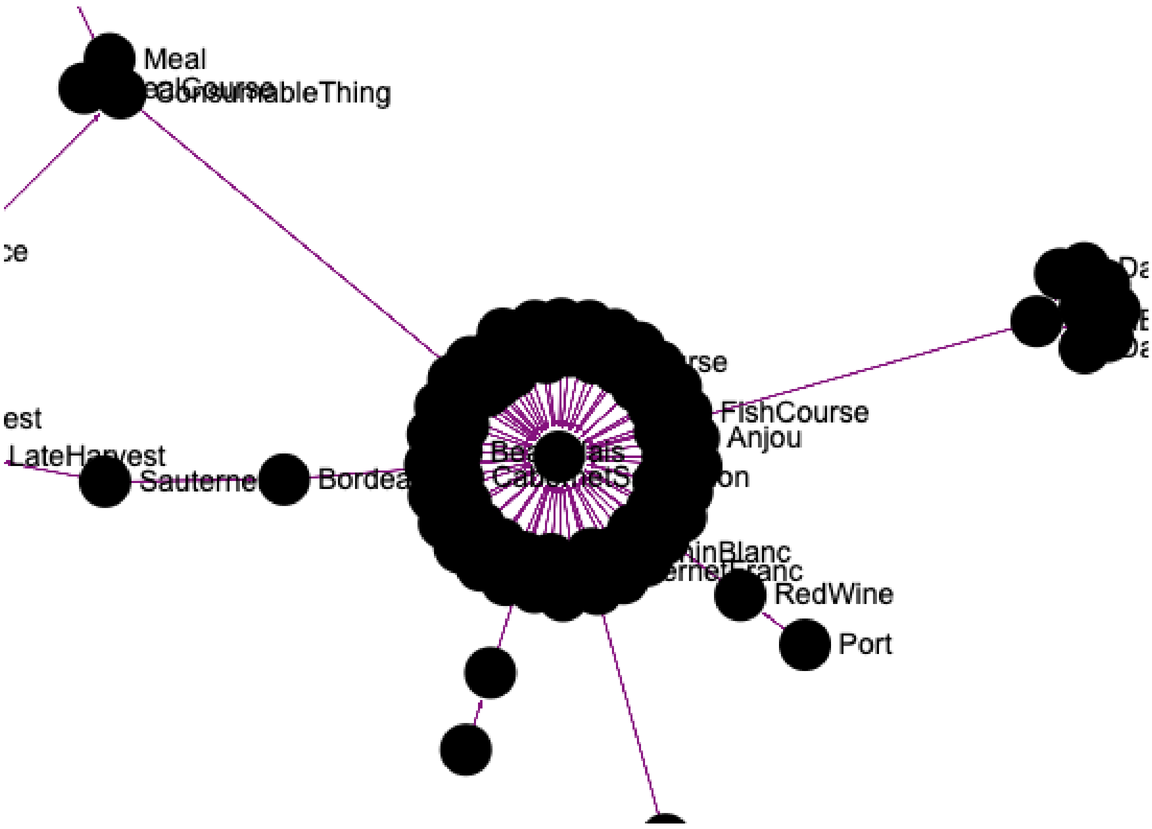


Figure 6.5: Excerpt of a subclass graph with too many direct subclasses of owl:Thing

The entry point for ontology extension features is currently limited to form field creation during form filling, which means that currently not all ontology extensions can be done in the tool, for example the creation of new classes independently of form fields. From a technical point of view, introducing new concepts when creating forms could be considered, but this would increase the tool's complexity and shift its focus towards a general ontology editor.

6.2.3 Evaluation of the Validation features

Validation with OWL reasoning

General Points As described in chapter 5, several methods for ensuring a valid knowledge base were implemented, with a significant portion relying on a logical reasoner. While theoretically helpful and easily implementable using Jena transactions for rollback mechanisms, practical issues arise with reasoner runtime, which can range from seconds to minutes, as first tests showed. This arises the question if a reasoner can even be used in a web context, as the user has to wait for the result. To analyze this further, a performance test series was conducted, which is presented in the following paragraph.
Also, a reasoner can only scan the current state of a knowledge base, requiring scans at every change to ensure consistency. Inconsistent imported ontologies pose a problem, as in these no operation could be done because the reasoner is always failing. To counteract this, for this prototype a "force"-mode was created so that when this situation occurs, a change can still be done. For a productive system of course this approach can lead to problems.

Finally, ensuring usability for domain experts is challenging. To make error messages more readable the ManchesterOWLSyntax was used as shown in subsection 5.5.2, which leads to a better presentation of the affected triples. But this is not yet enough to really communicate with a domain expert. This syntax only slightly improves the concrete syntax of the triples, but in an error case they are insufficient for effective communication for an OWL beginner. A possible solution might be a short text summary based on the axioms and an additional evaluation step to present understandable text for common consistency problems, like the usage of a forbidden property value.

Performance To assess the extent of performance degradation when utilizing reasoners, a series of performance tests was conducted. The setup was as follows: Different sizes of knowledge bases, consistent and inconsistent ones, were processed by two different reasoners, and the runtime of the reasoning process was measured. As knowledge base again the consistent Food & Wines ontology was used, with a varying number of individuals from specific classes. As classes the newly introduced Order class and Chardonnay were used. The Order class is relatively simple with few restrictions, making reasoning straightforward. In contrast, the Chardonnay class has up to 11 restrictions, some of which are inferred by the class system, making the reasoning process more complex. This setup allowed for experimentation with the impact of checking restrictions. The following configurations were used:

- S: Standard Food & Wines without additional individuals
- O100: S with additional 100 orders (each with one property value)
- O1000: S with additional 1000 orders (each with one property value)
- C100: S with additional 100 chardonnays (each with one property value)
- C1000: S with additional 1000 chardonnays (each with one property value)

For this performance test, only the number of individuals was varied, while other ontology elements, such as the number of properties or restrictions, remained constant to avoid complicating the performance test further.
Another test series was conducted with the same five setups but included an inconsistent individual in each to analyze performance with incorrect knowledge bases.

The tests were conducted with two reasoners: HermiT and JFact. Both were successfully integrated with the OntoFormGenerator and used in the tests. For HermiT also the performance of generation of explanation axioms was tested separately, while this could not be successfully implemented with JFact. In the first test run, only the status of consistency or inconsistency was checked without generating explanation axioms. All test measurements were repeated ten times, and the arithmetic mean was calculated to mitigate fluctuations due to varying CPU times.

The measured time is the duration of the validation API call as reported by the web browser, Mozilla Firefox. All tests were conducted on a locally running web server, using an Apple M2 processor and 16 GB RAM. This setup excludes frontend rendering times and network latencies but includes the overhead of the Spring framework in handling API calls and responses. Therefore, the values should be considered as relative, comparable times rather than absolute results.

The results are presented as table in Table 6.1 and as graph in Figure 6.6.

The data indicates that reasoner runtimes, and the feasibility of using reasoners for live consistency checks, are highly dependent on the size of the knowledge base. For small knowledge bases with only 100 individuals, the runtimes are approximately

	1	100	1000
O, HermiT	910	938	1121
C, HermiT	910	1782	29678
O, JFact	1022	1096	1025
C, JFact	1022	1505	1912
O, HermiT, incons.	755	794	998
C, HermiT, incons.	755	840	959
O, JFact, incons.	830	845	954
C, JFact, incons.	830	939	1073

Table 6.1: Reasoner runtimes: Absolute numbers (with removed outliers; in ms)

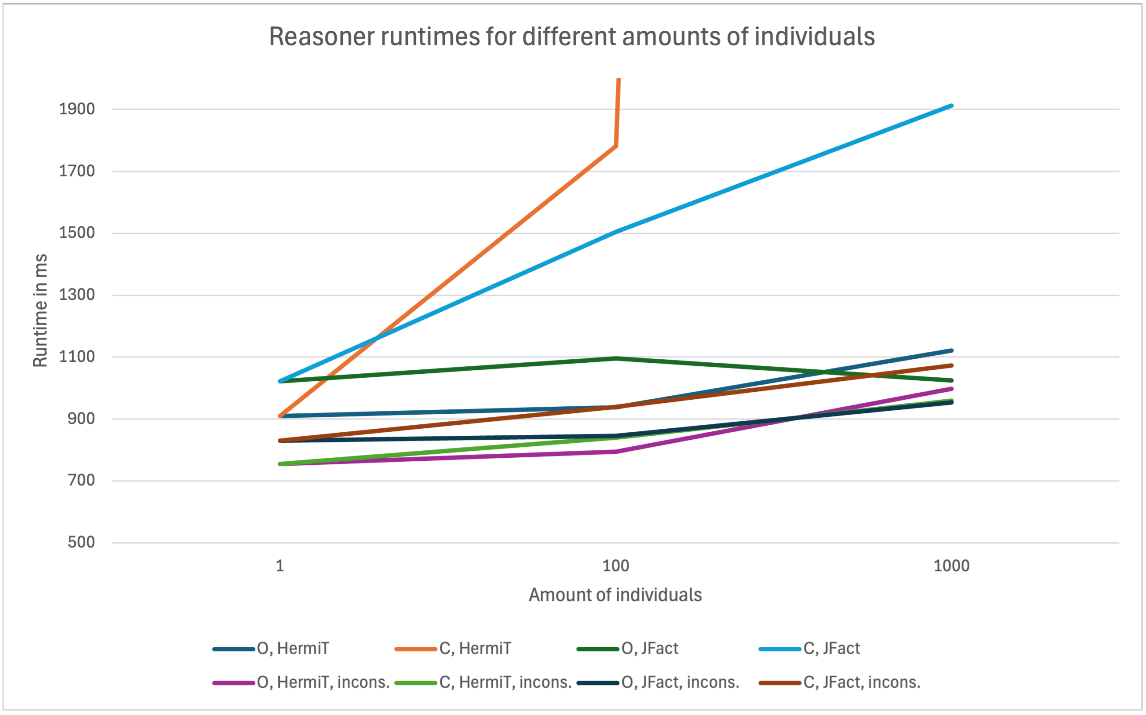


Figure 6.6: Reasoner runtimes: Graphical depiction

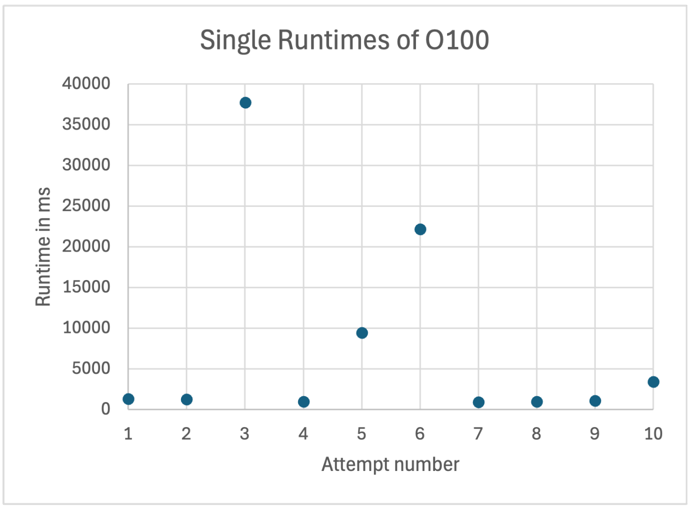


Figure 6.7: Single reasoner runtimes of (O100, JFact, cons.) in ms

	S	O100	O1000	C100	C1000
Without explain.	755	794	998	840	959
With explain.	1780	1745	2076	2196	124177

Table 6.2: Comparison of runtimes with explanation generation (in ms)

one second, which is acceptable in a web context, especially since individual insertion is also very fast. For all tests the runtime increases with a higher number of individuals, but to a different degree depending on which individuals are added and which reasoner is used. Notably, HermiT exhibits significantly longer runtimes with many Chardonnay individuals due to the numerous restrictions to validate, reaching up to half a minute. JFact also shows a substantial increase in runtime, although it is generally faster than HermiT. However, JFact has a significant drawback: it is not very stable, with some validations taking more than ten times the average duration or even resulting in endless loops. This instability is evident even with smaller knowledge base sizes, as shown in Figure 6.7. This probabilistic performance issue is a significant problem for user experience and did not occur with HermiT.
Inconsistent knowledge bases are detected much faster than proving consistent ones in both reasoners, as shown in Table 6.1. This is a problem because consistent knowledge bases are the standard case, while inconsistent knowledge bases should be the exception. So, the runtime of the average case of a validation run can be very slow.

In practice, inconsistent knowledge bases should not only be detected but the causing axioms should also be provided as explanations. For HermiT, this significantly increases the total runtime, as shown in Table 6.2. The runtimes are at least doubled when generating explanations, and for larger, more complex knowledge bases, the runtime increases dramatically. No results could be presented for JFact, but it is likely that the runtime would also increase.

Overall, the experiment demonstrates that reasoner validation is only suitable for smaller knowledge bases or those with few properties and restrictions.
An open topic is the potential optimization of the reasoner process, possibly by validating only parts of the knowledge bases, especially when only one individual is inserted. This would require backend logic to use only relevant parts of the knowledge bases, which is a topic for further research.

Summary of using OWL reasoning In summary, the usage of OWL reasoning remains questionable for a web-based tool. As advantages, it can ensure with certainty that the knowledge base is consistent, and it can provide detailed error messages. But, communicating these error messages to a domain expert is difficult. Also, the runtime of the reasoner is a critical point, as it can be too long for a web context, especially in the good case of a consistent knowledge base, although it remains open, if optimization of the reasoner process could improve the runtime.
These results also highlight the benefit of using other validation methods, such as checking restrictions before filling forms in the UI.

Other Approaches

The other implemented approaches of validating the knowledge base come with own advantages and disadvantages. Notably, incorporating restrictions earlier in the process and embedding them directly into the workflow could be a promising direction. This approach eliminates the need for complex reasoner checks by integrating restrictions at appropriate points within the form editor and filler. Also, this method aligns with the tool's overarching strategy of implicitly adhering to OWL and RDFS rules concerning class systems, domains, and ranges.
The primary advantage of this approach is its superior performance, as it prevents the tool from being blocked for extended periods. Additionally, depending on the feature, the implementation can be seamlessly integrated into other UI elements, obviating the need for special explanations. For instance, the "allValuesFrom" restriction demonstrates that end users do not need to be explicitly informed about any restrictions. This approach also proves beneficial by not only focusing on domain experts but also presenting restrictions to ontology experts. This is particularly useful for cardinality values, ensuring that forms do not permit inconsistent entries.
However, the main drawback is that, unlike the general reasoner application, this approach does not allow for a universal implementation for all types of restrictions.

Instead, each restriction type requires its own solution. This includes various OWL constructs that have not yet been implemented [MPSP12]:

- Union, intersection and complement class expressions
- Existential quantification restrictions: “someValuesFrom”
- Individual values restrictions (“hasValue”) and self-restrictions
- Restriction types for datatype properties

The third approach, using the research on ontology design pitfalls, did not bring many effects, but mostly just because many of the pitfalls are already addressed by the tool and its form-based approach. The remaining pitfalls in this specific example only offer minor stylistic benefits, such as consistent naming of concepts.

Ultimately, there are multiple methods for automatically validating knowledge bases created via forms, but none of the investigated methods are perfect. Finally, this whole topic of consistent knowledge bases is highly dependent of the own use case. The more restrictive an ontology, the more checks are required, like with the wine classes in the Wines & Food ontology. This leads to the consideration that a less restrictive ontology, resembling a taxonomy, might be easier to manage in a form-based approach. The key question is the balance what restrictions are implicitly built by correctly designing forms and choosing form fields including cardinality correctly, and what is content of the ontology itself. Another factor is the tool's functionality: the more features, such as cardinality constraints, that are offered, the more checks are needed for restriction violations.

6.3 Summary and Open Questions

In summary, this project showed that a general form generator for ontology knowledge bases is feasible to implement. As shown in the preceding chapters, a simple system capable of creating forms and populating them with data was developed in an architectural and technically rather easy solution. This is largely due to the availability of necessary technologies, such as Ontology APIs, triple stores, and HTML for form display, making it relatively straightforward to create a system with basic functionality.
However, the project also highlighted some minor design challenges in the general form generator domain that are not frequently addressed in the literature, particularly concerning IRI generation and the handling of imports and exports.

In this project the first time the idea of a form generator was combined with the idea of ontology extension to develop a concrete application ontology from a more general domain ontology. Therefore, by designing a concrete workflow for this application, an idea was created to integrate the ontology engineering workflow into the form generator, a concept not previously seen in the literature. Consequently, the decision was made to integrate the engineering workflow directly into the form filler, allowing users greater flexibility in creating new fields and, if necessary, new concepts. It was explicitly decided to support only the addition of concepts and individuals, not their deletion or modification. While technically feasible, the latter approach presents conceptual and usability challenges, as discussed in previous sections. Some ideas were suggested and implemented, such as a sophisticated search function and automation to assist users in creating new concepts. These ideas also touch on the broader topics of user acceptance and usability, exemplified by the introduction of a draft system as an additional help feature.

One significant challenge is the huge complexity of OWL ontologies. OWL allows a variety of complex features to define a semantic framework for an ontology, but the mapping of these features to easy understandable forms for domain experts is not trivial. In many cases, the approach involves restricting users and prohibiting the use of certain features, such as only allowing simple one-subclass relations or excluding special property types like functional or transitive properties.
However, this approach is not universally applicable. For example, when using OWL, adding individuals and concepts to the knowledge base without causing contradictions or weakening the overall ontology design can be challenging depending on the ontology's complexity. This would also be a problem when only using RDFS, as the subclass system also exists there. But, of course the restrictions are not as strong as in OWL, so a reducing of the supported ontologies to RDFS could be a solution for a simpler tool.
Various solutions for validating the modified knowledge base and ontology were proposed and evaluated, but each has its own drawbacks. These drawbacks include technical issues, such as runtime performance, and usability challenges, particularly in communicating problems to users. Additionally, it was demonstrated that automatic ontology validation is inherently complex, and many aspects cannot be fully automated.
Another issue is that OWL ontologies do not need to be complete. While some rules can be inferred by OWL's semantic rules, many information fields, such as domain and range values of properties, can remain unspecified in a valid OWL ontology. This poses a challenge for any form generator, which must find ways to offer individual lists and extract ontology information despite incomplete data. Also, effective validation of the knowledge base becomes more difficult with missing information.

So, this project demonstrated that even a well-designed form generator is highly dependent on the underlying ontology. Therefore, it is crucial for ontology editors to carefully design ontologies used as domain ontologies, considering the following aspects:

- Carefully use restrictions. The more restrictions and rules an ontology has, the more likely the validation mechanism could fail and consistency is harder to guarantee.
- Use a clear class hierarchy and appropriate properties with understandable names and descriptions, so that is easy to find fitting classes and properties, reducing the likelihood of a user creating a redundant new class.
- Create the ontology as complete as possible regarding annotation properties, domain and range values, so that the form generator can work with the ontology as effectively as possible, and extract all necessary information for setting up forms.

However, many open questions remain. The primary issue is that the suggested features and ideas have not yet been validated in a real-world scenario. Thus, it remains uncertain whether a domain expert can effectively use the tool and whether it is truly comprehensible. This is particularly important for features lacking established UX principles, such as the class graph. The only way to address this question is through a real user study. Domain and ontology experts should use the tool and test the proposed workflow. A user study was not conducted in this thesis due to time constraints. Additionally, the Food & Wines ontology, while suitable for testing general concepts, is insufficient for a realistic application test. The concept search is relatively simple in this ontology, making it more interesting to test the tool in a complex scientific domain, such as biology or medicine, to evaluate the proposed features and workflow. Conducting a user study of this scope was beyond the capacity of this thesis. But especially because this combination of form generator and ontology extension is a novel concept, such a real-life study should be a priority for future research.

Furthermore, general User Experience research could be valuable in identifying additional features and improvements for better communication with domain experts. This is closely related to usability. As a user study was not conducted, this topic was not addressed in this thesis. This would involve a deeper analysis of how to present forms and additional features clearly and understandably for all users, including the functions of text fields, the impact of property names and comments, and the purpose of each button.
A user study might also reveal that the underlying assumptions of the general workflow do not align with reality. For instance, it could be that domain experts have more knowledge of the concepts than assumed, making the tool overly restrictive. In general, usability research is interesting for these kinds of tools. The survey indicated

that the popularity of these generators is not very high, making user acceptance a critical open topic, especially when compared to other electronic forms without an ontology background.

Finally, another open question is, how the communication between domain and ontology experts can be improved also in this tool, for instance by using messaging systems when a new concept is created.
But on the other hand, an interesting question also is, how far the need for an ontology expert can be reduced. Especially, with more validation features, more user guidance, and features to erase or correct faulty concepts, it could be possible that an ontology expert is only needed in exceptional cases. This might be an interesting idea for further studies.

Chapter 7

Conclusion

In summary, the objectives, which were set for this thesis, were successfully achieved. First, a comprehensive survey on ontology-based form generation approaches was conducted. For this purpose, also a basic introduction to OWL and general ontology concepts was given. Building on the survey results, a prototype for an ontology-driven form generator was developed, focusing on extending a general domain ontology into a specialized application ontology during the form filling process. Both the survey and the prototype addressed the research questions of this thesis, and the findings are summarized as follows:

1. Which ontology-driven data acquisition form generators already exist in research projects? The survey revealed that several tools for ontology-driven form generation exist, although this research area remains niche with limited approaches. Each of these tools has its own specific purpose, often only as secondary tool for another research question, where the tool is used for information acquisition. Depending on that also the impact on the ontology's knowledge base varies for each tool. Some tools focus on directly populating the ontology with new data, while others emphasize saving the complete forms with questions and answers. All tools utilize the ontology, typically by using datatypes and property ranges.
Interestingly, aside from basic functions, few additional features are implemented in these tools, particularly regarding the usability of electronic forms in general or ontology modification. Validation of the knowledge base, beyond extracting some elements, is also not a focus in these tools. So, it could be shown that this is still an open topic with many opportunities for further research, making the proposed prototype a valuable contribution to this field.

2. How can an own ontology-driven data acquisition form generator be implemented? The prototype was successfully implemented, and the main features of a basic ontology-driven form generator could be realized. The basic implementation was straightforward due to the availability of technologies for easy implementation of ontology-driven software: For web forms, HTML forms are directly suitable, and in the Java technology stack many libraries for ontology operations are available including integrated triple stores for a persistence layer. Also, these modern, open-source technologies are suitable for long-term projects, with the hope that the prototype can be further developed in the future. The straightforward tech stack allowed a focus on good software design, resulting in a layered architecture, clear API design and documentation. Also, some attention was given to creating a good user experience with some basic usability features.

3. How can additional features regarding ontology extension, especially ones not implemented in previous projects, be added to this tool? Based on the survey results, a focus was set on developing the workflow of ontology engineering for domain experts without ontology knowledge in the prototype. Building up on a suggested workflow for collaboration between domain experts and ontology experts using a form generator, features for form and ontology extensions were built and evaluated. In the evaluation of the prototype it could be shown that these features were successfully implemented, improving the basic form generator tool. These features in detail include options to temporarily extend forms and permanently add classes and properties within a provided framework. The design of these features, especially regarding usability, was discussed, and important questions regarding the validation of such a modified knowledge base were investigated, although many open questions remain.

The results of this thesis provide a solid foundation for further research in combining form generation with ontology engineering. Specific open points for further development of the prototype were discussed. These include the need for real-life applications with actual users and more studies on the user experience of using ontology-generated forms.
Additionally, the prototype could be further developed into a platform for collaborative ontology development. This would involve adding communication features and a multi-user setting with different roles. A potential idea is to establish a notification system for domain expert additions, allowing ontology experts to review and decide on the inclusion of new concepts in the ontology. This could also extend to the form editing component, where domain experts suggest permanent changes, and ontology experts decide on their inclusion.

Also, the survey indicated other directions for ontology-based forms that could be explored. For instance, the automatic generation of forms from ontologies, in contrast to manually created ones, could be an interesting topic for further research.

Appendix

A1. Analysis of the 40 Pitfalls of Ontology Design

For validation purposes the work by Poveda-Villalón et al. was regarded about 40 common pitfalls in ontology design [PV+14]. Only 32 of these pitfalls are automatically testable and therefore relevant for the OntoFormGenerator. The results of the analysis are shown in Table 7.1.

ID	Pitfall	Result	Comment
2	**Synonyms as Classes**	**implemented**	**Using Datamuse for synonym search and TextRazot for Lemmatization as simple Proof of Concept**
3	Using wrong primitives for Typing and Subclassing	already ensured	Already ensured by Jena
4	Creating unconnected ontology elements	already ensured	Already ensured by the form/ontology expansion approach
5	Defining wrong inverse Relationships	not relevant	Inverse relationships not covered by the tool
6	Cycles in the class hierarchy	already ensured	No cycles possible in new classes

7	**Multiple concepts in one class**	**implemented**	**Simple word filter including "and", "or"... as Proof of Concept**
8	Missing annotation properties	partially implemented	Label and description are given for new classes and properties; other properties not implemented
10	Missing disjointness	open	Too complicated for domain experts; implementation of automatism may be difficult
11	Missing domain and range	already ensured	Required fields when creating property
12	Missing equivalent properties when importing	not relevant	No manual import possible
13	Missing inverse relationship	open	Similar to 10
19	Swapping intersection and union	not relevant	Only one domain/ range
20	Misusing ontology annotations	not relevant	Only label/ description used
21	**Using a miscellaneous class**	**implemented**	**Simple word filter as Proof of Concept**

22	**Using different naming criteria**	**implemented**	**Algorithm to compare naming convention of new concept to other ones implemented**
24	Using recursive definition	open	
25-29	Pitfalls of defining inverse, symmetric, or transitive properties	not relevant	No special properties definable in the tool
30-31	Pitfalls of defining equivalent classes	not relevant	No imports (similar to 12)
32	Classes with the same label	open	
33	Property chain with only one property	open	
34	Untyped class	already ensured	Prohibited by Jena
35	Untyped property	already ensured	Prohibited by Jena
36	URI contains file extension	already ensured	URIs are automatically created by the tool
37	Ontology not available	not relevant	Cannot be ensured by the tool
38	No ontology declaration	already ensured	Ensured by Jena export
39	Ambiguous namespace	not relevant	Cannot be ensured by the tool

40	Namespace hijacking	open	

Table 7.1: Analysis of the relevance of 40 pitfalls of Ontology Design for OntoFormGenerator

A2. Definition of the Restaurant Example Ontology

```
    <?xml version="1.0"?>
<rdf:RDF xmlns="http://www.semanticweb.org/dominik/ontologies
    /2024/6/untitled-ontology-56/"
      xml:base="http://www.semanticweb.org/dominik/ontologies
         /2024/6/untitled-ontology-56/"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xml="http://www.w3.org/XML/1998/namespace"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://ontologies.ontoformgenerator.
        de/MyRestaurant">
         <owl:imports rdf:resource="http://www.w3.org/TR/2003/PR-
            owl-guide-20031209/food"/>
    </owl:Ontology>

    <owl:ObjectProperty rdf:about="http://ontologies.
        ontoformgenerator.de/MyRestaurant#hasOrderedDrink">
         <rdfs:domain rdf:resource="http://ontologies.
            ontoformgenerator.de/MyRestaurant#Order"/>
         <rdfs:range rdf:resource="http://www.w3.org/TR/2003/PR-
            owl-guide-20031209/food#PotableLiquid"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://ontologies.
        ontoformgenerator.de/MyRestaurant#hasOrderedFood">
         <rdfs:domain rdf:resource="http://ontologies.
            ontoformgenerator.de/MyRestaurant#Order"/>
         <rdfs:range rdf:resource="http://www.w3.org/TR/2003/PR-
            owl-guide-20031209/food#EdibleThing"/>
    </owl:ObjectProperty>
    <owl:DatatypeProperty rdf:about="http://ontologies.
        ontoformgenerator.de/MyRestaurant#hasID">
        <rdfs:domain rdf:resource="http://ontologies.
            ontoformgenerator.de/MyRestaurant#Order"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/
            XMLSchema#int"/>
    </owl:DatatypeProperty>

    <owl:DatatypeProperty rdf:about="http://ontologies.
        ontoformgenerator.de/MyRestaurant#hasSpecialWish">
        <rdfs:domain rdf:resource="http://ontologies.
            ontoformgenerator.de/MyRestaurant#Order"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/
            XMLSchema#string"/>
        <rdfs:comment>Field for special wishes or remarks that
            customers have for this order.
Examples for special wishes: Other sides dishes, alternate
   ingredients</rdfs:comment>
        <rdfs:label>hasSpecialWish</rdfs:label>
    </owl:DatatypeProperty>

    <owl:DatatypeProperty rdf:about="http://ontologies.
        ontoformgenerator.de/MyRestaurant#hasTimestamp">
        <rdfs:domain rdf:resource="http://ontologies.
            ontoformgenerator.de/MyRestaurant#Order"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/
            XMLSchema#dateTime"/>
    </owl:DatatypeProperty>


    <owl:Class rdf:about="http://ontologies.ontoformgenerator.de/
        MyRestaurant#Order"/>
</rdf:RDF>
```

Bibliography

[AA20] Sahin Aydin and Mehmet N. Aydin. “Ontology-based data acquisition model development for agricultural open data platforms and implementation of OWL2MVC tool”. In: *Computers and Electronics in Agriculture* 175 (2020).

[AM23] Hussam Alkaissi and Samy I McFarlane. “Artificial Hallucinations in ChatGPT: Implications in Scientific Writing”. In: *Cureus* 15.2 (2023).

[BA+10] J.A. Bargas-Avila et al. “Simple but Crucial User Interfaces in the World Wide Web: Introducing 20 Guidelines for Usable Web Form Design”. In: *User Interfaces*. InTech, 2010.

[Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik 1: Basiskonzepte und Requirements-Engineering*. 3rd ed. Heidelberg: Spektrum, 2009.

[Bal11] Helmut Balzert. *Lehrbuch der Softwaretechnik - Entwurf, Implementierung, Installation und Betrieb*. 3rd ed. Heidelberg: Spektrum, 2011.

[Bec+04] Sean Bechhofer et al. *OWL Web Ontology Language Reference - W3C Recommendation*. 2004. URL: `https://www.w3.org/TR/owl-ref/` (visited on 11/13/2024).

[Bec+14] David Beckett et al. *RDF 1.1 Turtle - W3C Recommendation*. 2014. URL: `https://www.w3.org/TR/turtle/` (visited on 11/13/2024).

[BG14] Dan Brickley and R.V. Guha. *RDF Schema 1.1 - W3C Recommendation*. 2014. URL: `https://www.w3.org/TR/rdf-schema/` (visited on 11/13/2024).

[BGP09] Eva Blomqvist, Aldo Gangemi, and Valentina Presutti. “Experiments on Pattern-based Ontology Design”. In: *K-CAP ’09: Proceedings of the fifth international conference on Knowledge capture*. ACM Digital Library, 2009, pp. 41–48.

[Bha+08] Ravish Bhagdev et al. “Creating and Using Organisational Semantic Webs in Large Networked Organisations”. In: *The Semantic Web - ISWC 2008, Proceedings of the Seventh International Semantic Web Conference*. Karlsruhe, Germany: Springer, 2008, pp. 723–736.

[BHS04] Franz Baader, Ian Horrocks, and Ulrike Sattler. "Description Logics". In: *Handbook on Ontologies*. Ed. by Steffen Staab and Rudi Studer. Berlin, Heidelberg: Springer, 2004, pp. 3–28.

[BKB00] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. "Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service". In: *CHI 2000* 2.1 (2000).

[Bro24a] Broadcom. *Building REST services with Spring - Spring Guide*. 2024. URL: `https://spring.io/guides/tutorials/rest` (visited on 11/13/2024).

[Bro24b] Broadcom. *Dependency Injection - Spring Framework*. 2024. URL: `https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html` (visited on 11/13/2024).

[BT+21] Shadi Baghernezhad-Tabasi et al. "IOPE: Interactive Ontology Population and Enrichment Guided by Ontological Constraints". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 13080 LNCS. Springer, 2021, pp. 321–336.

[CA08] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reuseable .NET Libraries*. 2nd edition. Addison-Weasley Professional, 2008.

[Cam24] Cambridge Semantics. *RDFS vs. Owl - Cambridge Semantics*. 2024. URL: `https://web.archive.org/web/20240725192257/https://cambridgesemantics.com/blog/semantic-university/learn-owl-rdfs/rdfs-vs-owl/` (visited on 11/22/2024).

[dbl24] dblp team. *What institution is behind dblp? - dblp*. 2024. URL: `https://dblp.org/faq/What+institution+is+behind+dblp.html` (visited on 11/13/2024).

[DM08] Martin Dzbor and Enrico Motta. "Engineering and Customizing Ontologies: The Human-Computer Challenge in Ontology Engineering". In: *Ontology Management: Semantic Web, Semantic Web Services, and Business Applications*. Ed. by Martin Hepp et al. New York City, USA: Springer, 2008, pp. 25–58.

[DPL24] Kyzer R. Davis, Brad G. Peabody, and Paul J. Leach. *Universally Unique IDentifiers (UUIDs) - RFC 9562*. 2024. URL: `https://datatracker.ietf.org/doc/html/rfc9562` (visited on 11/13/2024).

[DZP10] Euthymios Drymonas, Kalliopi Zervanou, and Euripides G.M. Petrakis. “Unsupervised Ontology Acquisition from Plain Texts: The OntoGain System”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6177 (2010), pp. 277–287.

[Fal+08] Matthew E. Falagas et al. “Comparison of PubMed, Scopus, Web of Science, and Google Scholar: strengths and weaknesses”. In: *The FASEB Journal* 22.2 (2008), pp. 338–342.

[Fat+20] Rubia Fatima et al. “Google Scholar vs. Dblp vs. Microsoft Academic Search: An Indexing Comparison for Software Engineering Literature”. In: *Proceedings - 2020 IEEE 44th Annual Computers, Software, and Applications Conference, COMPSAC 2020*. Institute of Electrical and Electronics Engineers Inc., July 2020, pp. 1097–1098.

[FR12] Mark Fichtner and Vincent Ribaud. “Paths and shortcuts in an event-oriented ontology”. In: *Communications in Computer and Information Science* (2012), pp. 214–226.

[Gar+12] Roberto Garcia et al. “OntoWiki - An Authoring, Publication and Visualization Interface for the Data Web”. In: *Semantic Web* 1 (2012), pp. 1–5.

[Ger24a] Germanisches Nationalmuseum Stiftung des öffentlichen Rechts. *Features of WissKI - wiss-ki.eu*. 2024. URL: `https://wiss-ki.eu/features` (visited on 11/13/2024).

[Ger24b] Germanisches Nationalmuseum Stiftung des öffentlichen Rechts. *WissKI software system, Source code*. 2024. URL: `https://git.drupalcode.org/project/wisski.git` (visited on 11/13/2024).

[GL02] Michael Gruninger and Jintae Lee. “Ontology: Applications and Design”. In: *Communications of the ACM* 45.2 (2002), pp. 39–41.

[Gli+14] Birte Glimm et al. “HermiT: An OWL 2 Reasoner”. In: *Journal of Automated Reasoning* 53.3 (2014), pp. 245–269.

[Gon+17] Rafael S. Gonçalves et al. “An ontology-driven tool for structured data acquisition using Web forms”. In: *Journal of Biomedical Semantics* 8.1 (2017).

[Goo24] Google. *GitHub repository of Google Material Lite: Material Design Components in HTML/CSS/JS*. 2024. URL: `https://github.com/google/material-design-lite` (visited on 11/22/2024).

[Goy+20] Annamaria Goy et al. “Building Semantic Metadata for Historical Archives through an Ontology-driven User Interface”. In: *ACM Journal on Computing and Cultural Heritage* 13.3 (2020).

[GP09] Aldo Gangemi and Valentina Presutti. “Ontology Design Patterns”. In: *Handbook on Ontologies*. Ed. by Steffen Staab and Rudi Studer. Berlin, Heidelberg: Springer, 2009, pp. 221–243.

[Gru93] Thomas R. Gruber. “A translation approach to portable ontology specifications”. In: *Knowledge Acquisition* 5 (1993), pp. 199–220.

[GU96] Michael Gruninger and Mike Uschold. “Ontologies: Principles, Methods and Applications”. In: *The Knowledge Engineering Review* 11.2 (1996).

[Gua98] Nicola Guarino. “Formal Ontology and Information Systems”. In: *Formal Ontology in Informations Systems*. Ed. by Nicola Guarino. Padova, Italia: IOS Press, 1998, pp. 3–18.

[GW09] Nicola Guarino and Christopher A. Welty. “An Overview of OntoClean”. In: *Handbook on Ontologies*. Ed. by Steffen Staab and Rudi Studer. Springer Berlin Heidelberg, 2009, pp. 201–220.

[GW12] Christine Golbreich and Evan K. Wallace. *OWL 2 Web Ontology Language New Features and Rationale (Second Edition) - W3C Recommendation*. 2012. URL: `https://www.w3.org/TR/owl2-new-features/` (visited on 11/22/2024).

[Gö21] Christoph Göpfert. *OnForm GitLab Repository*. 2021. URL: `https://gitlab.hrz.tu-chemnitz.de/vsr/onform` (visited on 11/22/2024).

[Hal12] Wesley Hales. *HTML5 and JavaScript Web Apps*. O’Reilly, 2012.

[Hep08] Martin Hepp. “Ontologies: State of the Art, Business Potential, and Grand Challenges”. In: *Ontology Management: Semantic Web, Semantic Web Services, and Business Applications*. Ed. by Martin Hepp et al. 1st. New York City, USA: Springer, 2008, pp. 3–24.

[Hit+08] Pascal Hitzler et al. *Semantic Web*. Berlin, Heidelberg: Springer, 2008.

[Hit+12] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer (Second Edition) - W3C Recommendation*. 2012. URL: `https://www.w3.org/TR/owl2-primer/#What_is_OWL_2.3F` (visited on 11/22/2024).

[HKB09] Wolfgang Holzinger, Bernhard Krüpl, and Robert Baumgartner. “Automated Ontology-Driven Metasearch Generation with Metamorph”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5802 (2009), pp. 473–480.

[Hor12] Matthew Horridge. *OWL 2 Web Ontology Language Manchester Syntax (Second Edition) - W3C Working Group Note*. 2012. URL: `https://www.w3.org/TR/owl2-manchester-syntax/` (visited on 11/22/2024).

[Hor+14] Matthew Horridge et al. “A domain specific ontology authoring environment for a clinical documentation system”. In: *Proceedings - IEEE Symposium on Computer-Based Medical Systems*. Institute of Electrical and Electronics Engineers Inc., 2014, pp. 329–334.

[HPS14] Patrick J. Hayes and Peter F. Patel-Schneider. *RDF 1.1 Semantics - W3C Recommendation*. 2014. URL: `https://www.w3.org/TR/rdf11-mt/` (visited on 11/22/2024).

[HR10] Armin Haller and Florian Rosenberg. “A Semantic Web Enabled form model and restful service implementation”. In: *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. Perth, WA, USA: IEEE, Dec. 2010, pp. 1–8. (Visited on 11/25/2024).

[HS04] Udo Hahn and Stefan Schulz. “Building a Very Large Ontology from Medical Thesauri”. In: *Handbook on Ontologies*. Ed. by Steffen Staab and Rudi Studer. Berlin, Heidelberg: Springer, 2004, pp. 133–150.

[IBM24] IBM Corporation. *Advantages of Java - IBM Documentation*. 2024. URL: `https://www.ibm.com/docs/en/aix/7.2?topic=monitoring-advantages-java` (visited on 11/22/2024).

[Inf13] Information Management Group at the School of Computer Science. *FaCT++ reasoner - OWL research at the University of Manchester*. 2013. URL: `http://owl.cs.manchester.ac.uk/tools/fact/` (visited on 11/22/2024).

[Jac+14] Mathieu Jacomy et al. “ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software”. In: *PLoS ONE* 9.6 (2014).

[Jan24] Paul Jansen. *TIOBE Index - TIOBE*. 2024. URL: `https://www.tiobe.com/tiobe-index/` (visited on 11/22/2024).

[JG09] Caroline Jarret and Gerry Gaffney. *Forms that work: Designing Web Forms for Usability*. Amsterdam, Netherlands: Morgan Kaufmann, 2009.

[Kal+02] Aditya Kalyanpur et al. “An RDF Editor and Portal for the Semantic Web”. In: *Proceedings of the ECAI 2002 Workshop on Semantic Authoring, Annotation & Knowledge Markup, SAAKM@ECAI 2002, Lyon, France, July 22-26, 2002*. Ed. by Siegfried Handschuh et al. Vol. 100. CEUR Workshop Proceedings. Lyon, France: CEUR-WS.org, 2002.

[KV09] Markus Krötzsch and Denny Vrandecic. “Semantic Wikipedia”. In: *Social Semantic Web: Web 2.0 - Was nun?* Ed. by Andreas Blumauer and Tassilo Pellegrini. X.media.press. Berlin, Heidelberg: Springer, 2009, pp. 393–421.

[KWP16] Graham Klyne, Cerys Willoughby, and Kevin Page. “Annalist: A practical tool for creating, managing and sharing evolving linked data”. In: *Proceedings of the Workshop on Linked Data on the Web, LDOW 2016, co-located with 25th International World Wide Web Conference (WWW 2016)*. Ed. by Sören Auer et al. Vol. 1593. CEUR Workshop Proceedings. Montreal, Canada: CEUR-WS.org, 2016.

[Law19] Abram Lawendy. “Domain Engineering as a Solution for Bridging the Gap Between Ontology Engineers and Domain Experts”. PhD Thesis. Clausthal University of Technology, 2019.

[LGG21] André Langer, Christoph Göpfert, and Martin Gaedke. “CARDINAL: Contextualized Adaptive Research Data Description INterface Applying LinkedData”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12706. Springer, 2021, pp. 11–27.

[LYR09] Zhanjun Li, Maria C. Yang, and Karthik Ramani. “A methodology for engineering ontology acquisition and validation”. In: *AI EDAM* 23.1 (2009), pp. 37–51.

[Mai+17a] Pierre Maillot et al. “FORMULIS: Dynamic Form-Based Interface for Guided Knowledge Graph Authoring”. In: *Knowledge Engineering and Knowledge Management. EKAW 2016*. Lecture Notes in Computer Science 10180 (2017). Ed. by Paolo Ciancarini et al.

[Mai+17b] Pierre Maillot et al. “Nested Forms with Dynamic Suggestions for Quality RDF Authoring”. In: *Database and Expert Systems Applications. DEXA 2017*. Lecture Notes in Computer Science 10438 (2017). Ed. by Djamal Benslimane et al.

[Mas12] Mark Massé. *REST API Design Rulebook*. Sebastopol, USA: O’Reilly, 2012.

[Mot+17] Boris Motik et al. *HermiT Source Code*. 2017. URL: `https://github.com/phillord/hermit-reasoner/tree/master` (visited on 11/23/2024).

[MPSP12] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition - W3C Recommendation*. 2012. URL: `https://www.w3.org/TR/owl2-syntax` (visited on 11/23/2024).

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008.

[Ora19] Oracle Corporation. *Core J2EE Patterns Transfer Object - Java Technical Details*. 2019. URL: `https://www.oracle.com/java/technologies/transfer-object.html` (visited on 11/23/2024).

[Orn24] Gilvan Orneas. *Clean Architecture with Spring Boot - Baeldung*. 2024. URL: `https://www.baeldung.com/spring-boot-clean-architecture` (visited on 11/23/2024).

[Poh14] Oliver Pohl. “rdfedit: User supporting web application for creating and manipulating RDF instance data”. In: *Metadata and Semantics Research - 8th Research Conference, MTSR 2014, Karlsruhe, Germany, November 27-29, 2014. Proceedings*. Ed. by Sissi Closs et al. Vol. 478. Communications in Computer and Information Science. Karlsruhe, Germany: Springer, 2014, pp. 54–59.

[PV+14] María Poveda-Villalón et al. “OOPS! (OntOlogy Pitfall Scanner!): supporting ontology evaluation on-line”. In: *International Journal on Semantic Web and Information Systems* 10.2 (2014).

[Qu22] Yuanwei Qu. “Geological Information Capture with Sketches and Ontologies”. In: *The Semantic Web: {ESWC} 2022 Satellite Events - Hersonissos, Crete, Greece, May 29 - June 2, 2022, Proceedings*. Ed. by Paul Groth et al. Vol. 13384. Lecture Notes in Computer Science. Hersonissos, Crete, Greece: Springer, 2022, pp. 275–284.

[RRSP21] Petko Rutesic, Mirjana Radonjic-Simic, and Dennis Pfisterer. “An Enhanced Meta-model to Generate Web Forms for Ontology Population”. In: *Knowledge Graphs and Semantic Web - Third Iberoamerican Conference and Second Indo-American Conference, KGSWC 2021, Kingsville, Texas, USA, November 22-24, 2021, Proceedings*. Ed. by Boris Villazón-Terrazas et al. Communications in Computer and Information Science. Kingsville, Texas, USA: Springer, 2021, pp. 109–124.

[Sad+18] Fouad Sadki et al. “Semantically Structured Web Form and Data Storage: A Generic Ontology-Driven Approach Applied to Breast Cancer”. In: *Studies in Health Technology and Informatics*. Vol. 255. IOS Press, 2018, pp. 205–209.

[SB+13] Anila Sahar Butt et al. “ActiveRaUL: A Web form-based User Interface to Create and Maintain RDF data”. In: *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*. Ed. by Eva Blomqvist and Tudor Groza. Vol. 1035. Sydney, Australia: CEUR-WS.org, 2013, pp. 117–120.

[Sch05] Guus Schreiber. *OWL restrictions*. 2005. URL: `https://www.cs.vu.nl/~guus/public/owl-restrictions/` (visited on 11/23/2024).

[Sch+07] Sebastian Schaffert et al. “Semantic Wiki”. In: *Informatik-Spektrum* 30.6 (2007), pp. 434–439.

[Sch+14] Guus Schreiber et al. *RDF 1.1 Primer - W3C Working Group Note.* 2014. URL: https://www.w3.org/TR/rdf11-primer/ (visited on 11/23/2024).

[Sch21] Sebastian Schmidt. *Visual Ontology Modeling for Domain Experts and Business Users with metaphactory - Metaphacts.* 2021. URL: https://blog.metaphacts.com/visual-ontology-modeling-for-domain-experts-and-business-users-with-metaphactory (visited on 11/23/2024).

[SG12] Martin Scholz and Guenther Goerz. “WissKI: A virtual research environment for cultural heritage”. In: *Frontiers in Artificial Intelligence and Applications.* Vol. 242. IOS Press, 2012, pp. 1017–1018.

[SHH23] Cogan Shimizu, Karl Hammar, and Pascal Hitzler. “Modular ontology modeling”. In: *Semantic Web* 14.3 (2023), pp. 459–489.

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. *OWL Web Ontology Language Guide - W3C Recommendation.* 2004. URL: https://www.w3.org/TR/owl-guide/ (visited on 11/23/2024).

[Tec24] Technische Informationsbibliothek. *TIB Terminology Service Documentation.* 2024. URL: https://service.tib.eu/ts4tib/swagger-ui.html#/search-controller/filteredSearchUsingGET (visited on 11/23/2024).

[Tha22] Shreeya Thakur. *Advantages And Disadvantages Of Java Programming Language - Unstop.* 2022. URL: https://unstop.com/blog/advantages-and-disadvantages-of-java (visited on 11/23/2024).

[The24a] The Apache Software Foundation. *Apache Jena - TDB Documentation.* 2024. URL: https://jena.apache.org/documentation/tdb/ (visited on 11/23/2024).

[The24b] The Apache Software Foundation. *Apache Jena Homepage.* 2024. URL: https://jena.apache.org/ (visited on 11/23/2024).

[Tud+13] Tania Tudorache et al. “WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the web”. In: *Semantic Web* 4.1 (2013), pp. 89–99.

[Tud16] Tania Tudorache. *PropertyFormPortlet.* 2016. URL: https://protegewiki.stanford.edu/wiki/PropertyFormPortlet (visited on 11/23/2024).

[Vce+17] Petr Vcelak et al. “Ontology-based Web Forms – Acquisition and Modification of Structured Data”. In: *10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI 2017)*. Ed. by Qingli Li et al. Shanghai, China: IEEE, 2017, pp. 1–5.

[WHA24] WHATWG. *HTML Forms - HTML Standard*. 2024. URL: `https://html.spec.whatwg.org/multipage/forms.html` (visited on 11/23/2024).

[Wis+19] Dawid Wisniewski et al. “Analysis of Ontology Competency Questions and their formalizations in SPARQL-OWL”. In: *Journal of Web Semantics* 59 (2019).

[Wor21] World Wide Web Consortium. *LargeTripleStores - W3C Wiki*. 2021. URL: `https://www.w3.org/wiki/LargeTripleStores#Apache_Jena_(16.7B)` (visited on 11/23/2024).

[Zin20] Camilla Zinnarosu. “Analisi e design della User Interface per gli utenti finali del progetto PRiSMHA”. MA thesis. University of Turin, 2020.