Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

MUSE4Anything

Fabian Bühler

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Dr. phil. Johanna Barzen
Commenced:	Oktober 1, 2020
Completed:	March 18, 2021

Abstract

Many projects in the digital humanities need specialized tools to store and work with their data. MUSE is a digital humanities project researching film costume patterns that uses a custom-built data repository. MUSE4Music is another digital humanities project following the methodology from MUSE. Because the data for the MUSE4Music project has a different structure, a completely new custom data repository had to be built. Although the methodology used in both projects is generic, the tools used are not. Building a custom data repository for a similar project requires a huge upfront investment. A generic data repository could solve this problem and enable a variety of different projects that use similar methods as the MUSE project. This thesis is about the development of such a generic MUSE4Anything repository. Experiences from both the MUSE and MUSE4Music project are used to inform the implementation. The MUSE4Anything repository supports completely user definable ontologies. For this purpose the MUSE4Anything repository contains a taxonomy editor and an editor for the object types of the ontology.

Contents

List of Figures

List of Listings

1 Introduction

The digital humanities [BDL+12] are steadily advancing how they gather and use data for their research. This requires new methods and tools, to store and work with the data. How to curate data, share data with other projects or even store it safely long term are questions that trouble digital humanities projects [Poo17]. With experience from similar projects new and tools, that are specifically built to support digital humanities projects, can be created.

MUSE [Bar17] is a digital humanities project aiming at identifying film costume patterns. Candidates of costume patterns can be found by analysing existing patterns and similarities in costumes found in films. Methods from computer sciences as well as the computer as a tool are used to identify clusters of similar costumes used as pattern candidates, that then are verified manually [Bar18; BFL18]. For these methods to be effective, the features of many costumes have to be gathered into a computer accessible repository of costumes.

Over years costumes from multiple films were catalogued in great detail, following a previously defined ontology. The ontology defines all relevant attributes to describe a costume e. g., single clothing pieces, accessories, and related elements like the role that wore the costume, the actor playing that role and the film it appeared in. Parts of the ontology use taxonomies to define the value space of certain attributes. The hierarchical nature of the taxonomies allows both humans and computers to understand relations between the values defined in the taxonomy. The structure of the costume data in the MUSE repository is defined by the ontology. Most of the attributes have structured value spaces defined by taxonomies. Therefore, the costume data in the repository has a uniform appearance that lends itself well to computer assisted analysis using data mining or machine learning techniques.

To support the data input process and to store the costume data, the MUSE repository was developed. The MUSE repository is a specialized tool that can be accessed from a standard web browser. Data about the films, roles, and their costumes can be entered into the repository using html forms that assist the user with error checking and searchable taxonomies. While the tool supports editing of the data, editing of the ontology is not possible. Currently only rudimentary editing of some taxonomies is supported.

The costume data in the repository can then be analysed to identify costume patterns. Data mining and machine learning techniques are used to find pattern candidates. In this process the semantic knowledge integrated into the ontology and the taxonomies is used to measure the similarity of costumes. The pattern candidates can then be manually verified by an expert of the domain. Verifying the pattern candidates involves inspecting the relevant individual costumes in the MUSE repository. For this the repository provides search functionality and a costume graph as a more visual representation of the costume.

Based on MUSE, MUSE4Music [BBE+16; Büh17; EBH17] was developed, focusing on patterns in historical music from the 19th century. Both projects are naturally similar, as they aim at identifying and formalizing knowledge about a specific domain in the form of patterns. Instead of costumes,

detailed musical features of musical excerpts are captured in the specialized MUSE4Music data repository. Because the MUSE repository is a highly specialized tool, it could not be adapted to the new use case from musicology. An entirely new repository had to be developed for MUSE4Music even though the structure of the repositories is similar, except for the differences based on the project specific ontologies. Currently a prototype of the repository is already functional, but time-consuming changes and revisions of the ontology need to be incorporated manually.

A new project using the generic methodology behind MUSE and MUSE4Music would need another custom data repository. Implementing such a custom data repository requires a large investment of time and money. This is a huge hurdle to take before such a research project can even begin to gather data into the repository. Experience from the MUSE and MUSE4Music projects also shows, that the ontology of such a project will likely be revised during the beginning of the data gathering. This also means that the custom data repository likely needs continuous adjustments in the beginning.

To solve this problem, a generic data repository that can be used with a custom ontology is needed. It has to be adaptable to different domains and their respective ontologies and must allow the user to input the ontology into the tool, without requiring programming knowledge. For this the repository should include a basic editing interface for the ontology and the embedded taxonomies.

This would also solve another problem that both projects have encountered. The ontology for MUSE and MUSE4Music was specified using mind-map tools. Mind-maps are easy to understand and tools to create them are readily available. Whenever these mind-maps change (or the repository is updated) the changes also have to be applied in the repository (or the mind-maps). This always involved manual comparison of the mind-maps with the source code or the database schema of the repository. The manual comparison cannot be automated, is slow, and often some minor mistakes are overlooked. Integrating the ontology management into the generic data repository would render this comparison unnecessary.

This master thesis is about implementing such a generic data repository for digital humanities projects following the MUSE methodology. This generic MUSE4Anything repository is flexible enough to support a variety of use cases from different domains, such as the two already mentioned use cases of film costumes and historical music.

Structure of the Thesis

The thesis begins with the discussion of the use cases and their requirements for a MUSE4Anything repository in Chapter 2. Following is Chapter 3, a short survey of existing tools relevant for the identified use cases and requirements.

Chapter 4 is about the implementation of the MUSE4Anything repository. It starts with an architecture overview of the finished repository. Then the database, the HTTP API based on the REST principles, and the user facing front-end are described in more detail.

The implementation is evaluated in Chapter 5. A small example use case is used as a tool for demonstrating the capabilities of the MUSE4Anything repository. The use case is chosen to allow easy comparisons the existing MUSE4Music repository. Finally, Chapter 6 contains a brief summary of the thesis.

2 Requirements for MUSE4Anything

This chapter lays the groundwork for the implementation of the MUSE4Anything repository. Based on experience with the MUSE and MUSE4Music projects and how their data repositories are used in their context, use cases for the generic MUSE4Anything repository are identified. Requirements are then specified for the identified use cases. Relevant tool categories are also explored for each use case.

MUSE4Anything is a project without a direct customer to voice what his requirements are for this tool. The requirements have to be based on justifiable and objectively verifiable arguments. A good baseline for requirements of the MUSE4Anything repository are requirements from the MUSE and MUSE4Music repositories. To understand these requirements, one must first understand how these data repositories are used in their project context.

Figure 2.1 shows how the MUSE and MUSE4Music repositories were used in their project. The top of the figure shows the ontology with the embedded taxonomies, that are separately maintained using mind-map tools. A software developer is needed to implement the data repository below, based on the ontology. If that ontology changes, the software developer needs to implement these changes in the repository as well. If only the content of a taxonomy has changed, then the researcher maintaining the ontology can use the built-in taxonomy editor interface, to update the taxonomy directly in the tool.

The data repository is split into three views in the figure. The *Taxonomy* view is for viewing and editing the content of taxonomies. In the MUSE4Music repository all taxonomies can be edited while in the MUSE repository only some taxonomies are editable. The *Data Entry* view is for entering the data into the repository. The drop-down fields shown in the figure are representing the form inputs, that are used to record the individual attributes. At the bottom of the figure is the *Data Visualization* view. The MUSE4Music repository has a mode for viewing the data, that removes most of the visual elements needed for data input like the validity indicators. With less distracting elements the actual data is easier to focus on. The MUSE repository has a special costume graph displaying the relationships of the basic elements of a costume.

The *Data Visualization* view is especially important in the analysis phase of the projects. Pattern candidates from the computer assisted analysis methods need to be verified by someone with extensive domain knowledge, who can judge whether a pattern candidate is in fact indicative of an actual pattern. For this the researcher often needs to look at the data again to confirm a pattern.

To make a generic MUSE4Anything repository, the ontology and taxonomies cannot be hardwired into the data repository. This includes the database schema of the repository. Adapting the repository to another ontology should also not require special knowledge or training, to not create another kind of developer role. This means that the MUSE4Anything repository must feature an easy-to-use

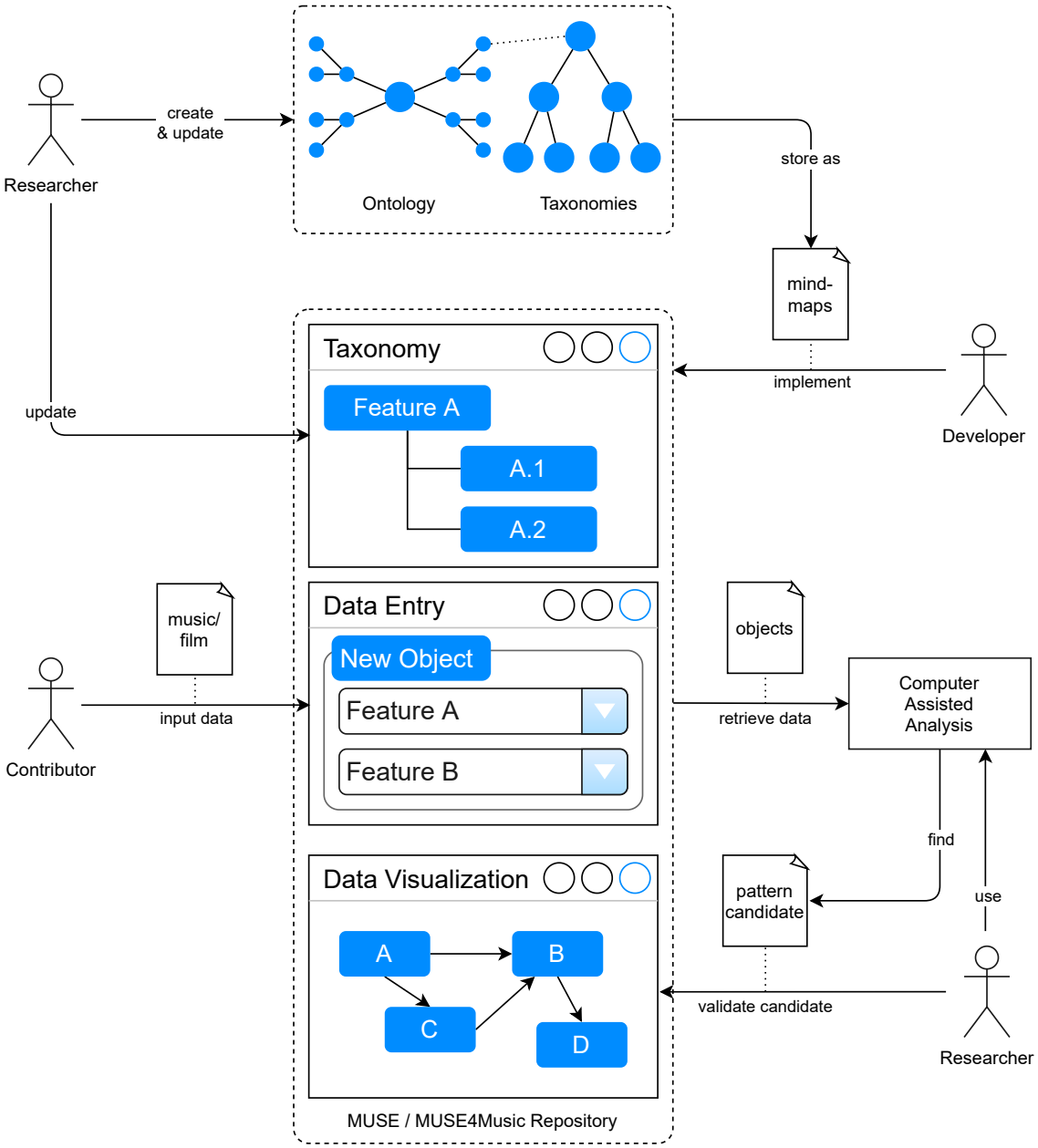

Figure 2.1: The MUSE and MUSE4Music repository as used in their project context. The figure shows how different roles interact with the data repository. The three screens in the central column represent different views, that are tailored to the different use cases. The ontology and the embedded taxonomies above are a separate resource that is managed separately from data repository. The developer is a special role in that he does not interact with the data repository, but implements and updates the repository based on the ontology.

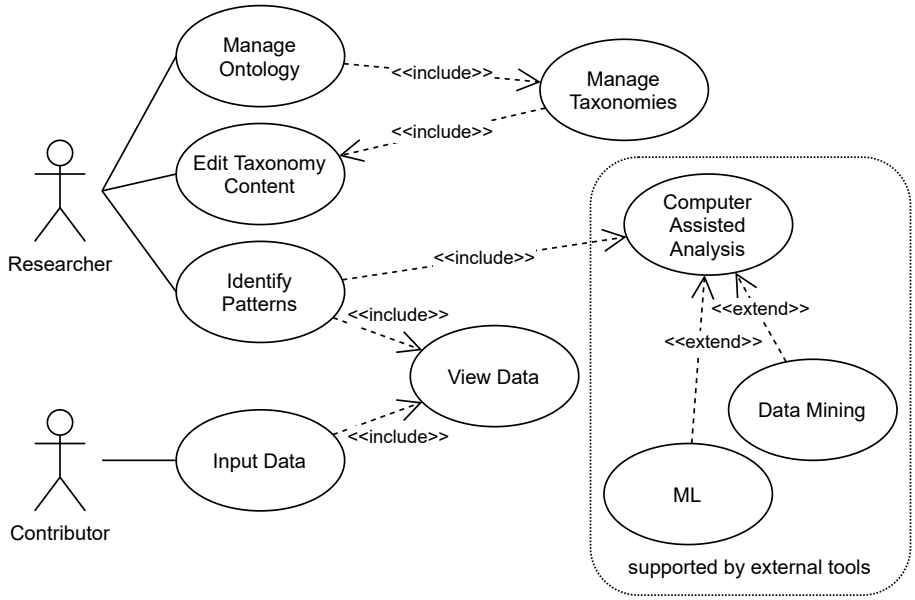

Figure 2.2: The generalized use cases of the MUSE and MUSE4Music repositories.

editor for the ontology and the embedded taxonomies. As a positive side effect, integrating the ontology management into the data repository makes the repository the single source of truth for the ontology. At best, all manual comparison of mind-maps with the repository can be avoided.

With that knowledge, the use cases the MUSE4Anything repository needs to support can be formalized. The use cases are shown in Figure 2.2. They are split between the two roles of *researcher* and *contributor*. The contributor is shown as a separate role, as usually many people are needed to input the data into the repository. Additionally, contributors usually don’t take part in the other use cases. The use cases for the *computer assisted analysis* are separated, as they rely on external tools. The following sub-sections describe the four main use cases, as well as their requirements.

2.1 Use Case “Input Data”

The most commonly utilized use case of the MUSE4Anything repository will be *data input*. A small amount of data could still be analysed manually. But the computer assisted analysis techniques e. g., data mining and machine learning, can handle much more data at once. Since the goal of both MUSE projects is to let the computer help with the analysis, they need a large amount of data.

Inputting that data will at first always be a manual process, as automatically recognizing certain attributes of an object and matching them to the correct taxonomy element would require specialized tools. This means that many people will be needed for gathering the large amount of data and entering it into the data repository. It is also possible, that some part of the data input does not need specific domain knowledge. This would allow the use of untrained workers or even crowdfunded work for this use case. To accommodate this, the data repository should not require much, if any, briefing before anyone can use it to input new data.

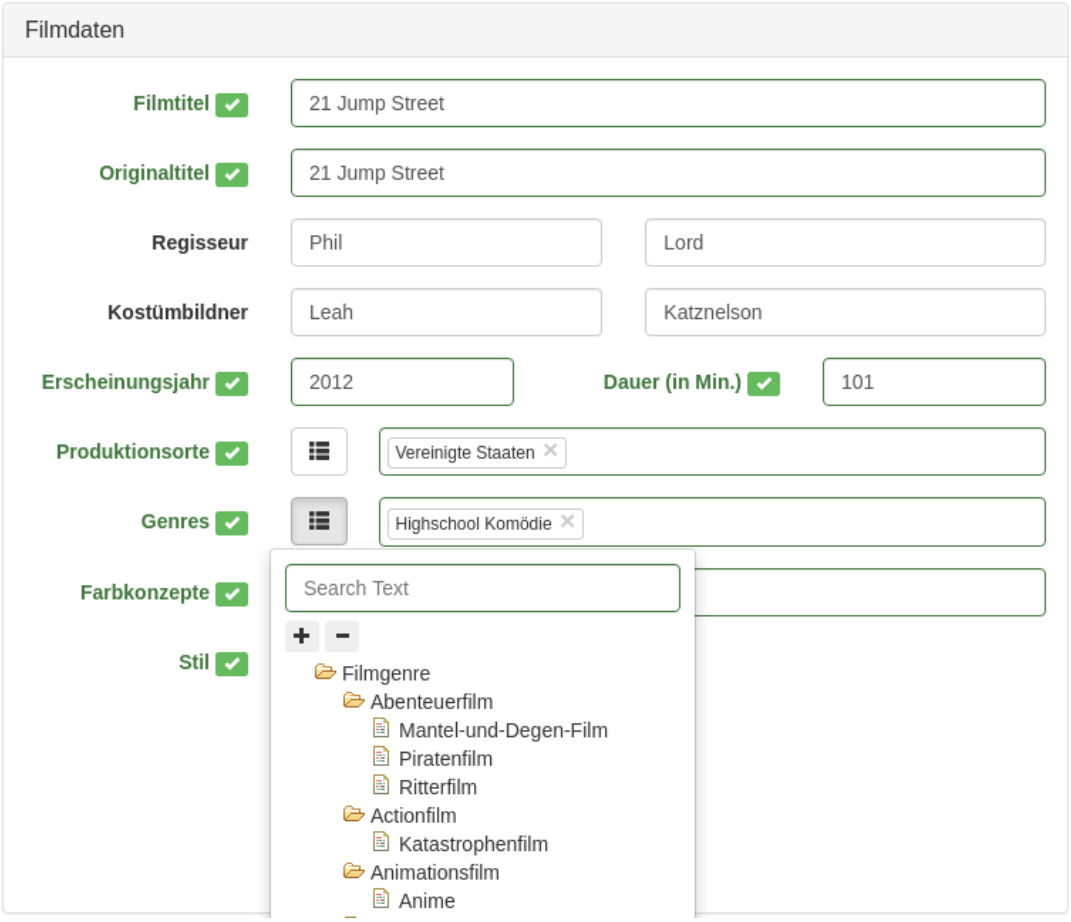

Figure 2.3: Data input form in MUSE with an open taxonomy select element. The taxonomy can be filtered with the search field above.

Data input is a manual and repetitive work that always follows the same schema, the ontology. The repetitive nature makes mistakes more likely to happen. The data repository should therefore be designed to minimize the possibility of mistakes. When they do happen, they should not cause undesirable things like loss of data. The MUSE and MUSE4Music repositories already prevent or minimize some common mistakes. Using taxonomies to capture most attributes already eliminates spelling mistakes and simplifies consensus, by limiting the possible values that apply for an attribute. Deciding between two existing entries of a taxonomy is easier, than answering an unconstrained question. To help the user find the right taxonomy entry, both existing repositories provide a searchable tree view for each input field based on a taxonomy. The taxonomy select element of the MUSE repository can be seen in Figure 2.3 in the context of the input form of a film object.

Another way to reduce the chance for mistakes, is to ask for all possible attributes of an object. While not every attribute can always be filled in, it is better to have the whole list of attributes present when entering the data. This allows the user to check, if he has forgotten one of the attributes, simply by looking at the attributes he has not yet filled out. Combining this with visual indicators that highlight empty fields or fields with wrong inputs, further minimizes the potential for such errors.

Figure 2.4: The data input form for an opus in the MUSE4Music repository.

Especially the MUSE4Music repository has objects with many attributes that do not fit on one screen, as can be seen in Figure 2.4. Descriptions for the attributes help the user to remember how to fill that attribute out. The MUSE4Music repository also allows for taxonomy items to have a description. This description is also displayed in the taxonomy input fields. It is particularly useful for items that are rarely used.

Another way to make data input easier, is to make the tool itself easy to use. Both the MUSE and the MUSE4Music repository can be used with any up to date web browser. No installation is necessary to use the repository. This also means, that the tool can be used with many devices and even remotely.

To use the ontologies from MUSE and MUSE4Music the data input needs to support at least taxonomies, true-/false, numbers, text, complex attributes in the form of nested objects, lists and references to other objects. Objects being a mapping from an attribute to its value. Many other data types can use a text input e. g., dates, times and time-spans. However, using a text input for these data types may lead to suboptimal error checking. Lists and objects must also be able to contain other lists and objects. The data repository must also be able to handle many objects. MUSE alone already counts over 5000 costumes with a total of over 28000 basic elements.

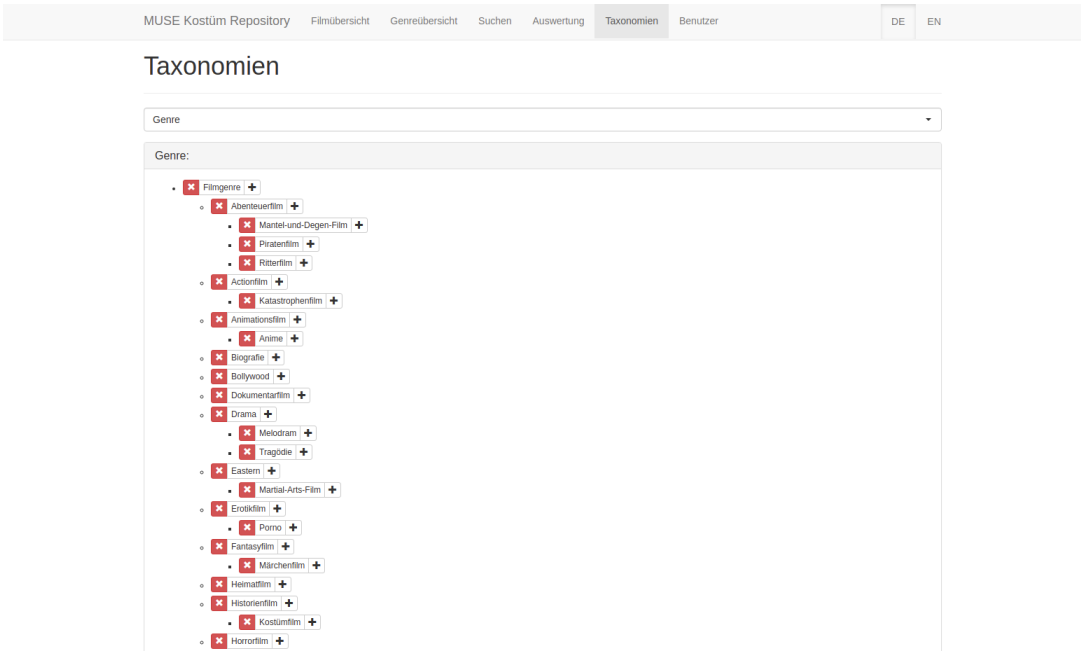

Figure 2.5: The taxonomy editor of the MUSE repository.

Requirements of the use case "Input Data"

Requirements using "*must*" must be met. Requirements using "*should*" may be discarded with good reasons, for example a very complex implementation that is not in proportion to the expected benefit. Requirements using "*may*" are completely optional.

- The MUSE4Anything repository *must* be usable without installation of a custom software.
- The MUSE4Anything repository *must* support basic data types like strings, numbers, boolean values, references to taxonomy items and references to other objects.
- The MUSE4Anything repository *must* support complex data types including nested objects and lists.
- The MUSE4Anything repository *must* show errors to the user as soon as possible and reject malformed data.
- The MUSE4Anything repository *must* always use the current version of the user definable ontology for data input.
- The MUSE4Anything repository *must* be able to handle many objects.
- The MUSE4Anything repository *should* make it hard to lose data on accident. Deleting or overwriting data (on accident) should not lead to data loss immediately.

2.2 Use Case “Edit Taxonomy Content”

The taxonomies are the part of the MUSE and MUSE4Music ontologies that change most often. Most of these changes are a new item, that was added to the taxonomy, or one that was removed, because it was not used. In the MUSE4Music repository all taxonomy items also have a description that is even more likely to change, if the original description was lacking in some sense. To accommodate this, both repositories already include an editor for the content of existing taxonomies. Though the MUSE repository (pictured in Figure 2.5) only allows editing for some taxonomies.

An item of a taxonomy has a name, a short title, and a description. The description is used as a hint for the user in the data input use case. The items are usually organized in a tree structure, but can also form a flat list if no hierarchy can be established. The sorting of items should be controllable by the user, as alphabetical order is often not the most intuitive order.

In both projects, mind-maps were used to model the taxonomies. This means that the editor for taxonomies has to compete against mind-map tools. They are easy to use and provide a graphical visualization of the tree structure of a taxonomy. The taxonomy editor should provide a similar visualization for editing the tree structure and the taxonomy items.

In the MUSE4Music project some taxonomies had certain items multiple times. The tree structure of the taxonomies does not allow for the same item to be used in multiple branches. To allow this, taxonomies would need to have a graph structure. The next step from a tree to a graph would be a directed acyclic graph. Preventing cycles in the graph helps to simplify the implementation of the required user interface components.

Requirements of the use case “Edit Taxonomy Content”

- The MUSE4Anything repository *must* be usable without installation of a custom software.
- The MUSE4Anything repository *must* provide an easy-to-use interface to edit the taxonomy content. The interface should not be more complex than a normal mind-map editor.
- The MUSE4Anything repository *must* support taxonomies where the items form a flat list or a tree structure.
- The MUSE4Anything repository *may* support taxonomies where the items form an acyclic directed graph.
- The MUSE4Anything repository taxonomy items *must* have at least a name and a description property.
- The MUSE4Anything repository *must* allow objects that reference old or deleted taxonomy items to exist.
- The MUSE4Anything repository *may* record changes to the taxonomy structure and taxonomy items in a edit history or use other means of versioning the taxonomy.

2.3 Use Case “Manage Ontology”

The MUSE and MUSE4Music repositories do not have a feature that allows editing the complete ontology. The MUSE4Music repository already experimented with UI and form generation from the OpenAPI 2 specification of the back-end API. Some of the experience gathered there, can also help inform the requirements for a completely generic user interface and what the back-end would need to support that.

The ontology defines how the data looks like, by describing objects, their attributes, and the possible values of these attributes. For the MUSE4Anything repository to fit any ontology, it cannot hardwire the ontology into any part of the program including the database. The ontology must be stored as data that can be changed. While it may be easier to restrict changes in the ontology to only be possible before the repository is used to input the actual data, this would have several negative consequences. Changing the ontology either becomes impossible or any data already in the repository cannot easily be migrated when the ontology is changed. Given the experience with MUSE4Music, where the ontology was revised multiple times during the development of the prototype implementation of the repository, it is likely for an ontology to change. A generic data repository should actively support such changes or at least make them as easy as reasonably possible.

Managing the ontology also includes managing the taxonomies that are part of the ontology. This includes adding or removing taxonomies. Editing the content of the taxonomies is of course also included in this use case.

Because the ontology cannot be known beforehand, large parts of the user interface and also parts of the back-end and database need to be generated from the ontology data. In the MUSE4Music repository the UI generation was done based on the OpenAPI specification, which is very similar to JSON Schema. JSON Schema is also a relatively easy to understand specification and powerful enough to model all required input types described for the input data use case. Therefore, JSON Schema or a similar standard would be a good candidate for how to represent and store the ontology data.

Similar to taxonomy items, the attributes in the ontology also should be sortable by the user editing the ontology. This sorting should be used in the data input interface. A description for the attributes, that is then accessible in the data input user interface, would also be beneficial.

A mind-map like editor, like for the taxonomy content, is not strictly needed. As long as the user interface for editing the ontology has enough documentation, one can expect the few researchers who are actively developing the ontology to invest the time to learn how to use the tool. However, this requirement needs to be revisited, once the MUSE4Anything repository can be tested in practical settings.

Requirements of the use case “Manage Ontology”

- The MUSE4Anything repository *should* be usable without installation of a custom software.
- The MUSE4Anything repository *should* provide an easy-to-use interface to edit the ontology.
- The MUSE4Anything repository *must* support defining all data types mentioned in the requirements of the input data use case.

- The MUSE4Anything repository *should* allow the user to influence how the data input form is generated. This includes the order of the input fields, their labels, and optionally their descriptions.
- The MUSE4Anything repository *may* allow the user to reuse or extend existing data type definitions.
- The MUSE4Anything repository *must* retain all versions of a data type definition, to allow for existing objects to reference the version they were created with.

2.4 Use Case “Identify Patterns”

Identifying patterns is the final use case of the MUSE4Anything repository. It includes the *Computer Assisted Analysis* use case. To be able to use techniques like data mining and machine learning, the data has to be accessible in a format that is easy to use for these techniques. For the MUSE repository this was solved by allowing direct access to the database of the repository for the custom-built analysis tools. Exporting the data in a standardized format like JSON or XML could also be a solution here. However, in the case of the MUSE4Anything repository the ontology also has the same requirements to be accessible for the computer assisted methods. This allows for such tools to be built in a generic way and to support any ontology MUSE4Anything can support.

To verify the pattern candidates that are identified with the computer assisted analysis, the data repository should have dedicated views for viewing the data. All extra visual elements that are helpful for the data input, can be distracting when only viewing the data is the goal. At the minimum, displaying the data without all form elements is a requirement of the MUSE4Anything repository. Domain specific visualizations like the costume graph in the MUSE repository depicted in Figure 2.6 cannot be done in a completely generic way. However, a generic object graph to view object to object relationships may be possible.

Requirements of the use case “Identify Patterns”

- The MUSE4Anything repository *must* provide a view explicitly for viewing the objects that has minimal visual clutter.
- The MUSE4Anything repository *may* provide a graph view to visualize object references.
- The MUSE4Anything repository *may* provide a view to compare two or more objects.
- The MUSE4Anything repository *must* provide machine-readable data in a standardized format like JSON via the API or direct database access for use with external tools.
- The MUSE4Anything repository *must* provide the ontology in a machine-readable and standardized format like JSON via the API or direct database access for use with external tools.

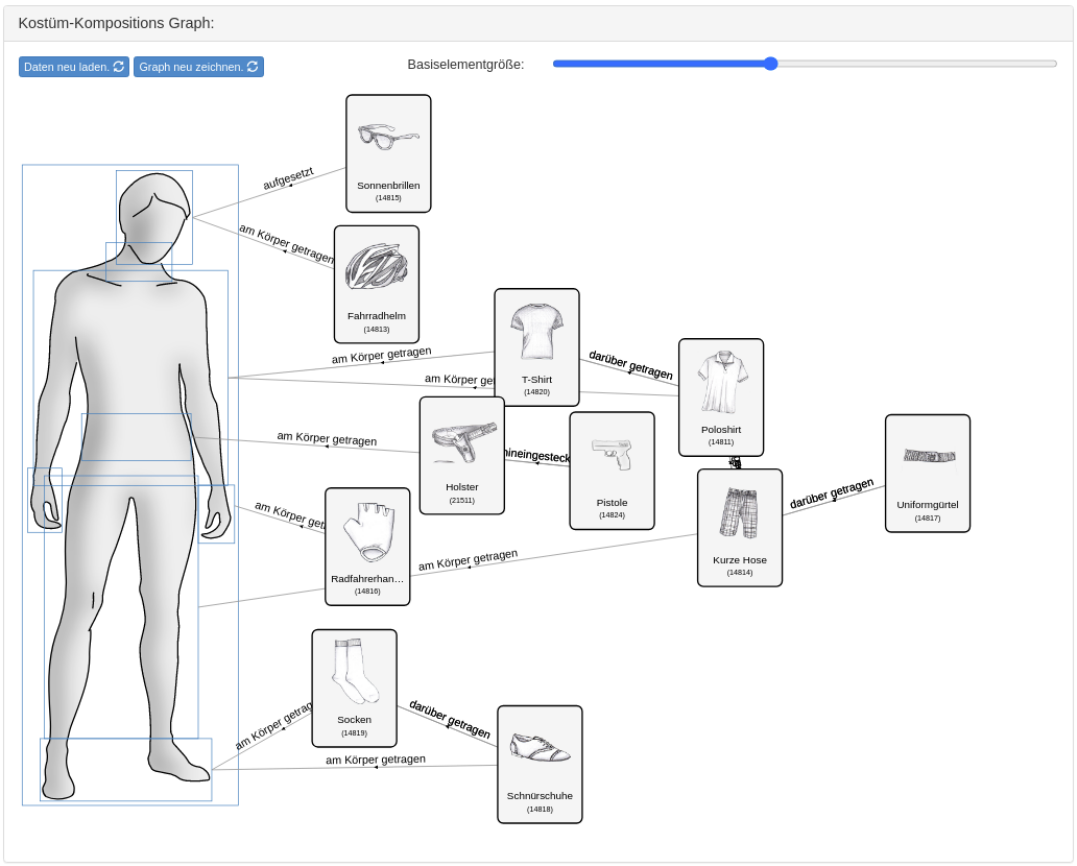

Figure 2.6: The costume graph of the MUSE repository. The graph shows the basic elements of a costume, their relations to each other, and where on the body they are worn. Layouting is performed with a custom force directed layout algorithm.

3 Related Work

This chapter is about related work in form of already existing tools and software libraries. With the use cases of the MUSE4Anything repository known, one can search for existing tools that have similar use cases. They can be inspiration for the actual implementation or, if some conditions are fulfilled, they could even be used as part of the MUSE4Anything repository.

To find relevant tools without relying on chance, the search space has to be narrowed down. With the use cases and their requirements known, the type of tools useful for the use case can be inferred. For the first mentioned use case *input data* UI or form generators would be most useful. They can take a description of how the data should be presented to the user and generate the graphical user interface from that. Form generators build a data input form, where the user can enter the data, from such a description.

For the *edit taxonomy content* use case, mind-map tools are a good start. Also tools that are meant specifically for taxonomies or ontologies should be included here. These tools are also relevant for the *manage ontology* use case. Additionally, database schema generators, that take a schema definition in a format like JSON Schema and generate the database schema, may also be relevant for this use case, depending on the ontology format.

The last use case to *identify patterns* contains two very different use cases. The *view data* use case has similar requirements as the input data use case. UI generators are relevant tools for both these use cases. The *computer assisted analysis* use case is still part of active research in the MUSE project. It is also not directly part of the data repository and relies on external tools that only need access to the data. Therefore, no tools for this use case will be included in this thesis.

3.1 Existing Ontology and Taxonomy Tools

Because the MUSE4Anything repository has to provide basic editing functionality for taxonomies and ontologies, only tools that can do both are included in this search. The method for searching was a normal internet search using DuckDuckGo [Duc21] as the search engine set to English. The default language for search results would have been German, which could have excluded tools only available in English. DuckDuckGo was chosen because it is a privacy-friendly search engine, that does not customize the search results for every user. This should make the search more repeatable, as previous searches have no influence on the current search. Only the first page of results was considered, excluding all ads. In some cases links were followed, if they were likely linking to generic ontology management tools. The search queries used were “ontology tool” and “owl ontology tool”. Raw search results can be found in Appendix A.

The following paragraphs are about the ontology tools found with that search. For brevity only tools that had an update in the last years or are otherwise relevant e. g., because of a unique feature are included in this summary. Also tools that are commercial applications and do not provide

much information are not be included. If a tool is actively maintained and open source, a more detailed analysis of whether and how the tool could be used for the MUSE4Anything repository is performed.

Protégé [21h; Mus15] is an open source ontology editor developed at the Stanford University School of Medicine. It is written in Java and there exist a desktop and a web version that can be used in the browser. The desktop version can be extended using plug-ins. Plug-ins can contribute new views, layouts, settings, and more to augment and customize the Protégé tool. However, most of these plug-in types are likely incompatible with the web version of Protégé indicated by the absence of documentation for plug-ins for WebProtégé.

Protégé uses OWL [W3C12] for ontologies. Its main focus is on developing the ontology and allowing automatic reasoning based on the ontology. Data instances, called individuals in OWL, are rarely even mentioned in the user documentation. This is completely different from the MUSE projects where the data is the main focus.

Protégé also does not have similarly strict input checking for data as the MUSE and MUSE4Music repositories do. It does have autocompletion but still allows users to define arbitrary attributes. Nested objects must be defined as two individuals that have a relation. The same hold for lists. Together this makes the most often used use case of the MUSE data repositories cumbersome and likely to introduce errors. It may be possible to add a view or even a custom editor, that provides the same input checking and nested object editing as the MUSE repositories, but this would not be able to entirely replace the Protégé UI. With the original Protégé UI still accessible by default, the tool would need more training for the users. It is also unclear if such a plug-in is even possible for WebProtégé, which would mean that every user first has to install the desktop version locally including the installation for the plug-in.

The ontologies of MUSE and MUSE4Music use only a fraction of what OWL is able to model. While some learning curve is acceptable for the ontology management, the editor in Protégé is more powerful and also more difficult to learn than a generic mind-map tool. Making the editor easier to use and restricting it to only allow what is necessary for a MUSE ontology, would run into the same problem as for the data input use case.

Adapting Protégé to fit the use cases and requirements of the MUSE4Anything repository is a big task. The main benefit would be that all visualization tools and the OWL reasoning of Protégé could also be used for free. Given that the OWL reasoning is not useful for the purpose of extracting unknown patterns from a large amount of data, only the visualizations remain as a benefit. A drawback would be the huge amount of complexity that comes from supporting OWL for ontologies. This complexity is built into Protégé and cannot easily be hidden for a user even with plug-ins. Using Protégé plug-ins would also likely mean that the MUSE4Anything repository must be installed locally and cannot be used with a web browser.

The visualizations Protégé renders for ontologies or that are provided by plug-ins can be a good resource, when building similar visualizations for the MUSE4Anything repository. The graphical syntax, layout, and interactivity are all elements that can be studied as examples. If some graphical notation is the standard (or de-facto standard) for visualizing a certain concept, then this notation should also be used in the MUSE4Anything repository if there are no good reasons against doing so.

WebVOWL [LLM+19] is a visualization tool for OWL ontologies. It uses VOWL, the visual notation for OWL ontologies [LNHE16], as the graphical notation. An ontology can be loaded into the tool by providing an IRI. It also includes basic editing functions and can export the edited ontology. It can also export the visualization in the SVG format. This tool is particularly interesting because all that is needed to integrate it as an external visualization provider, is an API endpoint that exposes the ontology in the OWL format.

OWLGrEd [21g] is another graphical editor for OWL ontologies. Instead of VOWL it uses a graphical syntax inspired by UML class diagrams [BBČ+10]. It even has an extension that provides more graphical syntax from UML diagrams like composition relations and abstract classes. There also is an online version of the tool that can only visualize ontologies. The online version has a size limitation for ontologies and they can only be provided by a file upload.

Fluent Editor [21c] is a commercial product that is free for academic purposes. It has a custom visualization using a custom graphical notation. The tool is also interoperable with Protégé using a plug-in for Protégé. The unique feature of this tool is the ability to use what they call *Controlled English* as modelling language for ontologies. Using something close to natural language can be a huge benefit, as no special syntax must be learned and anyone should be able to read and understand the language with little to no learning. Although it is a huge advantage, this advantage can only be leveraged for English speakers. Both MUSE and MUSE4Music are German research projects where this advantage cannot be used.

Ontopia [21f] is a collection of tools for building Topic Maps [Pep+00] based applications. Topic Maps originated as a means to merge different indices and were generalized. They can also be used to model ontologies. The latest release is from 2015, but there is some activity on the GitHub repository. Ontopia also includes a visualization component. However, the documentation available does not show how the tool can be used to create ontologies or taxonomies (or equivalents).

Eddy [21b; LPSS16] is an open source ontology editor using the Graphol language. Graphol has a unique and expressive graphical syntax to represent the different ontology concepts. The ontologies can be exported in the OWL 2 format to be used with other tools like Protégé. Import of such ontologies is listed as a feature soon to come on the website. Eddy is also a tool that needs to be installed locally and is not able to communicate with a central server. Using Eddy would introduce a new source of possible inconsistencies when the ontology is edited by multiple people. The Graphol language however is also a possible graphical notation, that could be used for the ontology visualization in the MUSE4Anything repository.

VocBench [21j; SFT+20] is an open source ontology editor with a custom license. It has a client server architecture and the web front-end is implemented with Angular. It has multiple visualizations for an ontology, like a class diagram based one and a VOWL like visualization. The ontologies are based on OWL 2. Versioning and role based access control are also supported. It even allows users to create custom forms for data input. To use this feature the user has to program the custom

form using special rules expressed in the custom language PEARL, not to be confused with the programming language Perl. This is definitely to high of a bar for the users expected to be using the MUSE4Anything repository.

VocBench exposes the complexity of OWL ontologies to the user. This means that learning to create ontologies in this tool will take time and effort. Protégé also has this problem. For data entry the custom forms from VocBench can be used to reduce the complexity to a minimum. This could make it a viable solution for a MUSE4Anything repository. However, defining these custom forms is a very advanced feature of VocBench. To make it more easy to use would require generating the custom form descriptions with a user-friendly interface. As VocBench does not have a plug-in system for UI components, this would require forking the project. The results could later be submitted to the VocBench project. However, the custom licence of the source code is a blocking issue for this.

TemaTres [Fer21] is a tool to manage vocabularies and taxonomies. It also lists ontologies on the landing page, but the manual does not mention any feature that would support this. A main feature of this tool is that the vocabularies can be multilingual. Vocabularies can also be shared with other TemaTres instances, allowing references to other external vocabularies. There are also some plug-ins, one of them providing a visualization for the vocabulary. It could theoretically be used to manage the taxonomies of the MUSE4Anything repository as it provides an API to access the stored data and an RSS feed for changes. This would have the drawback of maintaining two tools. Supporting links to external vocabularies could also prove to be difficult.

DogmaModeler [21a] is an ontology modelling tool that uses “Object Role Modeling” [Hal20] as the graphical notation. The last release is from 2013 so the tool would not be included in the summary, if not for the different choice in graphical notation. “Object Role Modeling” is somewhat similar to Entity-relationship models. This could also serve as an inspiration for the graphical syntax of the MUSE4Anything ontology visualization.

Hozo – Ontology Editor is another ontology editor. It features a custom graphical notation for visualizing the ontology. Hozo hides some complexity that OWL allows from the user, by using higher level concepts closer to human understanding. The manual only mentions exporting as OWL and not as OWL 2, but this could already be implemented because the last tool version is from 2020 while the manual is from 2014. However, there is no source code available for this tool.

Database Schema Generators

Database schema generators can be used to generate the database schema from an ontology description, provided that the generator supports the description format. The MUSE4Anything repository could use such a generator to automatically build the database schema from the provided ontology. For this the usage of the repository has to be split into two phases. The first phase for editing the ontology only and the second phase for entering the data. The transition point between these phases would be the database schema generation. Because the database schema cannot easily be changed afterwards, this would most likely be a one time only process.

The experience with MUSE4Music shows, that the ontology is likely to change once it is in use. The MUSE project was similar in the beginning. Some things that the ontology developer thought to be important turn out to be used only once or twice, while other things that should have been in the ontology from the beginning are only found when they are encountered during data input. Ideally the MUSE4Anything repository would allow the necessary changes to the ontology even after the data input began. Using a fixed database schema, even one generated from a user defined ontology, would not allow this or at least complicate this use case. For this reason no actual database schema generators were evaluated for this project.

3.2 UI and Form Generators

Automatically generating form inputs from the ontology data is a necessary feature for the MUSE4Anything repository UI. The MUSE4Music repository already features a custom-built automatic form generator that uses the OpenAPI [OAI18] description of the rest API. Because the MUSE4Anything repository should also be accessible from a web browser the form generator should be built for the web. The search was conducted with the search engine of the npm repository [npm21]. Only the first page of search results was included. Only libraries that could generate a full form from a machine-readable description format (e. g., JSON Schema) are listed in this summary. Libraries without any documentation or libraries where the documentation did not show relevant examples were also excluded, because they cannot be evaluated without using the library. The search term used was “form generator”. Raw search results can be found in Appendix A.

JSON Forms [21d] is a form generator with support for the frameworks React, Angular, and Vue. The form styling can be controlled and even custom renderers for form elements can be supplied. Forms can be styled in Material design and the Bootstrap design system. It also supports nested objects. The generator can also follow references to other JSON Schemas but this only works out of the box with React.

vue-jsonschema-form [21k] is a form generator for the Vue framework. It is currently marked as work in progress. Like JSON Forms it supports custom renderers and custom layouts. It also supports nested objects. However, it is not mentioned whether references to other JSON Schemas are supported.

UI Schema for React [21i] is a UI and form generator for the React framework. Widgets for Material design and the Bootstrap design system are provided with the generator. It supports nested objects and custom renderers. Validation is only supported for some properties from JSON Schema. This form generator can load referenced JSON Schemas asynchronously.

3.3 Mind-Map Tools

In both the MUSE and MUSE4Music project mind-maps were used to model and maintain the ontology and taxonomies. However, mind-map tools do not provide graphical notations that are specific to ontology development. The MUSE4Music ontology for example was authored with the Freeplane [Fre20] tool and a custom graphical notation that could be realized in that program. Other typical mind-map programs like MindMup [21e] or mindmaps [Ric18] only offer basic styling options for nodes. They do not allow for a graphical syntax that is as expressive as the visualizations of the ontology tools. The layout algorithms of mind-map tools can also be used as inspiration when designing a layout algorithm for the ontology visualization of the MUSE4Anything repository. However, ontologies typically have more interconnections than a mind-map allows which could make mind-map like layouts impractical.

4 Implementation

This chapter is about the implementation of the MUSE4Anything repository. The core architecture is modelled after the MUSE4Music repository [Büh17]. Design decisions, that were already discussed for the MUSE4Music repository and are still valid for the MUSE4Anything repository, will not be discussed in detail again. Instead, differences in the design needed to support generic ontologies compared to the MUSE4Music repository and tooling changes are the focus of this chapter.

The high level architecture of the MUSE4Anything repository the same as the MUSE4Music repository architecture. The component diagram in Figure 4.1 shows the main parts of the MUSE4Anything repository. It is built with a client server architecture. The front-end communicates with the back-end server via an HTTP API. The back-end server itself is stateless and stores all data in an SQL database.

The tooling for the back-end server did not change much compared to the MUSE4Music repository back-end. Both use Python as the programming language and Flask [Ron] as the base framework. Database access is implemented using the SQLAlchemy ORM [SQL]. To be able to use the OpenAPI 3 standard [OAI18] for documenting the HTTP API the flask-smorest [Nc20] library was used. The MUSE4Music repsoitory used a different library that could only generate OpenAPI 2 specifications.

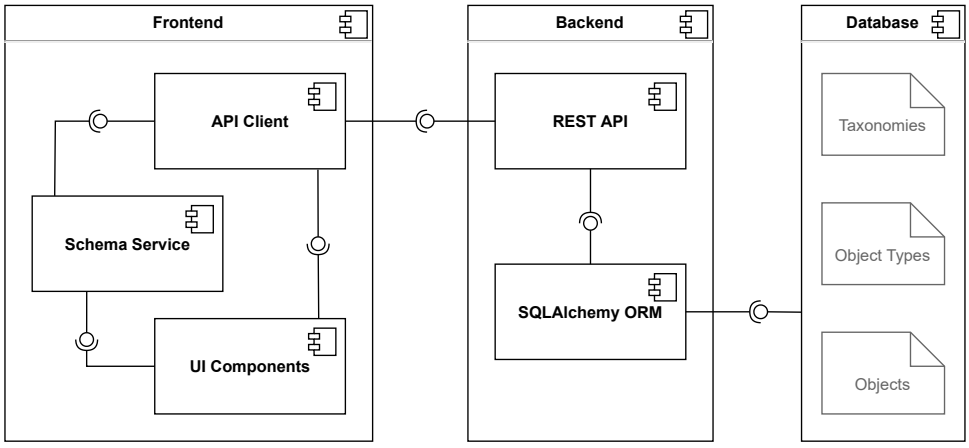

Figure 4.1: The architecture of the MUSE4Anything repository. The front-end is written in TypeScript as a single page application and uses the Aurelia framework. The back-end is written in Python and uses the Flask framework together with the SQLAlchemy ORM for the database handling and flask-smorest to implement the HTTP API. The back-end can be used with a SQLite, MySQL or MariaDB database.

The MUSE4Music repository front-end was implemented as a single page application using the Angular framework [Goo21]. A single page application avoids full page reloads by only loading and replacing the changed page content using JavaScript. This usually makes for faster page changes on user input. It is a useful technique to implement interactive applications as webpages. The main problem with Angular is that it frequently updates and updates usually include major breaking changes. The implementation of the MUSE4Music repository started with Angular 4 and the current version of Angular is 11. This makes the MUSE4Music repository hard to maintain. For the MUSE4Anything repository the Aurelia framework [Blu20] was chosen because it has a very stable API and the developers try to keep the API stable with updates.

Implementing a completely new data repository is an ambitious project for one developer.Because of the requirement to support user definable ontologies, many features had to be designed and implemented first to test if they work. The main priority was to deliver a fully working prototype of the MUSE4Anything repository that can later be upgraded to a polished product. The MUSE4Music back-end implementation used the same approach. This means that there are currently no automatic tests for the functionality of the MUSE4Anything repository. All test were performed manually with the user interface that was developed in parallel to the HTTP API. Because most of the key features like UI and form generation reuse the same functions, manual testing over time can have a decent coverage. While a full, comprehensive test suite was very much impossible given the time constraints of the thesis, all code was written in a way to facilitate adding tests at a later point in time. The code also contains the groundwork for internationalization of the API and user interface. The basic framework for user authentication is also already part of the code but unused as of now.

The code of the MUSE4Anything repository is open source and can be found on GitHub[1]. The link to the documentation on "Read the Docs" can be found in the readme and the repository description. Currently the documentation only covers the development setup, the project structure, and the custom format for API responses. The documentation also contains the current API documentation in the OpenAPI 3 format as a downloadable `api.json` file.

The following sections go into details about the implementation of the MUSE4Anything repository. The back-end implementation is split between database and API. The third section is about the user interface implementation.

4.1 Database

A key difference between the MUSE4Music repository and the MUSE4Anything repository is, that the ontology is not known at development time for the MUSE4Anything repository. For MUSE4Music the data was stored directly into an SQL database and each attribute had its own row in a table. However, tables and rows need to be specified beforehand and cannot be changed arbitrarily later. This makes the approach from the MUSE4Music repository impractical for MUSE4Anything.

The MUSE4Anything repository should also minimize the impact of a data input error. A great way of doing that would be an undo operation. This could be implemented on the client side, but with the user interface running in the browser it is hard to implement an undo feature that stays persistent

[1] https://github.com/Muster-Suchen-und-Erkennen/muse-for-anything

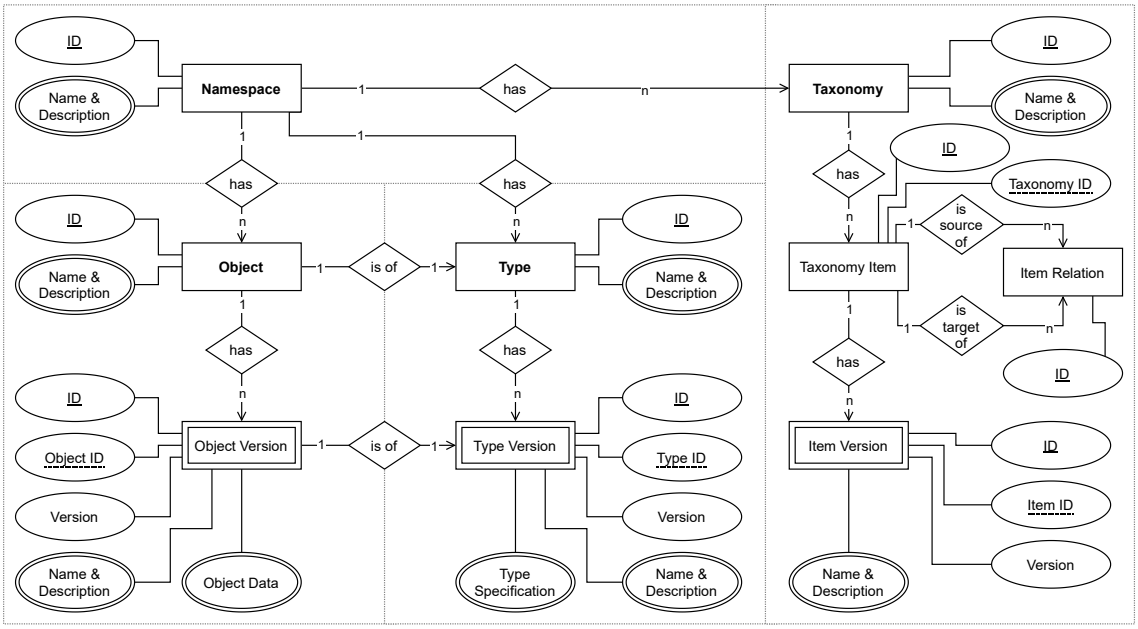

Figure 4.2: The data structure of the MUSE4Anything repository as an Entity-Relationship model. The diagram is partitioned into four sections with the main entity highlighted with **bold** text. Some attributes and relations were excluded from this diagram in favour of readability.

over multiple logins and devices. Another way is to version the data on the server side. When updating an object, a new version is created instead of the current data being overwritten by the update. This prevents data loss because of input errors, as all existing versions stay untouched.

If the data is versioned, then the taxonomies and the ontology definition should be versioned as well. The experience with MUSE and MUSE4Music show that both the taxonomies and the ontology containing the taxonomies will likely change over time, especially in the starting phase of the project. So if an old version of an object is modelled after an old version of the ontology and references old taxonomy elements, then all these should still exist in the database. Otherwise, the old version of the object cannot be correctly reconstructed as some parts may be missing.

A characteristic of the MUSE and MUSE4Music data is that an object can have many attributes. The attributes can have complex values that need to be represented as objects or even lists of objects at the data level. This means that the attributes of an object cannot be represented by a single flat key value data structure. Objects can also reference other objects.

Figure 4.2 shows the data structure of the MUSE4Anything repository. A *namespace* is used to group *taxonomies*, *types*, and *objects* together as well as to allow multiple ontologies and projects in one data repository. *Type* is a description of the attributes an object of this type should have. For both objects and types the actual data is stored in *object version* and *type version*. Namespaces and taxonomies are not versioned, because the name and description of them is not particularly relevant to the stored data and not versioning them reduces complexity for the implementation of the repository. *Taxonomy items* however are versioned.

To ensure readability, some attributes and relations were left out on purpose in Figure 4.2. Attributes that were left out are the metadata attributes *created at*, *updated at*, and *deleted at* that are used for almost all entities. These attributes store timestamps. For obvious reasons the *updated at* attribute is not used for entities that cannot change like the versions and the *item relations*. The *deleted at* attribute is also used as the tombstone to mark deleted items. This soft delete strategy in the database allows easy undoing of deletes and preserves relations even to deleted entities. Currently completely removing an entity requires direct database access.

Missing relations in the diagram are for example the relation from object version to taxonomy item or to an object that is referenced by this object version. Since the type defines the attributes of objects, the type version entity is also missing relations. A type version can reference a type or a taxonomy, that an object of this type version can reference an object or a taxonomy item of. The current version of type A can for example reference taxonomy T in the specification of a certain attribute. An object of type A then can reference an item of taxonomy T as value for that attribute.

The complex attribute *object data* contains the complete data of the object. As already stated above, it must be possible to store objects with nested lists, objects, and combinations in this attribute. Because the complex attribute *type specification* contains the specification of the object data, it can be expected to be similarly if not more complex. The database that is used to realize this data structure must support these complex attributes. This is further complicated by the fact that the missing references mentioned above are inside these complex attributes. Also changes in the ontology will affect the structure of these attributes.

A schema-less database would allow different object versions, that are from different versions of the ontology and therefore have a different internal structure, to coexist in the same database. However an SQL database has some benefits that are not found in most schema-less databases. The schema allows the SQL database strict type checking. With foreign keys the database can also enforce the integrity of references. A hybrid scheme, where only the complex attributes are stored in a schema-less database and everything else in an SQL database, could be used to take advantage of the benefits of both database types. Using such a hybrid scheme comes with the disadvantage of having to maintain two different databases at the same time. Also implementing transactions that involve both databases can be a problem.

To support similar use cases of unstructured data in an SQL database one solution is to store them in a binary or text column in a serialized format like XML or JSON. Many SQL databases already include SQL functions to allow working with JSON data in a text column. MariaDB for example supports working with JSON since 2017 [Mar17]. With this only an SQL database with basic support for JSON columns is needed to implement the data structure in Figure 4.2. This also means that all the experience from building the MUSE4Music repository with an SQL database can be applied to the MUSE4Anything repository as well.

JSON was already used as the data format of the MUSE and MUSE4Music repository API. It is easy to understand and natively supported by Python and JavaScript. This makes it an ideal choice for APIs that are used by single page applications running in the browser. Now JSON objects can be directly stored as JSON in the MUSE4Anything database, without the need for further data conversions. This also simplifies the question of how to specify the object types. MUSE4Music already used the OpenAPI 2 specification [Ope14] to describe the API responses. The specification includes data models for the objects from the MUSE4Music ontology. OpenAPI 2 data models use a specification format adapted from JSON Schema [IETF18] with only minor

modifications. For the MUSE4Anything repository the API description is done with the OpenAPI 3 specification [OAI18]. With the new version even more incompatibilities to JSON Schema were removed from the OpenAPI specification. The trend there seems to be to just use the JSON Schema specification without modifications in the future. Therefore the MUSE4Anything repository uses JSON Schema to specify object types.

JSON Schema is itself described using JSON Schema and is written in JSON. This means that the tooling needed to validate the object data can be used to validate the type definitions as well without changes. The same is true for the automatic form and UI generation in the user interface. Also the implementation of these features in the MUSE4Anything repository can build upon experience from implementing the MUSE4Music repository.

Because the MUSE4Anything repository uses the same database type as the MUSE4Music repository, the same tooling can also be used. This shortens the development time as the libraries are already known. However, an SQL database cannot perform foreign key checks if the foreign key reference is inside a JSON object. To get around this issue, all references need to be extracted from the JSON objects first and stored in the database separately in a way that allows foreign key checks. This does duplicate some data, but it enables efficient queries for referenced objects and also enables more integrity checks for the database. Because all JSON data objects are versioned no extra logic for updating these references is needed.

The *item relations* use a different strategy for versioning than the other versioned entities. Because they have essentially no attributes, except for the automatically managed metadata attributes, there is no need to store past versions of the same relation. A relation between two taxonomy items can either exist or not. The soft delete means that a deleted relation can be restored again. To preserve the history of relations, a new relation with the same properties is created instead of resetting the *deleted at* tombstone attribute of the existing relation.

4.2 HTTP API

For the MUSE4Anything repository GraphQL [Fac18] was also considered for implementing the API. The other option was an HTTP API similar to the MUSE4Music repository. GraphQL was not chosen, because it relies on a strongly typed GraphQL schema, that does not allow for the flexibility needed to support generic ontologies along with the changing and user definable data structures for objects. For this reason the MUSE4Anything repository also uses a HTTP API.

Nullable fields are used extensively for the MUSE4Music repository as often only parts of an object are known or entered at a time. The OpenAPI 2 standard does not allow nullable fields. This made workarounds using magic values instead of `null` necessary for the MUSE4Music API. These workarounds cannot be used with a generic ontology. Because the OpenAPI 2 standard required these workarounds the tooling used to build the API had to be changed for MUSE4Anything. The API is implemented with flask-smorest [Nc20] that supports the OpenAPI 3 specification.

Like the MUSE4Music API, the MUSE4Anything API follows the principles behind the REST architectural style described by Fielding [Fie00] as close as possible. However, most of the considerations from the MUSE4Music API [Büh17] still apply. The design of the MUSE4Anything API

is built on the experience with the MUSE4Music API and tries to improve upon the original design where possible. The API design also follows the *WSO2 Rest API Design Guidelines* by Leymann et al. [LFM+16].

API Design Improvements

One improvement over the MUSE4Music API is that all collection resources use pagination by default. This avoids long loading times to fetch the full list of resources. It is also more robust for a large amount of resources in a collection. To get stable pagination even when the collection changes, a cursor is used instead of an offset. The cursor ensures that the next page always starts exactly after the last element of the previous page.

Another major improvement compared to the MUSE4Music API is, that the MUSE4Anything API can be used by a completely hypermedia-driven client. The MUSE4Music API already used the JSON+HAL format [Kel16] to include hypermedia information into the API responses. This format was easy to implement and use even without using special libraries. The format has some shortcomings that limit its usefulness as a hypermedia format.

Other hypermedia formats for JSON like Siren [SHD+17], the Ion Hypermedia Type [Ion18] and Collection+JSON [Amu13] were considered, but these types are more verbose than JSON+HAL and not as easy to implement and use. To solve the issues of JSON+HAL and at the same time retain its ease of use requires using a custom format. This custom format builds upon JSON+HAL but uses concepts from the before mentioned other hypermedia formats to solve some of the issues.

One issue with JSON+HAL is that it mixes its own attributes directly with the actual data. JSON+HAL specifies two attributes `_links` and `_embedded` for links to other resources and for embedding resources that are likely to be requested by a client into a request to avoid more requests over the network. Embedded data in the `_embedded` attribute is only useful for the API client and should be stripped before the data is used. The links are more closely related to the data. This is especially true for the *self* link that represents the current resource. However, other links are not directly part of the resources representation and should also be handled differently than the actual data. Because the JSON+HAL attributes are inserted into the resource data, it is more difficult to correctly separate them later. The "_" prefix of these attributes is also caused by this, as it is used to avoid name collisions with existing attributes.

To avoid this problem the custom format for API responses always wraps data in a JSON envelope. Every attribute of the custom format is specified in that envelope, except for the self link. The actual resource data is in the `data` attribute of the envelope and contains the self link that identifies that resource. The data can be easily separated from the envelope by just using the content of the data attribute. With the self link, the data can be linked to the request again at any time, by simply requesting the resource behind the self link again. Because the attributes of the custom format cannot collide with existing attribute names of the resource data, the "_" prefix can be dropped. The self link in the data uses the attribute name `self` without a prefix. This should be relatively safe from name collisions, as `self` is rarely used as an attribute name. The difference in the structure between JSON+HAL and the custom format can be seen in Listing 4.1. The custom format uses two uniform lists to represent links and embedded resources, where JSON+HAL uses maps with link relations as keys for both.

Listing 4.1 Comparison of the structure of JSON+HAL documents and documents using the custom format developed for the MUSE4Anything repository.

```
// JSON+HAL
{
    "_links": {
        "self": { /* self link */ },
        "someRel": { /* single link */ },
        "otherRel":[ /* list of links */ ]
    },
    "_embedded": {
        "otherRel": [ /* list of embedded resources with the link relation otherRel */ ]
    },
    /* other attributes of the resource */
}


// Custom Format
{
    "links": [ /* list of links */ ],
    "embedded": [ /* list of embedded resources */ ],
    "data": {
        "self": { /* self link */ },
        /* other attributes of the resource */
    }
}
```

This also highlights another problem of JSON+HAL that lies in the way the links are organized. All links are grouped by their link relation in `_links`. The attribute contains a map from a link relation to either a single link or a list of links. This complicates parsing the links, as both cases have to be properly handled. It is also not trivial to decide on which link to follow out of a list of links.

A link relation in JSON+HAL should be a URI that links to relevant documentation for that link relation. Parsing a link relation without using an existing library is even further complicated by the optional CURIE Syntax that can be used to shorten the link relations. Because these link relations are expected to contain characters that cannot be part of a JavaScript variable name, they cannot be accessed with the shorter `_links.rel` notation. A link can only have one relation as it is the key of the map. Some relations like the *self* relation or the *next* relation obscure information that can otherwise be given by the relation. For example a resource might normally be linked by a relation *namespace*, but when the resource is requested directly the same link to the same resource has to have the *self* relation, to signal that this is the canonical link for the requested resource.

As a final complication, links can also contain templated URLs. What values have to be used with a templated URL is not defined and a client can only rely on out-of-band information on how to use the templates correctly. Simply trying if the current resource has the correct attributes required by a URL template may work, but is not guaranteed to. If the representation of an object changes, this can break existing template URLs without knowing.

Listing 4.2 Comparison of links defined in JSON+HAL and the custom format.

```
// JSON+HAL
"first": {
    "href": "/api/namespaces/",
    "name": "namespace",
    "templated": false
}


// Custom Format
{
    "href": "/api/v1/namespaces/",
    "rel": ["collection", "first", "page"],
    "resourceType":"namespace",
    "doc": "/api/doc/namespaces/",
    "schema": "/api/schemas/namespace-schema/"
}
```

To address these problems, the custom format completely changes how the links are organized. Listing 4.2 shows an example link defined in both the JSON+HAL format and in the custom format. For the JSON+HAL link the map key containing the link relation was also included in the listing. The first change for the custom format is to use a single list to contain all links. The link relation of each link must now be specified on the link itself. With all links in a single flat list they can be accessed with ease by iterating over that list. An API client can still build a map out of these link relations for more efficient querying.

The second change is that the link to the documentation is separated from the link relation and given its own attribute (`doc`) in the link. This change makes parsing and using link relations easier, as simple string equality is now sufficient to compare relations. In JSON+HAL the relations can be a URL or a shorted version of the same URL by using the CURIE Syntax. To compare two relations in JSON+HAL, they first have to be transformed into comparable URLs. That the link to the documentation has its own attribute in the custom format makes the attribute name `doc` a label of the link. This makes the intention of the link clearer. Because the documentation is intended for a human audience, the custom format also has a *schema* attribute for links that link to a JSON Schema. A client can use this machine-readable documentation for example for UI or form generation. The schema can also be used by a client to validate its own requests to the API before sending them. Because the schema is directly provided by the API the client does not have to use out-of-band information to get the correct schema.

Because the link relation is no longer used as a map key in the custom format, a link can have more than one relation. The Siren format for example also allows multiple relations for a link. This is especially useful with relations like *self* and *next*, because they can now be used together with other relations and do not have to replace them entirely. Allowing multiple link relations also opens up the possibility to use non-standard relations. A client is expected to ignore all unknown link relations.

Listing 4.3 A link in the custom format describing the action to create a new namespace.

```
{
    "href": "/api/v1/namespaces/",
    "rel": ["create", "post"],
    "resourceType":"namespace",
    "doc": "/api/doc/namespaces/",
    "schema": "/api/schemas/namespace-schema/"
}
```

With multiple link relations combinations of certain relations can be given special meanings. The custom format defines multiple relations and how they influence the meaning of other relations. For example a *collection* is the link relation used to specify that the link leads to a collection resource. If that collection resource is paginated then this is expressed by adding the *page* relation to the link. The relation *nav* can be used to instruct the client to expose that link as a navigation target. Encoding this kind of information into the list of relations of a link, allows a client to make automated decisions on what to do with a link. At the same time the information is also accessible for humans, as the link relations have simple names that ideally can be understood intuitively.

This is a powerful feature of the custom format. It can also be used to encode actions that a client can perform. A link with the relations *create* and *post* can be used to create a resource by using the HTTP verb POST in the request to the API. For a link that encodes an action the schema attribute of the link is not used to describe the format of the action result. Instead, the schema describes the expected input for that action. This allows the client to generate a user interface for that action from the provided schema. The schema could also be used to check client side if the request data is valid. If for example the input format for one action changed, the API could decide to provide a legacy endpoint, that still accepts the old input format, and present both actions as separate links, with the legacy action marked as deprecated. A client can then decide which action to use, by checking if the schema validates. If the client is forced to use the deprecated action, it can choose to alert the user or developer of the client, but otherwise proceed as normal. This could be used to provide a safe upgrade path for clients without explicitly versioning the API.

For actions like a create-action it would be useful, if the client could know what type of resource will be created by that action. The custom format allows the API to specify exactly one link relation as the resource type of that link. With this addition the client can also differentiate actions based on their resource type. An example for this can be seen in Listing 4.3. For normal links the resource type can also be specified. This is especially useful for the self link, as it can be used by the client to determine the type of the resource reliably. For known resource types the client could then display information special to that type or perform other actions special to that type. Because the resource type is also a link relation, the client should treat it as part of the relations of a link.

JSON+HAL allows the API to specify the expected media type of a link and a name that can be used like an additional link relation. While the link name can be used as the second link relation this is not as powerful as an arbitrary number of link relations in the custom format. Knowing the media type of a link before accessing the linked resource is often not needed and the media type of an accessed resource is already part of the HTTP protocol.

Listing 4.4 An example of a link with a key, that can later be used with a matching keyed link to reconstruct the original link.

```
// link with key
{
    "href": "/api/v1/namespace/1/",
    "rel": [],
    "resourceType":"namespace",
    "key": {"namespaceId": "1"}
}


// keyed link
{
    "href": "/api/v1/namespace/{namespaceId}/",
    "rel": [],
    "resourceType":"namespace",
    "key": ["namespaceId"]
}
```

The last major change made for the custom format is how templated URLs are handled. The JSON+HAL specifies that links can contain templated URLs, but does not specify how these templated URLs should be used. To expand a URL template into a correct URL, the values for the variables in the template must be provided. The problem for an API client is how to get these values for a given URL template. A client could try to use the values from the current resource to expand the templated URLs defined for that resource. However, this is not guaranteed to work. Even for the simple example document given in the JSON+HAL specification, the templated URL uses a variable name that is not found anywhere else in the document. This means that a client has to rely on out-of-band information to use templated URLs. While the documentation link is technically not out-of-band information and should describe how to use the template, it is mainly intended for humans and thus inaccessible for the client by automatic means.

To solve this issue the custom format introduces the concept of URL keys. An example can be seen in Listing 4.4. The link with the key can be the self link of a resource or part of the links attribute of a resource. The keyed link containing the templated URL must be specified in the special `keyedLinks` attribute to keep both link types separate. Every keyed link with a templated URL advertises which URL key is needed to expand the URL template. If a URL key matches the advertised key, the client can use the values from the key to expand the template. The keys provide an explicit way for the API to tell the client how to use the templated URLs. A client can store the key of the current resource and later reconstruct the URL of the resource from the key with the matching templated link. This feature is useful for single page applications, like the MUSE4Anything user interface, that have their own client side URLs. Instead of encoding the full API URL of the resource into their client side URL they can encode the key in their URL for shorter and easier to read URLs.

Currently the custom format is still in development and does not define a way for the API to specify how a client can modify an existing key. This is needed to give a client the ability to use templated URLs for searching, where the search string should be provided by a user. It could also be useful for similar use cases, for example for templated URLs that contain the sorting order or a filter for

the results of a collection resource. For these use cases a client still has to rely on out-of-band information, but the out-of-band information needed can be minimized with the use of few, reused names for template variables that a client should be able to modify.

With the custom format the MUSE4Anything API client can navigate the whole API, without relying on out-of-band information of the structure of the API or available actions at each resource. In comparison, the client for the MUSE4Music API has out-of-band knowledge of actions that are available for certain resources built in. Also limited knowledge of the API structure was built into the MUSE4Music API client to navigate to a resource on a page refresh, because the client side URL did not contain the full API URL to the resource. The layout of the MUSE4Music API was also done with consideration of this limitation of the client. Most collection resources were directly accessible from the root API URL. For MUSE4Anything this limitation no longer exists, because the client can make use of the keys to build its client side URL and can build the corresponding API URL without any knowledge of the API structure.

API Structure

The structure of the MUSE4Anything API follows the database structure from Figure 4.2. Every entity has a collection resource and an atomic resource in the API. The hierarchy of the API resources follows the *has* relations. Some additional resources for providing the schemas used in the API are also part of the MUSE4Anything API. An OpenAPI 3 specification of the API is automatically generated. It can be viewed in the integrated Swagger UI and the JSON description of the API is also available in the documentation on Read the Docs. However, using this API documentation to generate a client that depends on the structure of the API would go against the REST principles.

Generating a client from the OpenAPI specification would introduce a tight coupling between the client and the API. With the custom format the API allows clients to navigate the structure of the API and discover the possible actions from the root resource, without ever depending on the URL structure of the API. The custom format tells the client everything he needs to know about the current application state by specifying actions, navigation links, the current resource type, and more. The MUSE4Anything API follows the concept of hypermedia as the engine of application state, briefly mentioned by Fielding in his dissertation [Fie00] and later elaborated in a blog post [Fie08], to the author's best ability. This only shows its advantages when the client uses the hypermedia format to drive its application state. Therefore, it is not advised to generate a client from the OpenAPI specification of the MUSE4Anything API.

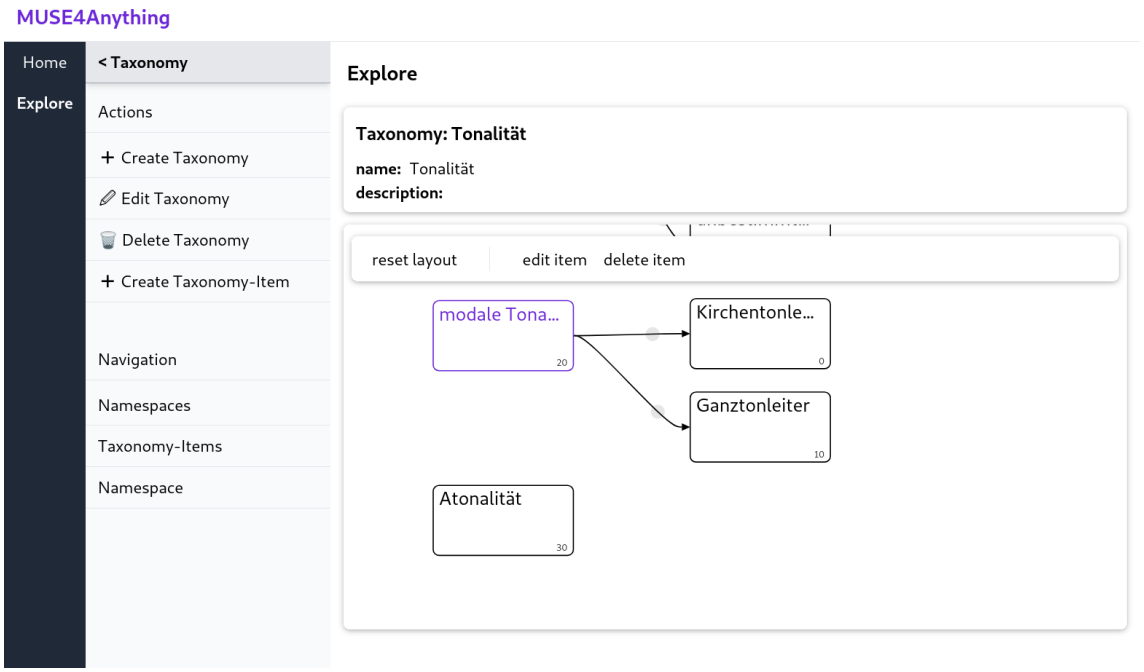

Figure 4.3: The taxonomy view of the MUSE4Anything repository with a zoomed in graph of taxonomy items.

4.3 User Interface

The web front-end of the MUSE4Music repository uses Angular [Goo21]. Because Angular has a high update frequency and the updates usually contain breaking changes, the MUSE4Anything repository web front-end uses Aurelia [Blu20] which has a much more stable API. The choice of the framework also means that the form generation libraries mentioned in Chapter 3 are incompatible and cannot be used. However, implementing a completely custom UI and form generator also allows for much greater customization potential than any of the form generators can offer. It also allows for a better integration of the UI and form generator with the API client. The MUSE4Anything UI generator can also benefit from the experience from implementing the UI generation for MUSE4Music.

The API client of the MUSE4Anything front-end takes full advantage of the custom format of the API responses. The navigation options displayed to the user, including available editing actions like creating or updating new objects, are fully dependent on the links provided with the request response. With the information encoded in the combinations of link relations, the client can make complex decisions on how and in which order to present these links to the user. Figure 4.3 demonstrates this in multiple ways. The side bar on the left contains the links for navigation and performing actions. Starting at the very top of the side bar, where the chevron on the left indicates that this is the link to the next level up in the hierarchy of resources. Below that follow the action links with descriptive symbols. They are sorted so all actions with the same type are grouped and actions for the current resource type are always above other actions. Below that are the navigation links.

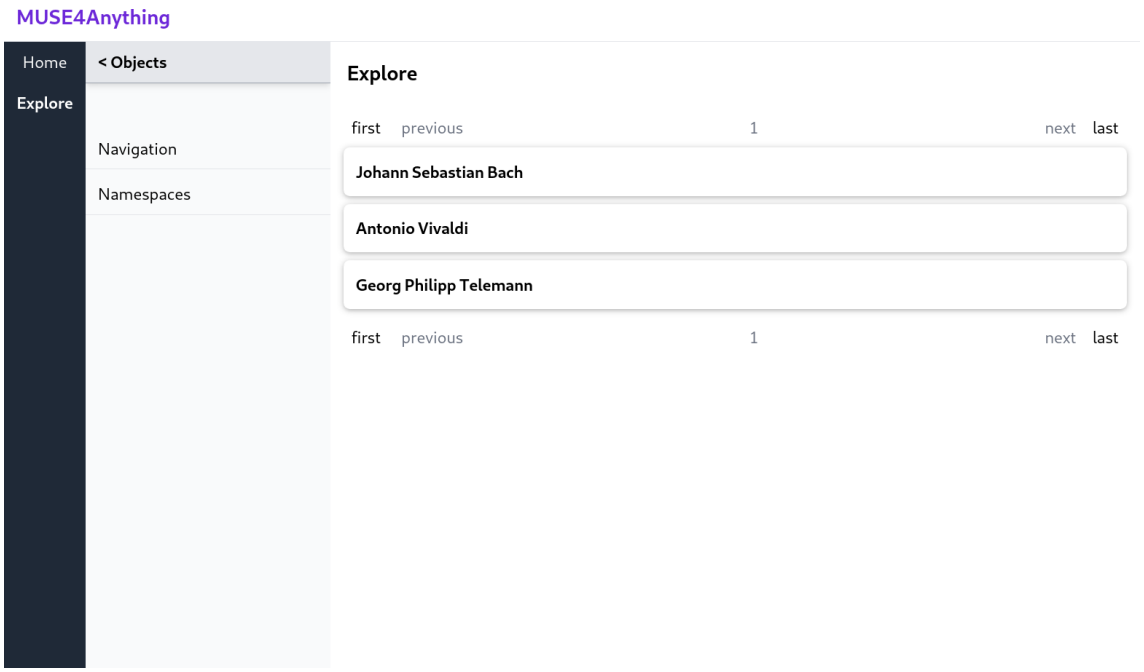

Figure 4.4: The view of a paginated collection of objects in a namespace.

To provide more meaningful information for some resource types, the client uses the resource type link relation of the current resource to decide if and what extra user interface elements to display. The right side of the screenshot in Figure 4.3 is entirely dependent on the resource type. The first card contains the direct information of the current resource. For each resource type there can also be additional information. The taxonomy view for example displays the current taxonomy items in an interactive graph view. This graph view also uses the action links of the resources to decide on which buttons to show. In this case the selected item can be edited or deleted. Another example of a resource type that displays additional information is the view for paginated collections seen in Figure 4.4. The page navigations above and below the page content is entirely dependent on the links provided by the API. Each resource on the page is treated as an embedded resource view. To avoid requesting each resource separately from the API, the client uses a local cache to store the resources the API already embedded into the response containing the page resource. The individual UI elements just naively request the resources every time, while the API client handles the cache to minimize the number of requests that reach the API.

The MUSE4Anything front-end also heavily uses UI and form generation to display or edit the current resource data based on the JSON Schema describing the data. Both UI and form generation use the same intermediate representation of the JSON Schema as basis for their generation. In the intermediate representation the information of the JSON Schema is present in a more uniform structure. References to other included schemas are already resolved. Because some elements of the JSON Schema specification make automatic UI generation impossibly difficult, the intermediate representation excludes the offending keywords by default. This includes the `oneOf`, the `anyOf` , and the `not` keyword. Only draft 7 of the JSON Schema specification is currently supported

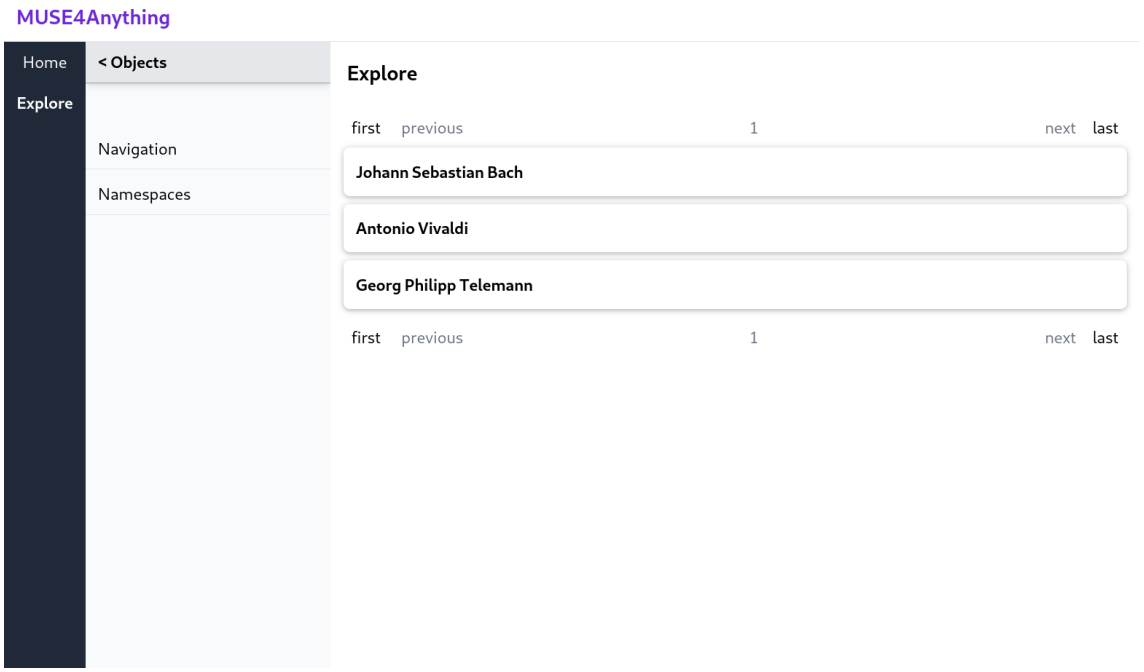

Figure 4.5: The view of single person object in the MUSE4Anything repository.

by the MUSE4Anything front-end, because the newer draft has less tool support and additional documentation. The newer draft also includes more keywords, for example some for conditional schema application, that make UI generation even more difficult.

The results of the automatic UI generation can be seen in Figure 4.5. Everything besides the box heading is auto generated UI. The figure shows the view of an object. The object data is entirely user configurable, except for the name and description, because they are used when only a smaller version of the object should be displayed. The smaller view of an object is displayed for example in a collection like in Figure 4.4. Always having a name and description attribute available just makes it trivial to decide what properties to display as a minimum. Otherwise, the UI would have to always display the whole object, cut it off at some point, or provide a way for the user to choose what properties of the object should be part of the minimal view.

Because JSON Schema is not built to support automatic UI generation, it naturally lacks some features for this use case. The standard allows users to use custom properties in a schema that have to be ignored by validators not knowing these properties. For the MUSE4Anything UI generation only a few additions are needed. An attribute to specify a custom type that requires special UI elements and an attribute to specify the order of properties are both required. Also, a custom attribute to define which properties should never be used for automatic UI generation was used. This is used for example to hide the metadata attributes by default in the UI generation seen in Figure 4.5. The resource contains the attributes for when the object was created or edited, but they are not shown to the user.

The MUSE4Anything front-end also has a special taxonomy editor that allows easy editing of all taxonomy items. The editor can automatically layout the taxonomy items and allows editing of the structure of the taxonomy by drawing or removing links between items. Items can be connected

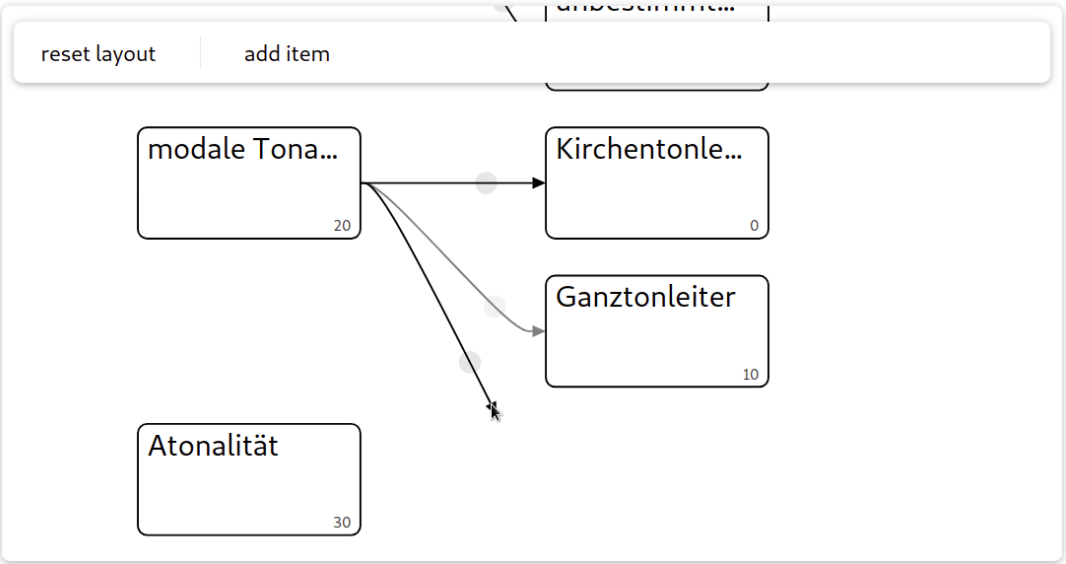

Figure 4.6: Removing an item link in the taxonomy editor by dragging the link into empty space.

to form a tree or a directed acyclic graph structure. Both the editor and the MUSE4Anything back-end ensure that the graph created by these links does not contain any cycles. The editor is implemented with the Grapheditor Webcomponent library [Büh20] that was originally developed for a different student project and was later maintained as a separate library and used in other projects. Figure 4.6 shows the user removing the link to the “Ganztonleiter” (whole-tone scale) taxonomy item, by dragging the arrow into the empty space. The existing links can be dragged from the semi-transparent circle on the arrow. New links can be created by dragging the semi-transparent drag handles that appear on hovering over a taxonomy item to the target taxonomy item. The core drag and drop functionality used for this is already provided by the Grapheditor Webcomponent library. Only the looks of the elements of the graph and what should happen on certain graph events has to be implemented by the user of the library.

5 Evaluation

This chapter is about the evaluation of the implemented MUSE4Anything repository consisting of the database, API, and user interface. The requirements from Chapter 2 will be used as the baseline for the evaluation. To showcase how the MUSE4Anything repository satisfies these requirements and how it compares to one of the existing data repositories, the evaluation will follow a small example use case. The use case is chosen such, that it can showcase all features of the MUSE4Anything repository with the minimal amount of complexity needed. This makes the examples shown in the thesis shorter and easier to understand, while still allowing for the evaluation of all features.

5.1 The Evaluation Use Case

A representative use case is used to evaluate the implementation of the MUSE4Anything data repository. The representative use case is modelled after a small part of the MUSE4Music ontology. Figure 5.1 shows the target use case as a combination of a class diagram and an object diagram. For

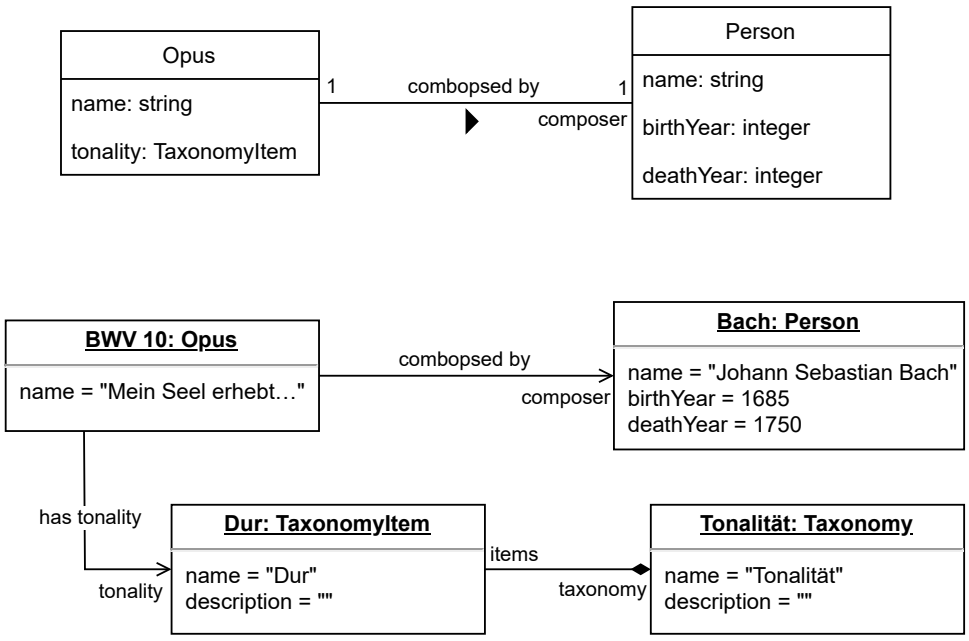

Figure 5.1: The example use case used to demonstrate the features of the MUSE4Anything repository. The use case includes the two object types *Person* and *Opus*. It tests normal attributes like *birthYear*, object references with the *composed by* relation, and taxonomy item references with the *tonality* attribute.

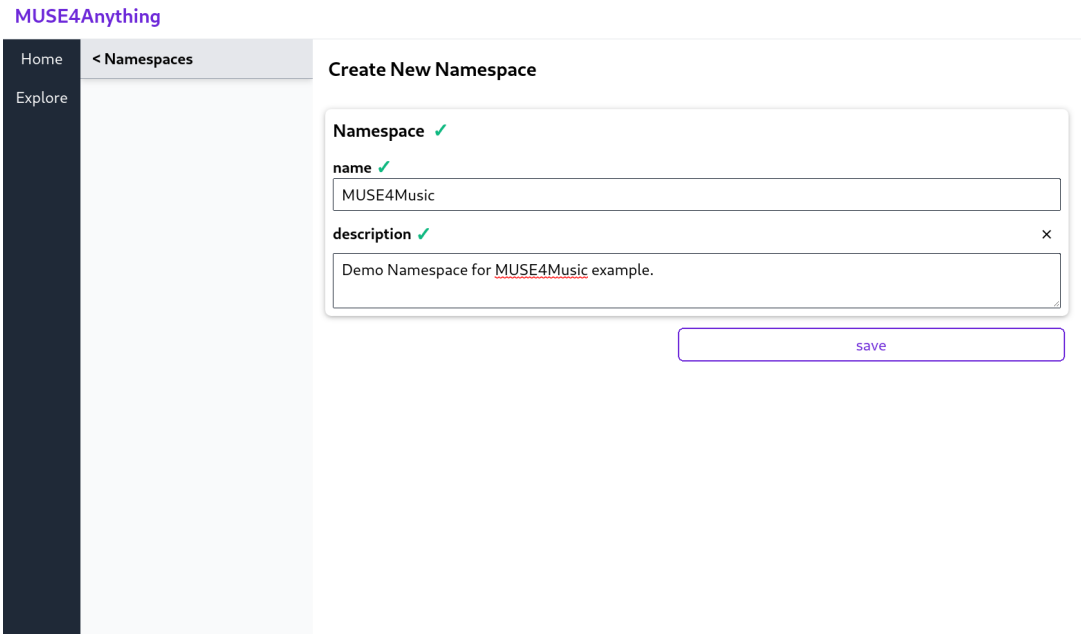

Figure 5.2: Form to create a new namespace in MUSE4Anything.

the evaluation a completely new namespace is used in the MUSE4Anything repository. Some parts of the diagram are in German, because they were taken directly from MUSE4Music, which is currently only available in German. The values used in this use case are purely for demonstration purposes. All features of the MUSE4Anything repository, that are also implemented in the MUSE4Music repository, are compared to the same features in the MUSE4Music repository.

5.2 Creating the Ontology

The first part of the evaluation is about the implementation of the "Manage Ontology" use case of the MUSE4Anything repository. Because this use case also contains the "Edit Taxonomy Content" use case, they will be evaluated together. To realize the example use case in MUSE4Anything, first a new namespace has to be created. This namespace will contain the two types *Opus* and *Person*, the taxonomy *Tonalität* (tonality) and all objects used. All forms in the MUSE4Anything repository use exactly the same automatic form generation based on JSON Schema. An example of how these forms look like can be seen in Figure 5.2 and Figure 5.3. Because Opus depends on both Person and Tonalität, it is created last.

The taxonomy Tonalität is directly copied from the MUSE4Music repository. Figure 5.4 is a side by side comparison of the taxonomies in their taxonomy editors. The MUSE4Music editor uses a tree view to display the taxonomy and has limited editing capabilities. It only allows users to change the name and description of taxonomy items, to create, and to delete items. Adjusting the sort order of the items or rearranging the tree structure is impossible in the editor. The new MUSE4Anything taxonomy editor uses a graph based view to display the taxonomy items and their relations. This allows the MUSE4Anything repository to even support directed acyclic graphs in taxonomies. The item relations can be freely manipulated directly in the graph view. Items can also be created,

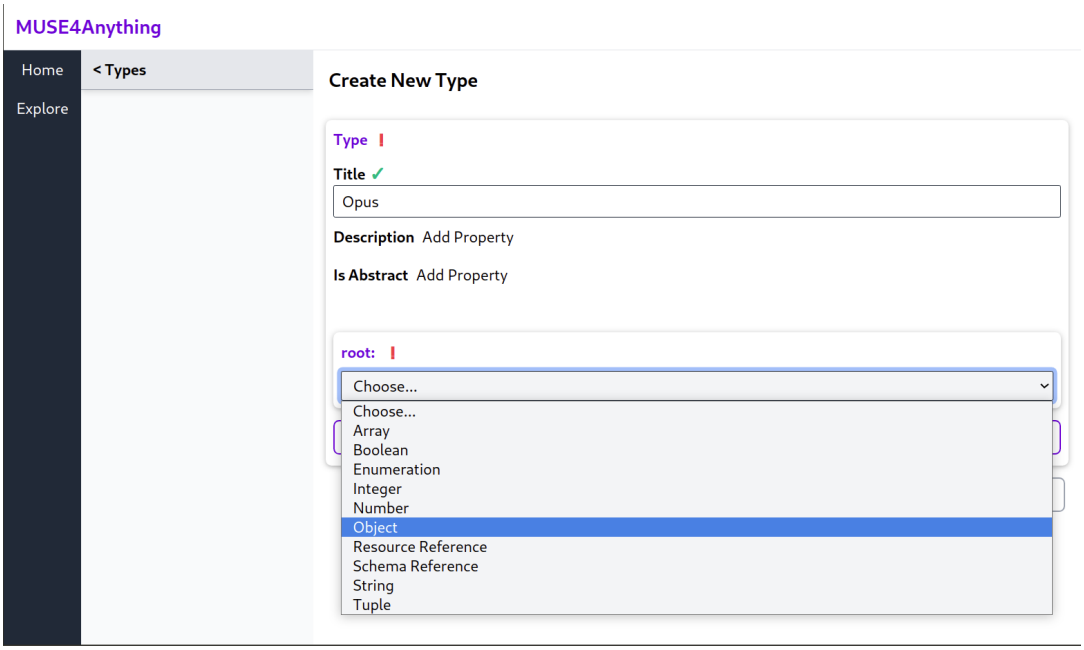

Figure 5.3: Form to create a new type in MUSE4Anything. To create a new type, the user first has to choose the data type to use. The red Exclamation marks show where the input is invalid or incomplete.

removed, and edited in the graph view using the form element in the lower right corner. The sort order of items is by default alphabetically for items on the same level. The order can be influenced by setting a different value for the sort key of an item, to overwrite the default sort order.

The type editor can be seen in Figure 5.3. A full screenshot of the type editor can be found in Appendix B. Figure 5.5 depicts the part of a type form defining a single property for the birth year of a person. Because all possible properties are always displayed, the full form ends up rather large, but this length is comparable to the length of some forms found in the MUSE4Music repository. The benefit of this is that all possible properties can be seen directly in the form and are not hidden behind other user interface elements or in a manual. This promotes trying out these properties. To keep the form in a manageable size, properties must first be explicitly added with the “Add Property” button beside them before they can be given a value. Properties can also be removed later with the “x” button to the right of the property name. The form also highlights all titles of properties that contain the currently focused property when editing. This allows for easier identification where the property is located in the hierarchy. The type editor also showcases the ability to edit lists and deeply nested objects.

The result of the Person type can be seen in Figure 5.6. The complete form of the person type was moved to the appendix because of its length. Currently the display of types follows the form of the JSON Schema closely.

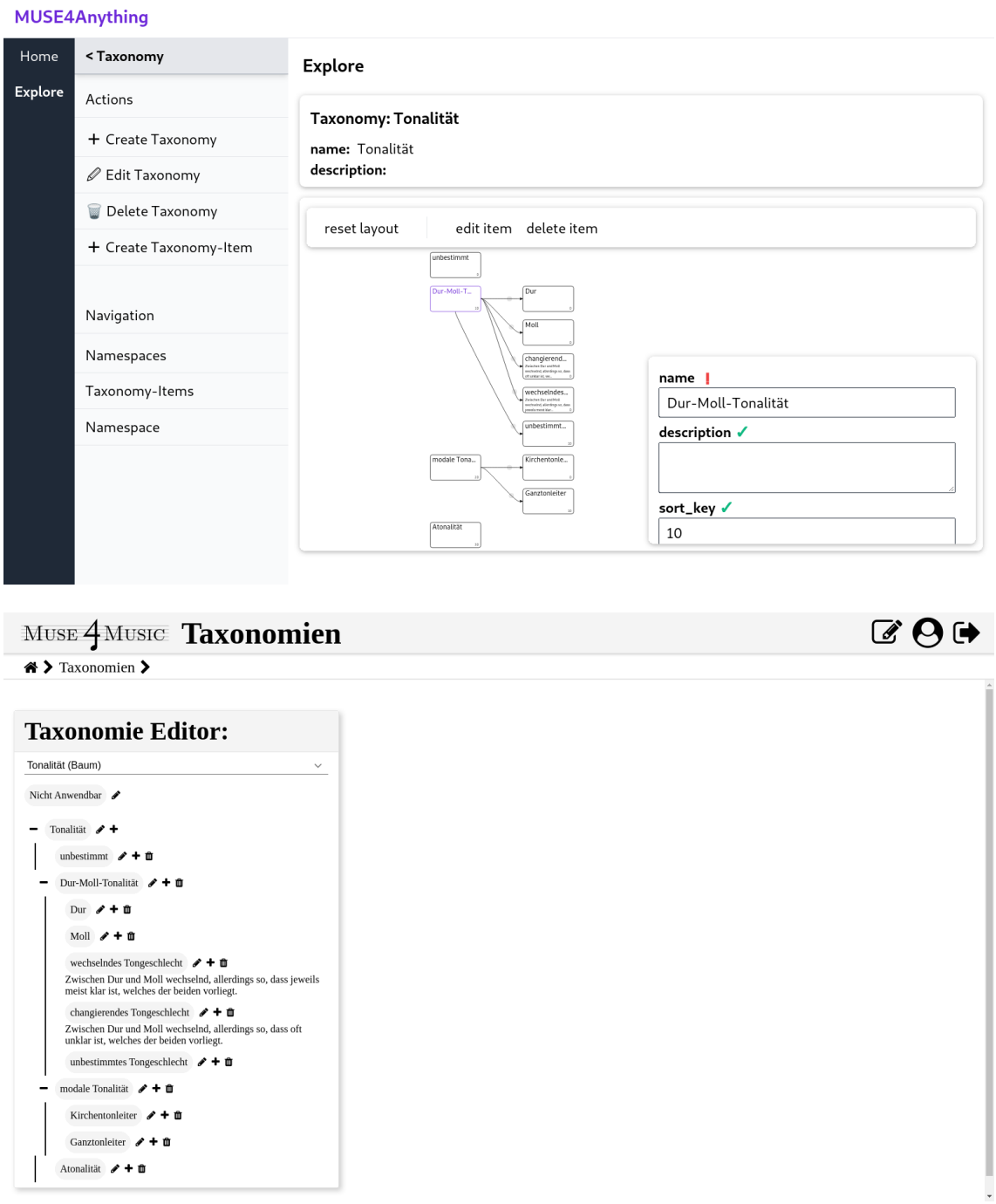

Figure 5.4: Comparison of the MUSE4Anything taxonomy editor (top) and the MUSE4Music taxonomy editor (bottom).

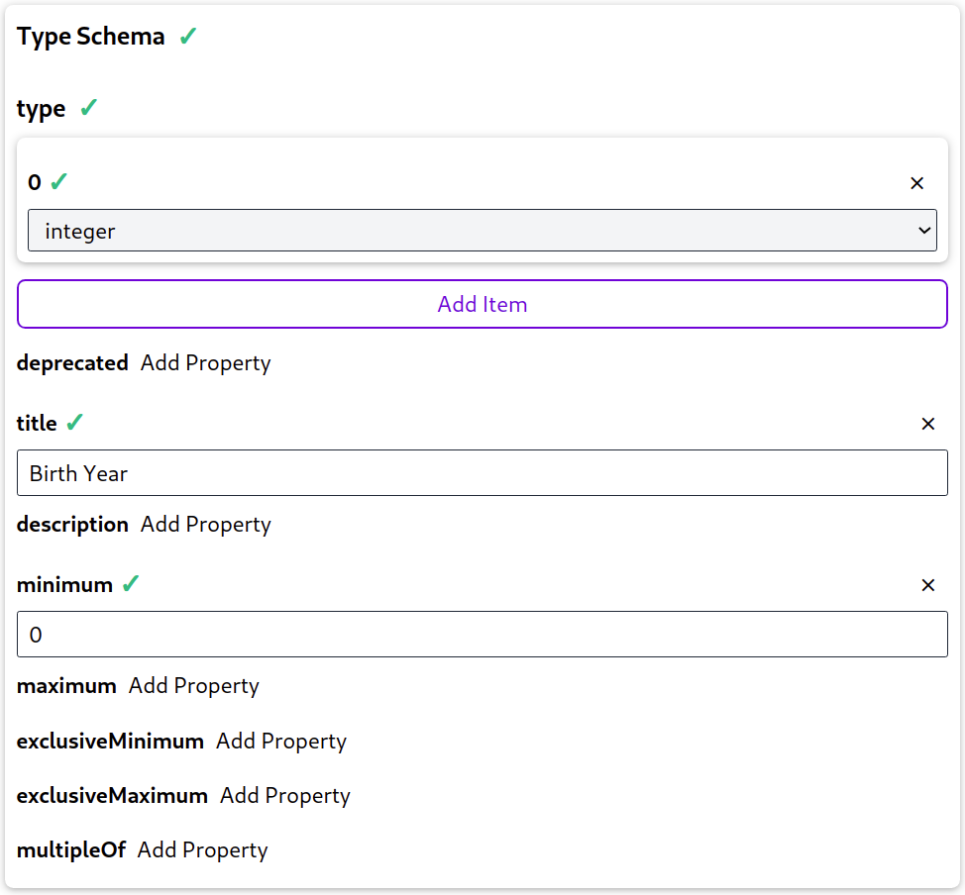

Figure 5.5: The form for a single property of an object isolated from the full type creation form.

Evaluation of the requirements of the use case "Manage Ontology"

- The MUSE4Anything repository *should* be usable without installation of a custom software.

  This requirement is satisfied, as the complete MUSE4Anything repository user interface is realized as a single page application accessible with a standard web browser.

- The MUSE4Anything repository *should* provide an easy-to-use interface to edit the ontology.

  The editor for types should be usable for anyone that already understands JSON Schema. Creating new types with properties, while ignoring most of the validation related attributes of JSON Schema, should also be doable after a short learning period. However, there is still work to do to make larger type definitions with many attributes more manageable for users, as the editor can get unwieldy because of length.

- The MUSE4Anything repository *must* support defining all data types mentioned in the requirements of the input data use case.

  This feature is satisfied, as the MUSE4Anything repository supports all JSON types and has custom types for taxonomy item references and references to other objects.

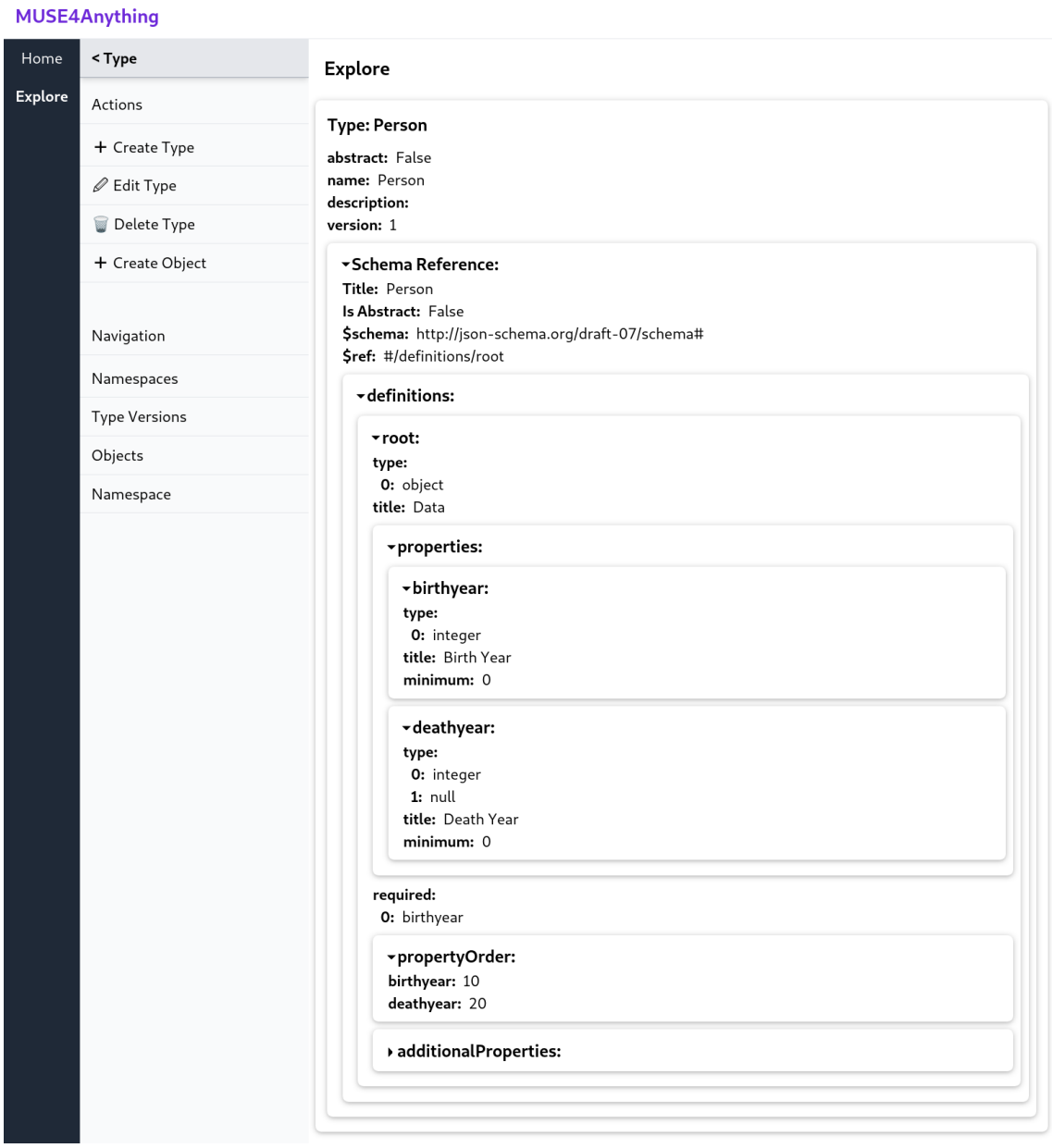

Figure 5.6: The created Person type.

- The MUSE4Anything repository *should* allow the user to influence how the data input form is generated. This includes the order of the input fields, their labels, and optionally their descriptions.

  This feature is satisfied through the title property of JSON Schemas. For objects the MUSE4Anything repository also uses a custom property in the JSON Schema to allow users to define the order of Properties.

- The MUSE4Anything repository *may* allow the user to reuse or extend existing data type definitions.

  This feature is partially satisfied. The feature works technically for UI and form generation, but there is no user interface supporting this feature yet.

- The MUSE4Anything repository *must* retain all versions of a data type definition, to allow for existing objects to reference the version they were created with.

  This feature is satisfied, as the MUSE4Anything repository stores all versions of types and objects and only allows soft deleting.

Evaluation of the requirements of the use case "Edit Taxonomy Content"

- The MUSE4Anything repository *must* be usable without installation of a custom software.

  The requirement is already satisfied.

- The MUSE4Anything repository *must* provide an easy-to-use interface to edit the taxonomy content. The interface should not be more complex than a normal mind-map editor.

  This feature is satisfied by the taxonomy editor. The layout of the taxonomy-items is similar to the taxonomy editors in the MUSE4Music or MUSE repository, but the user interaction is closer to that of mind map editors. With the intentionally limited edit options compared to mind-map editors, it should be easy to learn.

- The MUSE4Anything repository *must* support taxonomies where the items form a flat list or a tree structure.

  This requirement is satisfied. The MUSE4Anything repository even supports directed acyclic graphs.

- The MUSE4Anything repository *may* support taxonomies where the items form an acyclic directed graph.

  See requirement above.

- The MUSE4Anything repository taxonomy items *must* have at least a name and a description property.

  This requirement is satisfied.

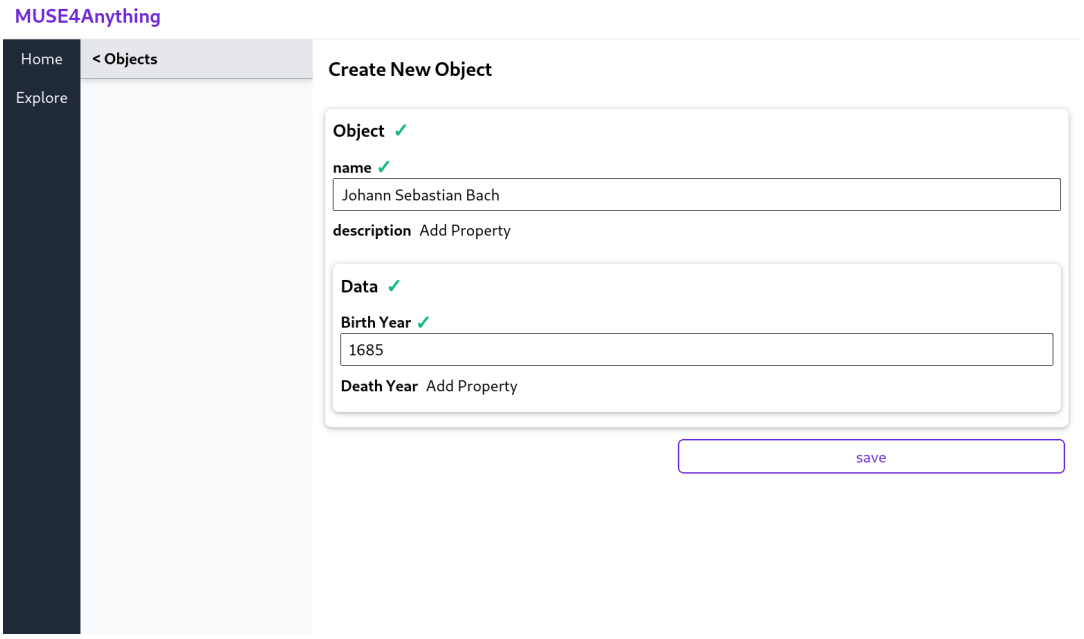

Figure 5.7: The form to create a new Person object in the MUSE4Anything repository.

- The MUSE4Anything repository *must* allow objects that reference old or deleted taxonomy items to exist.

  This requirement is satisfied. The MUSE4Anything repository stores old versions of the taxonomy items and only allows soft deleting taxonomy items.

- The MUSE4Anything repository *may* record changes to the taxonomy structure and taxonomy items in a edit history or use other means of versioning the taxonomy.

  The MUSE4Anything repository stores old versions of the taxonomy items and records changes to relations between items separately. This can be used to reconstruct old states of the taxonomy. In some cases a perfect reconstruction may not be possible, because the order of edits relies on time stamps that are susceptible to race conditions.

5.3 Creating and Viewing Objects

The second part of the evaluation is about the use case “Input Data” and by extension also “View Data”. Because the use case “Identify Patterns” also includes the “View Data” use case and is therefore related, the requirements for pattern identification will also be discussed at the end of this section.

The view of the Person type in Figure 5.6 already showcases most of the features for viewing data. When displaying the data, all properties without a value are omitted for a more compact view. Objects can also be collapsed to hide their contents temporarily. The nesting of objects is displayed by nesting the cards and the card shadows that create a 3d effect.

Figure 5.6 depicts the generated form for creating a new Person object. Objects in the MUSE4Anything repository always have a name and a description property that is used everywhere, where only a smaller view of the object is displayed. A type does not have to define these properties again. For this reason the type definition for Person did not have to include a property for the person's name. Required properties will always be displayed without the extra step of pressing the "Add Property" button. Because the property for the death year is not required, the "Add Property" button is shown by default.

Figure 5.8 shows a side by side comparison of the form to create an Opus object of MUSE4Anything and MUSE4Music, with the taxonomy item select element open in both forms. The taxonomy item select of MUSE4Music shows the taxonomy name as the root item and a special "Nicht Anwendbar" (not applicable) item that can be selected when no value from the taxonomy matches. In the MUSE4Anything taxonomy item select both these items can just be implemented as normal items, because the taxonomies are not limited to exactly one root node. The MUSE4Anything repository allows graphs with multiple top level items that have no parent item. For this example use case only the actual taxonomy items excluding the special items were copied to MUSE4Anything. This means that every node under "Tonalität" was copied.

Because all the extra visual elements in the data input forms can be distracting, both MUSE4Music and MUSE4Anything can display the data in a way more optimized for viewing. The MUSE4Music repository front-end has a global switch to change from edit mode to view mode in the top right corner. The MUSE4Anything repository front-end displays the data in view mode by default and allows the user to edit the data with the "Edit Object" action on the left. The data in MUSE4Anything is shorter, because only the minimal use case was implemented, while MUSE4Music already supports the full ontology. Both show the composer with his full name, but only MUSE4Music can display the complete data of the composer directly in the opus view. The composer object can be accessed by clicking a link. In MUSE4Anything the presented object name is the link to that object, while MUSE4Music presents an extra link icon. One difference is that the MUSE4Anything front-end currently dos not differentiate referenced objects from referenced taxonomy items visually like the MUSE4Music front-end does.

The data, that is displayed in the front-end, comes directly from the HTTP API of the MUSE4Anything back-end. As already mentioned in Chapter 4 the API uses JSON and JSON Schema, which itself is valid JSON, as the data formats for requests and responses. The data for objects and types is even stored as JSON in the database. The MUSE4Anything back-end supports SQLite, MariaDB and MySQL databases. All supported databases are common and there exist a lot of tools to interact with them.

Evaluation of the requirements of the use case "Input Data"

- The MUSE4Anything repository *must* be usable without installation of a custom software.

  This requirement is already satisfied.

- The MUSE4Anything repository *must* support basic data types like strings, numbers, boolean values, references to taxonomy items and references to other objects.

  This requirement is satisfied. The Opus type showcases both taxonomy references and object references. The Person type and the input form for that type show all the other data types.

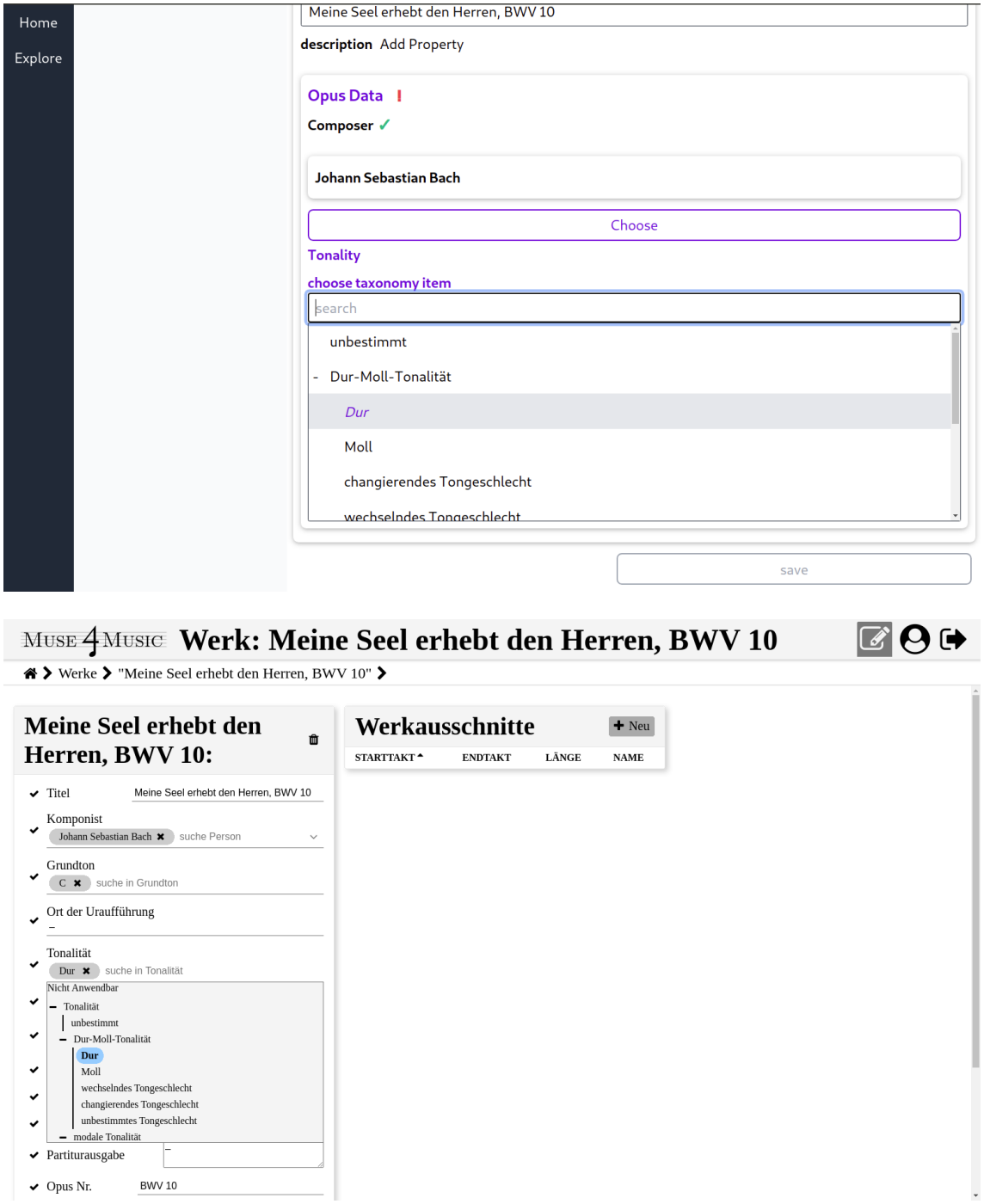

Figure 5.8: The opus input form with an open taxonomy item select element in both MUSE4Anything (top) and MUSE4Music (bottom).

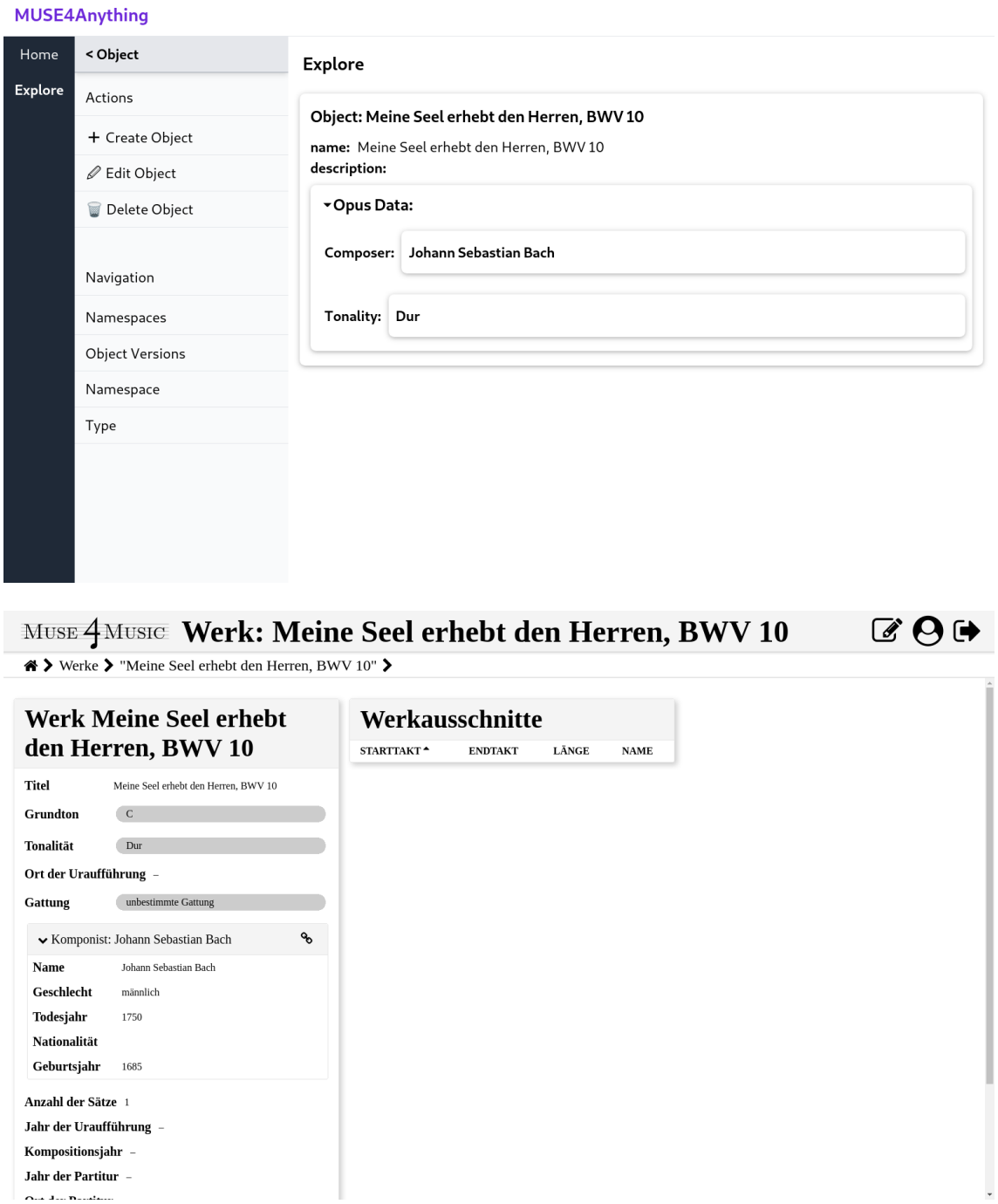

Figure 5.9: The opus view in both MUSE4Anything (top) and MUSE4Music (bottom).

- The MUSE4Anything repository *must* support complex data types including nested objects and lists.

  This requirement is satisfied. While the types defined as part of the use case are not complex, the form to create these types is. It showcases lists and multiple levels of nested objects. Because all forms in the MUSE4Anything user interface use exactly the same automatic form generation code, this can be generalized to forms that create objects. The same holds for the UI generation.

- The MUSE4Anything repository *must* show errors to the user as soon as possible and reject malformed data.

  The requirement is satisfied. The MUSE4Anything front-end already checks the user input for errors based on the type definition and displays a red exclamation mark indicator for malformed data. If the form contains errors, the save button is inactive. The front-end currently does not perform full JSON Schema validation on the user input. Some errors may not be displayed directly in the front-end. The MUSE4Anything API also checks incoming data and performs full JSON Schema validation to reject all malformed data.

- The MUSE4Anything repository *must* always use the current version of the user definable ontology for data input.

  This requirement is satisfied, because the MUSE4Anything API always links the most recent schema of a type for the corresponding create object action and always checks incoming data against the newest version.

- The MUSE4Anything repository *must* be able to handle many objects.

  The requirement is satisfied. The MUSE4Anything API uses pagination for all collection resources. This was tested manually with over 10000 items. The requests had a noticeable delay, but were generally answered in less than one second.

- The MUSE4Anything repository *should* make it hard to lose data on accident. Deleting or overwriting data (on accident) should not lead to data loss immediately.

  This requirement is satisfied, because all object data is versioned and can only be soft deleted via the API.

Evaluation of the requirements of the use case “Identify Patterns”

- The MUSE4Anything repository *must* provide a view explicitly for viewing the objects that has minimal visual clutter.

  This requirement is satisfied, as evidenced by the figures showing these views for the Person type and the opus object of the example use case.

- The MUSE4Anything repository *may* provide a graph view to visualize object references.

  This requirement is not satisfied.

- The MUSE4Anything repository *may* provide a view to compare two or more objects.

  This requirement is not satisfied.

- The MUSE4Anything repository *must* provide machine-readable data in a standardized format like JSON via the API or direct database access for use with external tools.

  This requirement is satisfied by the API and possibly by an admin providing direct database access to the analysis tools.

- The MUSE4Anything repository *must* provide the ontology in a machine-readable and standardized format like JSON via the api or direct database access for use with external tools.

  The requirement is satisfied, because the type definitions are stored in the same database and are accessible from the API in the same manner as objects are. This is also true for taxonomies.

5.4 Result of the Evaluation

This example use case shows that the MUSE4Anything repository can be used to model similarly complex ontologies as the MUSE4Music ontology and create and view objects based on the ontology. The data input experience in the MUSE4Anything front-end is similar to the experience in the MUSE4Music and MUSE front-end. A key feature, the taxonomy item select element, has the same functionality as the MUSE4Music counterpart. All this is achieved without a specific ontology being programmed into the MUSE4Anything source code. The ontology can be defined directly in the MUSE4Anything front-end using the same familiar interface as for data input. The content of taxonomies is fully editable, including the structure and order of taxonomy items. Even creating or deleting taxonomies is supported in the MUSE4Anything repository.

The data stored in the MUSE4Anything repository is available via the HTTP based API in the JSON format. It can also be accessed directly from the SQL database of the repository. The description for the data format is also available via API and in the SQL database in the JSON Schema format that is used to define the ontology. This allows for rich introspection into the data using automatic tools. This is best demonstrated by the automatic UI and form generation of the MUSE4Anything repository front-end. In conclusion the MUSE4Anything repository sufficiently supports all four use cases discussed in Chapter 2.

6 Summary and Outlook

The beginning the thesis discusses the requirements of a theoretical generic data repository that can be used with research projects following the MUSE methodology. The requirements were explored based on use cases identified for the already existing MUSE and MUSE4Music repositories. Requirements for the ontology management with the generic repository were inferred from experience with both projects and their ontologies. After establishing the use cases and their requirements, a short survey of existing tools that support similar use cases was done in Chapter 3, to determine tools that could be used for implementing the generic repository.

Chapter 4 then contains the implementation of the generic data repository named MUSE4Anything. The implementation mostly builds upon the implementation of the MUSE4Music repository.This thesis only highlights the differences compared to the existing implementation of the MUSE4Music repository, as it has a similar architecture. The main differences being the database design, that now supports the generic objects needed for a generic data repository, and the API design.

The database design had to support generic objects with a structure unknown at design time of the database schema. This is normally not something that a standard SQL database handles well, as they need a fixed schema for all tables that cannot easily be changed later. This was solved by using a column type that supports JSON objects.

The MUSE4Anything API uses a custom format for responses, that contains much more information compared to the JSON+HAL format used in the MUSE4Music API. The format was specially designed for most of this information to be machine-readable and understandable. This allows the API and the API-client to follow the REST principles more accurately. Both the API and the client use the custom hypermedia format as the engine of the application state. This is amongst the highest goals to achieve for APIs following the REST principles.

The MUSE4Anything front-end, that has to support all these generic objects, uses UI and form generation heavily. JSON Schema is used as the description format for the generation with small custom additions. While the UI and form generation was built on the experience with the MUSE4Music front-end, the MUSE4Anything front-end uses a different framework. Angular, the framework used for the MUSE4Music front-end, had many breaking changes in their frequent updates making the front-end hard to maintain. The Aurelia framework promises little to no breaking changes on future updates and was chosen for this reason.

At last, in Chapter 5 the implemented MUSE4Anything repository is evaluated against the requirements formulated in Chapter 2. The evaluation follows a representative example use case, that is based on a part of the existing MUSE4Music ontology and data repository. The example use case is tested in the MUSE4Anything repository and the MUSE4Music repository in parallel to allow for easy comparisons. Along with the example use case the requirements of the four main use cases from Chapter 2 and whether the MUSE4Anything repository satisfies them is discussed.

The results of the evaluation showed, that the MUSE4Anything repository can support ontologies of similar complexity as the MUSE4Music or MUSE ontology. The important difference between the MUSE4Anything repository and for example the MUSE4Music repository is that the user can define that ontology directly in the data repository. This was previously impossible. Specialized tools, customized to the ontology by a software developer, were needed to support the MUSE and MUSE4Music projects. The MUSE4Anything repository solves this issue completely. The experience for data input was intentionally designed to be similar to the experience in the MUSE4Music repository. Other features, for example the taxonomy editor, were even improved for the MUSE4Anything repository.

Ontologies defined in the MUSE4Anything repository can evolve, as the repository allows the user to create new versions of already defined object types. The versioning of types and objects ensures, that there is no possibility of accidental data loss. The MUSE4Anything repository can potentially completely replace the mind-map tools, that are currently used to maintain the MUSE and MUSE4Music ontologies. If the MUSE4Anything repository is used as the single source of truth for the ontology, this would also make manual comparisons of different versions of the ontology unnecessary.

Outlook

Before the MUSE4Anything data repository can be used in real research projects, basic authentication and user management still has to be implemented. However, the groundwork for this is already in the tool and should be easy to add. Then the MUSE4Anything repository is ready for use. It could even replace the custom-built MUSE4Music repository, as the current data repository is still unfinished.

There are many improvements possible. A graph like view of object relations is one such improvement. As mentioned multiple times in Chapter 3, one of the graphical notations seen in existing tools could be used to visualize the ontology. A visual editor for the types, that allows users to edit them in a mind-map like manner could improve the usability. Support for more basic types like time durations, dates or even domain specific types with a plug-in system is also on the list of possible improvements.

With the namespaces, multiple projects can share the same MUSE4Anything instance. Currently the namespaces completely isolate the ontologies. Allowing namespaces to reference types, taxonomies or objects of other namespaces would enable cooperative projects that connect different domains.

Last but not least, options for exporting the data are also possible improvements. This would allow sharing research data more easily. An export format that can be read by ontology tools like Protégé would allow a lot of tools to access the data. Then OWL reasoners could also be used for the ontologies created with the MUSE4Anything repository. Some visualization tools mentioned in Chapter 3 could also be used right away, given that the right export format is supported by the MUSE4Anything repository.

Bibliography

[21a] *DogmaModeler*. 2021. URL: http://www.jarrar.info/Dogmamodeler/index.htm (cit. on p. 22).

[21b] *Eddy*. 2021. URL: https://www.obdasystems.com/eddy (cit. on p. 21).

[21c] *Fluent Editor*. 2021. URL: https://www.cognitum.eu/semantics/FluentEditor/ (cit. on p. 21).

[21d] *JSON Forms*. 2021. URL: https://www.jsonforms.io (cit. on p. 23).

[21e] *MindMup*. 2021. URL: https://www.mindmup.com (cit. on p. 24).

[21f] *Ontopia*. 2021. URL: https://github.com/ontopia/ontopia (cit. on p. 21).

[21g] *OWLGrEd*. 2021. URL: http://owlgred.lumii.lv (cit. on p. 21).

[21h] *Protégé*. 2021. URL: http://protege.stanford.edu/ (cit. on p. 20).

[21i] *UI Schema for React*. 2021. URL: https://ui-schema.bemit.codes (cit. on p. 23).

[21j] *VocBench*. 2021. URL: http://vocbench.uniroma2.it (cit. on p. 21).

[21k] *vue-jsonschema-form*. 2021. URL: https://github.com/roma219/vue-jsonschema-form#readme (cit. on p. 23).

[Amu13] M. Amundsen. *Collection+JSON - Document Format*. Feb. 24, 2013. URL: http://amundsen.com/media-types/collection/format/ (visited on 02/28/2021) (cit. on p. 30).

[Bar17] J. Barzen. *MUSE – Muster Suchen und Erkennen*. 2017. URL: www.iaas.uni-stuttgart.de/forschung/projects/MUSE/ (cit. on p. 7).

[Bar18] J. Barzen. "Wenn Kostüme sprechen – Musterforschung in den Digital Humanities am Beispiel vestimentärer Kommunikation im Film". http://nbn-resolving.de/urn:nbn:de:hbz:38-91341. PhD thesis. University of Cologne, 2018. URL: http://kups.ub.uni-koeln.de/id/eprint/9134 (cit. on p. 7).

[BBČ+10] J. Bārzdiņš, G. Bārzdiņš, K. Čerāns, R. Liepiņš, A. Sprogis. "UML Style Graphical Notation and Editor for OWL 2". In: *Perspectives in Business Informatics Research*. Ed. by P. Forbrig, H. Günther. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 102–114. ISBN: 978-3-642-16101-8. DOI: 10.1007/978-3-642-16101-8_9 (cit. on p. 21).

[BBE+16] J. Barzen, U. Breitenbücher, L. Eusterbrock, M. Falkenthal, F. Hentschel, F. Leymann. "The vision for MUSE4Music. Applying the MUSE method in musicology". In: *Computer Science - Research and Development* (Nov. 2016), pp. 1–6. DOI: 10.1007/s00450-016-0336-1. URL: http://www.iaas.uni-stuttgart.de/RUS-data/ART-2016-17%20-%20The%20vision%20for%20MUSE4Music.pdf (cit. on p. 7).

[BDL+12] A. Burdick, J. Drucker, P. Lunenfeld, T. Presner, J. Schnapp. *Digital_Humanities*. Mit Press, 2012 (cit. on p. 7).

[BFL18] J. Barzen, M. Falkenthal, F. Leymann. "Wenn Kostüme sprechen könnten: MUSE – Ein musterbasierter Ansatz an die vestimentäre Kommunikation im Film". In: Digital Humanities. Perspektiven der Praxis. Berlin: Frank und Timme, May 2018, pp. 223–241. ISBN: 978-3-7329-0284-2. URL: https://www.iaas.uni-stuttgart.de/publications/INBOOK-2018-05-MUSE.pdf (cit. on p. 7).

[Blu20] Blue Spire Inc. *Aurelia*. 2020. URL: http://aurelia.io (visited on 02/27/2021) (cit. on pp. 26, 36).

[Büh17] F. Bühler. "Design und Implementierung eines Storage-Backends für MUSE4Music". B.S. thesis. University of Stuttgart, 2017. DOI: 10.18419/opus-9566. URL: http://elib.uni-stuttgart.de/handle/11682/9583 (cit. on pp. 7, 25, 29).

[Büh20] F. Bühler. *Grapheditor Webcomponent*. Nov. 8, 2020. URL: https://github.com/UST-MICO/grapheditor (visited on 03/17/2021) (cit. on p. 39).

[Duc21] DuckDuckGo. *DuckDuckGo*. Feb. 2021. URL: https://duckduckgo.com/ (visited on 03/17/2021) (cit. on p. 19).

[EBH17] L. Eusterbrock, J. Barzen, F. Hentschel. *Eine Ontologie symphonischer Musik des 19. Jahrhunderts*. Tech. rep. Universität Stuttgart – Fakultät Informatik, Elektrotechnik und Informationstechnik, 2017. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2017-02/TR-2017-02.pdf (cit. on p. 7).

[Fac18] Facebook, Inc. *GraphQL*. June 2018. URL: http://spec.graphql.org/June2018/# (visited on 02/28/2021) (cit. on p. 29).

[Fer21] D. Ferreyra. *TemaTres*. 2021. URL: https://vocabularyserver.com/index.html (cit. on p. 22).

[Fie00] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (cit. on pp. 29, 35).

[Fie08] R. T. Fielding. *REST APIs must be hypertext-driven*. Oct. 20, 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (visited on 02/28/2021) (cit. on p. 35).

[Fre20] Freeplane. *Freeplane*. Dec. 10, 2020. URL: https://www.freeplane.org/wiki/index.php/Home (visited on 03/10/2021) (cit. on p. 24).

[Goo21] Google. *Angular*. 2021. URL: https://angular.io (visited on 02/27/2021) (cit. on pp. 26, 36).

[Hal20] T. Halpin, ed. *Object Role Modeling. The Official Site for Data Modeling*. 2020. eprint: http://orm.net/pdf/ORMwhitePaper.pdf. URL: http://www.orm.net (cit. on p. 22).

[IETF18] A. Wright, H. Andrews, eds. *JSON Schema: A Media Type for Describing JSON Documents*. draft-handrews-json-schema-01. Version Draft 7. Internet Engineering Task Force. Mar. 19, 2018. URL: https://tools.ietf.org/html/draft-handrews-json-schema-01 (cit. on p. 28).

[Ion18] Ion Working Group. *The Ion Hypermedia Type*. May 29, 2018. URL: https://ionspec.org (visited on 02/28/2021) (cit. on p. 30).

[Kel16] M. Kelly. *JSON Hypertext Application Language – draft-kelly-json-hal-08 (work in progress)*. 2016. URL: https://tools.ietf.org/html/draft-kelly-json-hal-08 (cit. on p. 30).

[LFM+16] F. Leymann, J. Fonseka, S. Malalgoda, N. Dias, S. Medagammaddegedara, M. Amarasinghe. *WSO2 Rest API Design Guidelines*. Apr. 2016. URL: https://wso2.com/whitepapers/wso2-rest-apis-design-guidelines/ (visited on 02/28/2021) (cit. on p. 30).

[LLM+19] V. Link, S. Lohmann, E. Marbach, S. Negru, V. Wiens. *WebVOWL*. 2019. URL: http://visualdataweb.de/webvowl/ (cit. on p. 21).

[LNHE16] S. Lohmann, S. Negru, F. Haag, T. Ertl. "Visualizing Ontologies with VOWL". In: *Semantic Web* 7.4 (2016), pp. 399–419. DOI: 10.3233/SW-150200. URL: http://dx.doi.org/10.3233/SW-150200 (cit. on p. 21).

[LPSS16] D. Lembo, D. Pantaleone, V. Santarelli, D. F. Savo. "Eddy: A graphical editor for OWL 2 ontologies". In: *25th International Joint Conference on Artificial Intelligence, IJCAI 2016*. Vol. 2016. AAAI Press/International Joint Conferences on Artificial Intelligence. 2016, pp. 4252–4253 (cit. on p. 21).

[Mar17] MariaDB. *MariaDB 10.2.7 Release Notes*. July 12, 2017. URL: https://mariadb.com/kb/en/mariadb-1027-release-notes/ (visited on 02/28/2021) (cit. on p. 28).

[Mus15] M. A. Musen. "The protégé project: a look back and a look forward". In: *AI Matters* 1.4 (2015), pp. 4–12. DOI: 10.1145/2757001.2757003. URL: https://doi.org/10.1145/2757001.2757003 (cit. on p. 20).

[Nc20] Nobatek/INEF4, contributors. *flask-smorest: Flask/Marshmallow-based REST API framework*. 2020. URL: https://flask-smorest.readthedocs.io/en/latest/ (visited on 02/27/2021) (cit. on pp. 25, 29).

[npm21] npm, Inc. *npm*. Feb. 2021. URL: https://www.npmjs.com (visited on 03/17/2021) (cit. on p. 23).

[OAI18] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, U. Sarid, eds. *OpenAPI Specification*. Version 3.0.2. OpenAPI Initiative. Oct. 8, 2018. URL: http://spec.openapis.org/oas/v3.0.2 (cit. on pp. 23, 25, 29).

[Ope14] OpenAPI Initiative. 2014. URL: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md (cit. on p. 28).

[Pep+00] S. Pepper et al. "The TAO of topic maps". In: *Proceedings of XML Europe*. Vol. 3. Apr. 11, 2000, p. 77. URL: https://ontopia.net/topicmaps/materials/tao.pdf (cit. on p. 21).

[Poo17] A. H. Poole. ""A greatly unexplored area": Digital curation and innovation in digital humanities". In: *Journal of the Association for Information Science and Technology* 68.7 (2017), pp. 1772–1781. DOI: 10.1002/asi.23743. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.23743 (cit. on p. 7).

[Ric18] D. Richard. *mindmaps*. 2018. URL: https://github.com/drichard/mindmaps (cit. on p. 24).

[Ron] A. Ronacher. *Flask*. Pallets. URL: https://flask.palletsprojects.com/ (visited on 02/21/2021) (cit. on p. 25).

[SFT+20] A. Stellato, M. Fiorelli, A. Turbati, T. Lorenzetti, W. van Gemert, D. Dechandon, C. Laaboudi-Spoiden, A. Gerencsér, A. Waniart, E. Costetchi, J. Keizer. "VocBench 3: A collaborative Semantic Web editor for ontologies, thesauri and lexicons". In: *Semantic Web* 11 (2020). 5, pp. 855–881. ISSN: 2210-4968. DOI: 10.3233/SW-200370. URL: https://doi.org/10.3233/SW-200370 (cit. on p. 21).

[SHD+17] K. Swiber, T. Howard, M. Dobson, N. D. M. (grandcentrix), V. Tsukur, Adam, S. Ward, ramirahikkala, M. Raynham, javascript-journal, D. Beattie, D. Barnes. *kevinswiber/siren: Siren v0.6.2*. Version v0.6.2. Apr. 2017. DOI: 10.5281/zenodo.556783. URL: https://doi.org/10.5281/zenodo.556783 (cit. on p. 30).

[SQL] SQLAlchemy. *The Python SQL Toolkit and Object Relational Mapper*. URL: https://www.sqlalchemy.org (visited on 02/21/2021) (cit. on p. 25).

[W3C12] W3C OWL Working Group, ed. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. Dec. 11, 2012. URL: https://www.w3.org/TR/2012/REC-owl2-overview-20121211/ (cit. on p. 20).

All links were last followed on March 18, 2021.

A

Search results for the search term “ontology tool”

The search was performed with DuckDuckGo.

1. https://protege.stanford.edu
2. https://en.wikipedia.org/wiki/Tool_for_Ontology_Development_and_Editing_(TODE)
3. https://www.intechopen.com/online-first/taxonomy-and-ontology-management-tools-a-general-explanation
4. https://www.researchgate.net/post/What-is-the-best-ontology-editor-for-the-beginner
5. https://www.w3.org/wiki/Ontology_editors
6. https://www.mkbergman.com/862/the-sweet-compendium-of-ontology-building-tools/
7. http://www-legacy.geneontology.org/GO.tools_by_type.visualization.shtml
8. https://en.wikipedia.org/wiki/Ontology_engineering
9. http://geneontology.org/docs/tools-overview/
10. http://geneontology.org

Search results for the search term “owl ontology tool”

The search was performed with DuckDuckGo.

1. https://protege.stanford.edu
2. https://www.w3.org/OWL/
3. http://owlgred.lumii.lv
4. https://en.wikipedia.org/wiki/Web_Ontology_Language
5. https://www.academia.edu/11102376/Rabbit_to_OWL_Ontology_Authoring_with_a_CNL_Based_Tool
6. https://www.cambridgesemantics.com/blog/semantic-university/learn-owl-rdfs/owl-101/
7. http://geneontology.org/docs/download-ontology/

8. https://www.researchgate.net/post/What-is-the-best-ontology-editor-for-the-beginner
9. https://www.mkbergman.com/862/the-sweet-compendium-of-ontology-building-tools/
10. https://www.w3.org/wiki/Ontology_editors

Search results for the search term “form generator”

The search was performed on https://www.npmjs.com.

1. https://www.npmjs.com/package/@jsonforms/vanilla-renderers
2. https://www.npmjs.com/package/@jsonforms/core
3. https://www.npmjs.com/package/@jsonforms/react
4. https://www.npmjs.com/package/react-rrule-generator-offix
5. https://www.npmjs.com/package/react-rrule-generator-customo
6. https://www.npmjs.com/package/@jsonforms/material-renderers
7. https://www.npmjs.com/package/@jsonforms/angular
8. https://www.npmjs.com/package/vfg-field-object
9. https://www.npmjs.com/package/@jsonforms/angular-material
10. https://www.npmjs.com/package/vfg-field-array
11. https://www.npmjs.com/package/@ceecko/form-generator
12. https://www.npmjs.com/package/vue-form-generator-element-gt4w
13. https://www.npmjs.com/package/@qu-beyond/vue-json-schema-form-generator
14. https://www.npmjs.com/package/ui-schema-generator
15. https://www.npmjs.com/package/@roma219/vue-jsonschema-form
16. https://www.npmjs.com/package/json-form-generator
17. https://www.npmjs.com/package/@ui-schema/ui-schema
18. https://www.npmjs.com/package/@toolkip/form
19. https://www.npmjs.com/package/roosky
20. https://www.npmjs.com/package/jsonforms-core

B

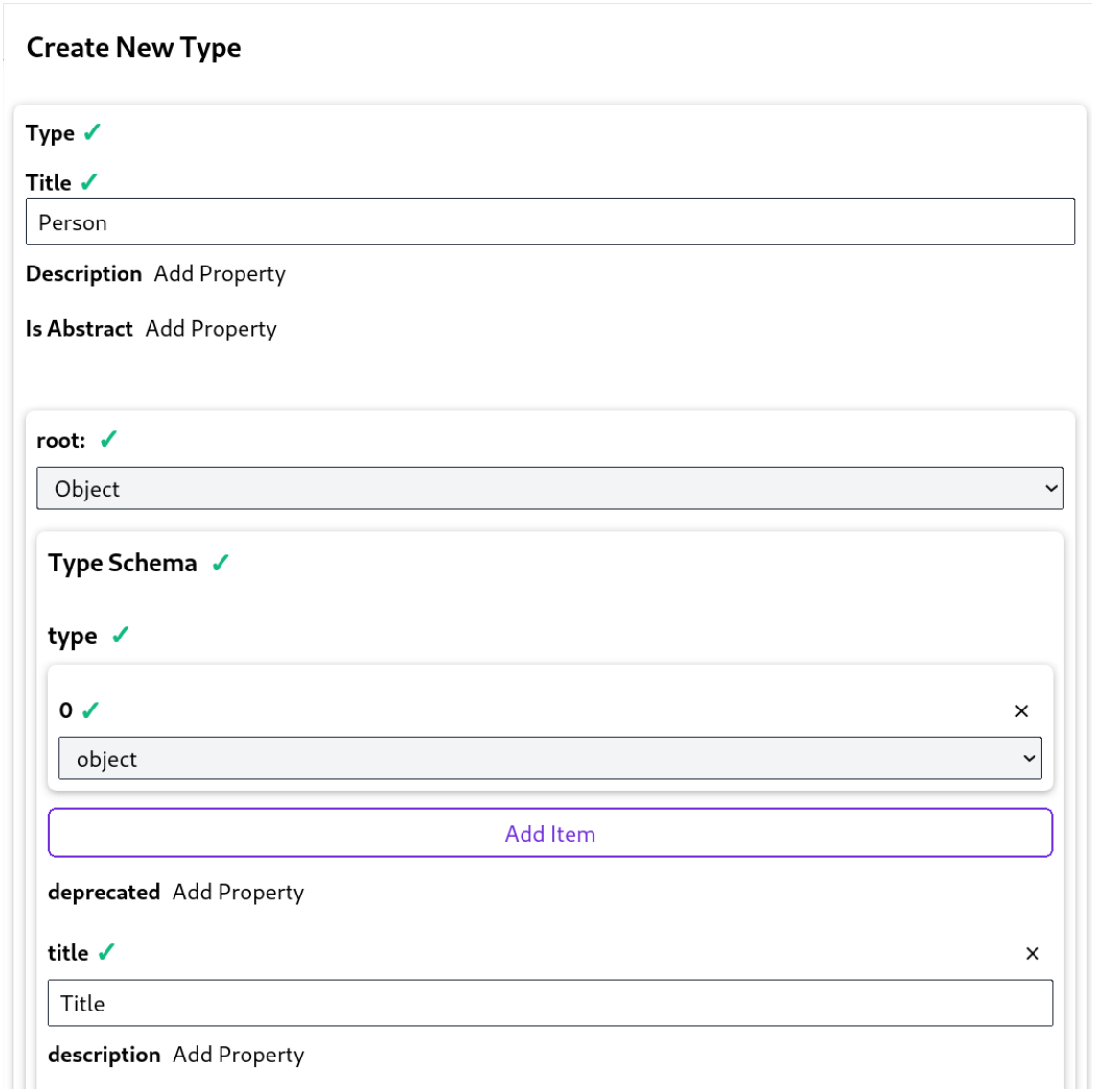

Figure B.1: The full view of the type editor for the type “Person” used in the example use case for the evaluation. Part 1/4: At the top are the properties for the type person. The card with the heading “root:” is the main schema of the type.

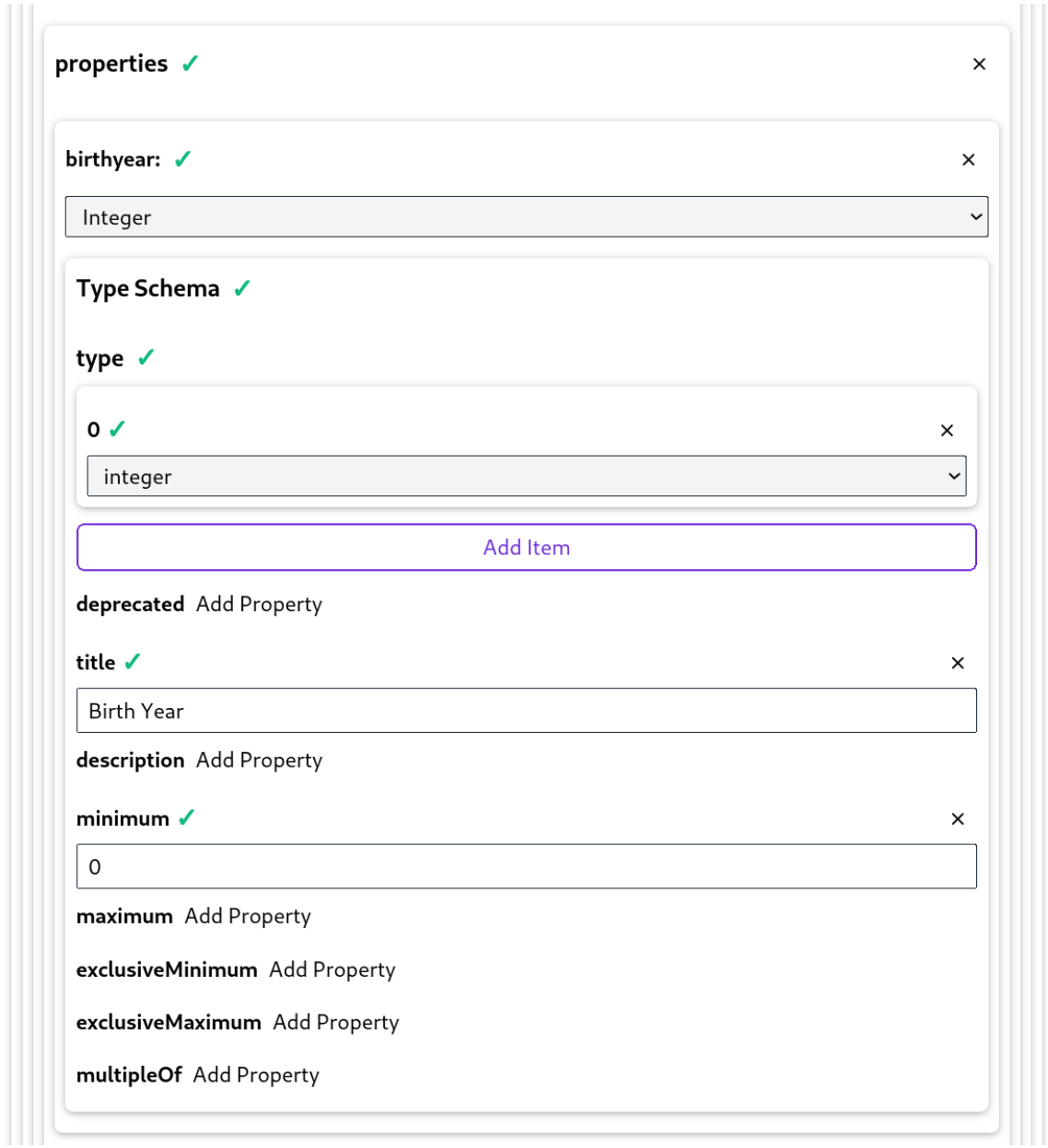

Figure B.1: The full view of the type editor for the type “Person” used in the example use case for the evaluation. Part 2/4: The properties of the root schema. This part shows the first property, that is for the birth year.

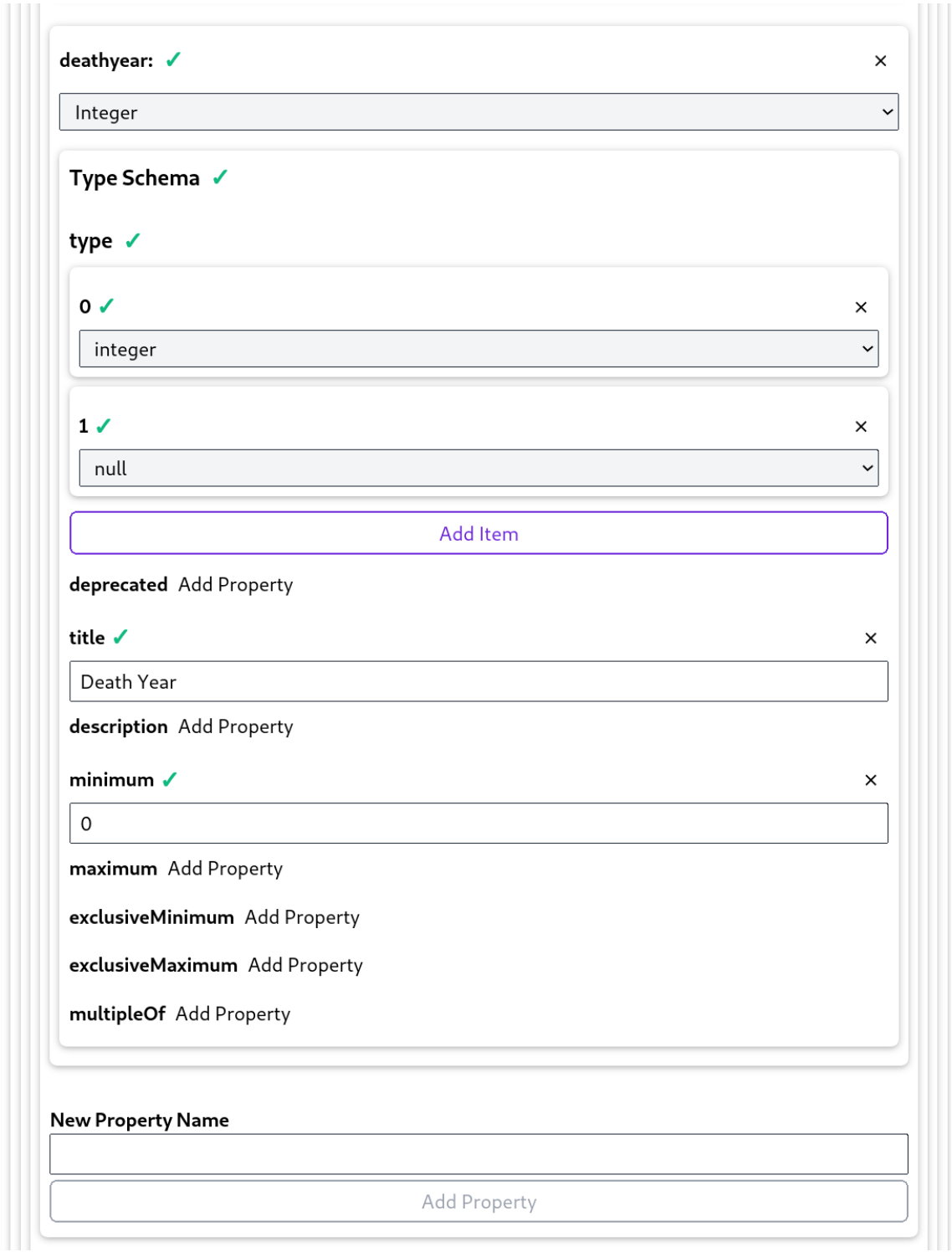

Figure B.1: The full view of the type editor for the type "Person" used in the example use case for the evaluation. Part 3/4: The properties of the root schema. This part shows the second property, that is for the death year. At the bottom are the UI elements to add new properties.

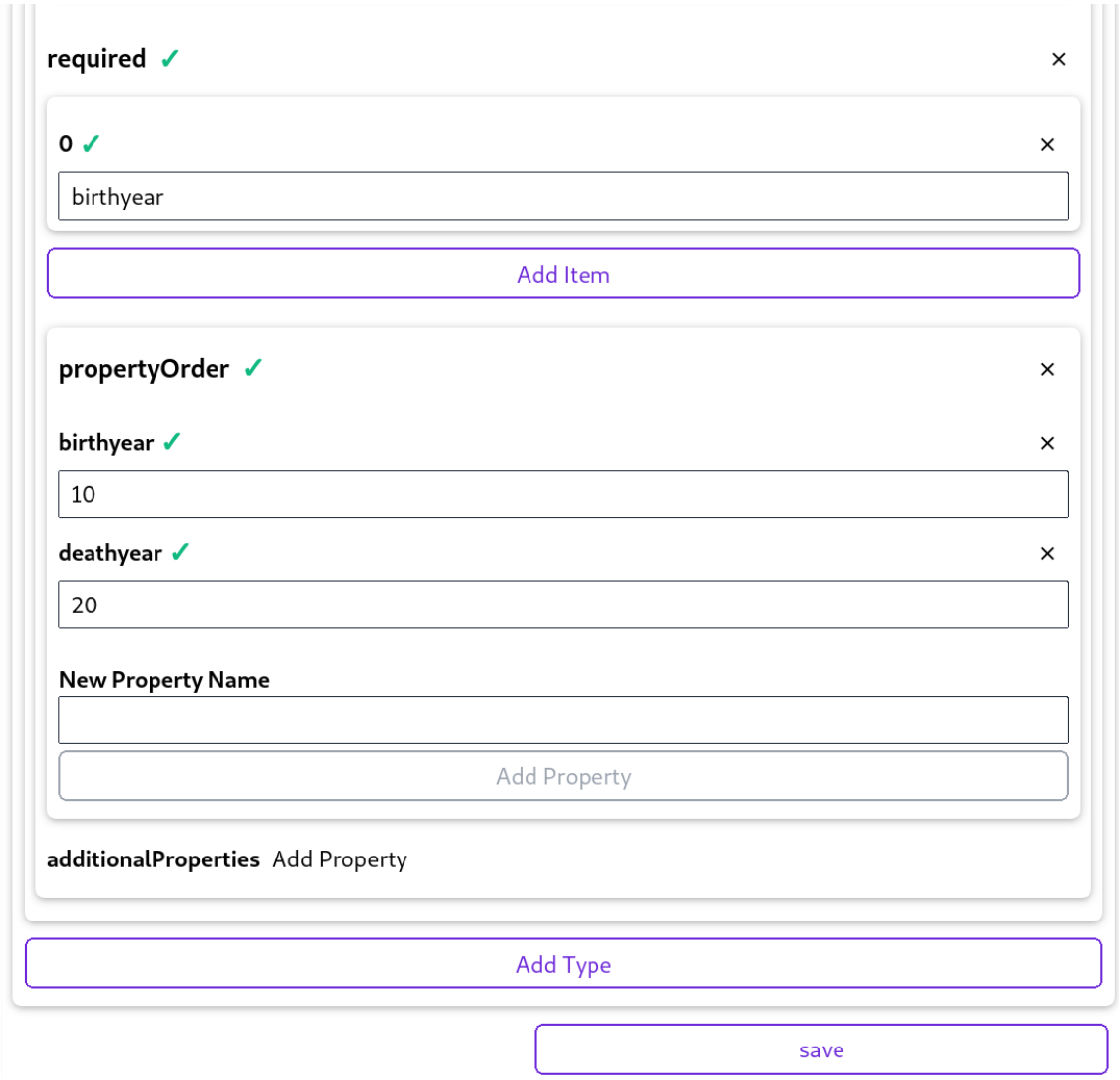

Figure B.1: The full view of the type editor for the type “Person” used in the example use case for the evaluation. Part 4/4: The required properties of the person type and the user definable property order.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature