Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Assisted Data Migrations for MUSE4Anything

Jan Weber

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Fabian Bühler, M.Sc.

Commenced:	July 16, 2024
Completed:	January 16, 2025

Kurzfassung

Beim Management von großen Datenspeichern bereitet vor allem die Datenpflege viel Aufwand. Typenschemata ändern sich über die Zeit aufgrund von sich wandelnden Anforderungen oder Fehlern während des eigentlichen Abstraktionsprozesses. Wenn sich solche Änderungen ergeben, müssen auch alle Objekte eines Typs aktualisiert werden, damit sie dem neuen Schema entsprechen. Andernfalls müsste ein Programm, das die Daten verwendet, jedes Objekt abhängig von der jeweiligen Version des Schemas anders behandeln. Die Anpassung aller Objekte an das neue Schema wird Datenmigration genannt. MUSE *(Muster Suchen und Erkennen)* und MUSE4Music sind spezifische Repositories für Kostüme beziehungsweise klassische Musik. Die Repositories sind für die Vorbereitung einer automatischen Analyse von kategorialen Daten gedacht. MUSE4Anything, Nachfolger dieser Implementierungen, ist hingegen ein generisches Datenrepository, in dem domänenspezifische Daten nach benutzerdefinierten Ontologien verwaltet werden können. Auch in dieser Umgebung können sich die definierten Typen, und die Anforderungen an diese, ändern. MUSE4Anything unterstützt das Ändern von Typen und Objekten bereits, jedoch gibt es bis jetzt keinen Mechanismus der die Daten automatisch migriert nachdem sich eine Typdefinition geändert hat. In dieser Masterarbeit wird MUSE4Anything um eine *migration engine* erweitert, die diese Prozesse ausführt. Durch die Automatisierung der Migration wird der nötige manuelle Aufwand deutlich reduziert, und die Benutzerfreundlichkeit, vor allem bei großen Datenmengen, verbessert.

Abstract

Maintaining data in large repositories requires a lot of effort. Type schemas change over time due to changing requirements or mistakes in the initial abstraction process. When these changes happen, objects of that type need to be updated as well to ensure they conform to the new schema. Otherwise, a program using the data would have to treat every object differently, depending on the schema version. The transformation of all objects to conform to a new schema is called data migration. MUSE (Muster Suchen und Erkennen, eng.: Search and Recognize Patterns) and MUSE4Music are specific repositories for costumes and classical music. These repositories are used to prepare categorical data for automatic analyses. MUSE4Anything, successor of these implementations, is a generic data repository that allows managing domain specific data conforming to user-defined ontologies. In this environment, defined types and their requirements may also change over time. While type and object modifications are already supported in MUSE4Anything, there is no implementation of automatic migration of objects after a type update yet. This thesis aims to extend the MUSE4Anything system with a *migration engine* that performs these processes. By automating data migration, the required manual effort is significantly reduced, and the user experience, especially for huge repositories, is improved.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Managing large amounts of data is an essential task in today's digitized world. This often includes abstracting real world observations into models and defining data types that represent these observations in the real world. Especially in object-oriented programming, the concepts of types and objects is well-known. Defining the schema of types, i. e., abstractions of real world entities, is a difficult task. On the one hand, it requires selecting a finite set of attributes to describe a model which could prove to be incomplete or erroneous. On the other hand, real world entities might change their nature over time which would also require changing the defined models. When a model, i. e., a type, changes, it is also necessary to think about objects of that type as they still conform to the old schema. As the update of a type might change the accepted structure of data, it is necessary to validate and update data after the type change. This process of transforming data from an old structure to a newly updated version, is called data migration. It has been an important topic of research for a long time [BMR11; RB01] and continues to be relevant to this day [HSHP21].

MUSE4Anything [Büh21] is a digital humanities tool to manage domain specific data. It allows describing domains by user-defined taxonomies and types based on JSON Schema. With these definitions, MUSE4Anything automatically creates entry forms to manage a data repository which enables the input and management of objects that confirm to these defined rules. With the user-defined taxonomies and types, MUSE4Anything provides a very generic data repository that can be built to work with different ontologies without having to implement a new repository for a new ontology. This is what sets MUSE4Anything apart from MUSE [BFL18] and MUSE4Music [BBE+16; Büh17; EBH17]. These are earlier implementations of the *Muster Suchen Und Erkennen, eng.: search and recognize patterns*, approach [Bar18] that deals with finding patterns in a domain by using machine-learning approaches on large data repositories.

In the generic nature of the MUSE4Anything repository, type definitions can change as well. Errors in the abstraction process or changes of the requirements, i. e., changes of the real world, can always happen. Therefore, it is important to include mechanisms that deal with the necessary changes after a type definition is changed. First, the type update needs to be validated and saved. Afterwards, the type update has to be applied on all objects of that type which includes checking whether they are still complying to the defined schema and saving the updates. MUSE4Anything already includes versioning on ontology and object level. This means, it is possible to perform and track ontology and data changes, i. e., performing and tracking modifications of types and objects. While handling these changes manually using editing forms is already possible in MUSE4Anything, there is no process to automate the migration of data following a type update.

Under these circumstances, data migration in MUSE4Anything faces a significant challenge. The system is built to deal with large data repositories, including the use of machine-learning methods. Although migrating data after type changes manually achieves the objective, it becomes increasingly labor-intensive and impractical for those large repositories. Therefore, a new mechanism needs to be introduced that updates the objects of the modified type. Changing the structure implies having

to check whether old objects are in conflict with the new change. Consider changing the data type from string to integer, which requires migrating data entries to an integer. In this example, it is also important to validate that the given string is numeric and can be updated to an integer. This process of automatic data migration is not yet part of MUSE4Anything, i. e., it is necessary to implement this match of older objects to the new schema which also includes checking the validity of type updates before they are saved to the database.

This thesis aims to significantly reduce the amount of user effort necessary when performing ontology changes in MUSE4Anything. In order to achieve this reduction of required user effort, the migration process needs to be automated as much as possible. Furthermore, an automatic approach would be much quicker in updating the data. Especially for large data repositories, the update process would become infeasible if done manually. When a data type is updated, processes confirming that the update is valid, are required. Once the schema update is validated, it is necessary to migrate the objects of that data type and to confirm that all objects follow the new schema and potentially apply changes if not. Objects that contain data which cannot be correctly migrated automatically, should not be migrated. Instead, they need to be presented to the user for a manual intervention. The migration process needs to be automated as much as possible to reduce time spent on the update, mistakes and the required effort. The migration engine presented here aims to overcome these challenges and extend MUSE4Anything with a new package that allows migrating data automatically.

Structure of the Thesis

The thesis is structured as follows. First, the fundamentals of this work are explored in Chapter 2. With the basics set, the design and the requirements for the extension of MUSE4Anything are elaborated in Chapter 3 with the help of use cases.

After having provided theoretical foundations, a design idea, and the requirements, the implementation of data migration in MUSE4Anything is introduced in Chapter 4. The implementation of the data migration and schema validation algorithms will be highlighted, as well as the integration of the migration engine into the MUSE4Anything system.

Chapter 5 gives an evaluation for the implementation, where the defined use cases are used to show the process of data migrations as it was realized and general requirements are checked. The evaluation will demonstrate how type updates and the subsequent data migration work from the user's perspective. Afterwards, Chapter 6 presents and compares work that has tried to solve similar problems as those stated in this thesis. Finally, Chapter 7 contains a summary and provides an outlook to possible extensions and improvements of the migration engine.

2 Background

This chapter provides the fundamentals of the extension of MUSE4Anything. MUSE, JSON Schema, and the data migration process will be explained in detail, to provide the necessary context for the design choices during development. Furthermore, the importance of examining the migration process will be explained, highlighting what makes it unique in the context of MUSE4Anything.

2.1 MUSE4Anything

MUSE (Muster Suchen Und Erkennen, eng.: search and recognize patterns) [Bar18; BLSW12; SBLE12] is a method to recognize patterns from data in a domain. Machine-readable data is used to build and analyze data repositories. MUSE uses these analyses to learn about patterns in the provided data which can be used in digital humanities. Initially, the scenario was exploring costume patterns in movies with MUSE [BFL18]. Later followed the application of the same method on classical music in MUSE4Music [BBE+16; Büh17; EBH17], which required a new implementation of the database. The approach of defining and developing a new data repository for each domain was deemed not practical. Due to the great effort needed to adapt these specific data repositories to different domains, interest in a more general solution arose. This is what MUSE4Anything aims to improve significantly.

MUSE4Anything [Büh21] is a generic data repository that allows to store user-defined ontologies. With that, it enables the user to define a domain, without relying on domains defined by the developers. This gives a lot of flexibility in which context the system can be used, and, unlike the earlier programs MUSE or MUSE4Music, is not limited to a single domain. MUSE4Anything supports the user in populating the repository with data by automatically creating input forms that align with the defined ontology.

In the context of MUSE4Anything, the terms *ontology* and *taxonomy*, have already been mentioned before and are essential for understanding the system's functionality. According to Furrer [Fur14], an ontology is a formal representation of knowledge within a specific *domain*. This formalization involves the *concepts* of the domain, their interrelations, and the attributes associated with them. Additionally, it includes domain-specific rules that define the behavior of entities within that domain. A domain represents a well-defined subset of the world, where concepts are the foundational components that constitute the domain.

In MUSE4Anything, an ontology is the formal description of the application scenario, i. e., the domain it is used for and all of its attributes. Taxonomies in MUSE4Anything are used to define the accepted values for these attributes. Beyond that, taxonomies provide a hierarchical structure that is readable by both humans and machines, and they allow defining relations between the attributes. Together, these concepts allow describing the domain in which MUSE4Anything is used.

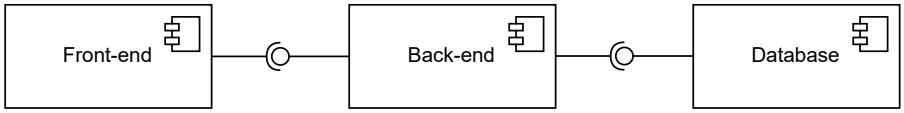


Figure 2.1: Simplified client-server architecture of MUSE4Anything

Another important part of constructing the ontology are *types*. Types usually are an abstraction of real world entities, e. g., one could think of the type *Person* defined by a *first_name* and *last_name*. While this type does not capture a person in its entirety, it provides a useful abstraction in certain scenarios, e. g., in employee management. It is also possible to create instances of that type which are called *objects*. The data stored in these objects is required to follow the restrictions that are defined in the type schema. For the *Person* schema from before, an example object could be defined as *first_name='John', last_name='Doe'*.

MUSE4Anything is built as a client server application as shown in Figure 2.1. The front-end is realized with the Aurelia framework [20] and uses an HTTP API to communicate with the back-end. The API in MUSE4Anything implements the concepts of hypermedia as the engine of application state (HATEOAS) [Fie08]. This ensures following the rules for a REST API, focusing on not being dependent on specific URIs. The client does not have any knowledge about the interaction with the server hard-coded, but instead, the server provides options on every request. The back-end is a Flask application [24b] written in Python. Communication between back-end and the database is handled by the SQLAlchemy ORM [24c]. The database is the central storage of all data, which makes the back-end server stateless. Consequently, user-defined types and objects are also stored in the database, with their definitions relying on JSON Schema, which will be explained in more detail shortly.

MUSE4Anything offers input forms to create and edit types, or to create and edit objects of a specific type. As the ontology is not defined from the beginning, these forms need to be created dynamically. It is important to note that the type schemas are not represented directly in the structure of the database. MUSE4Anything uses mostly SQL features with aspects from document-oriented databases for schemas and object data. Every type is stored in a database with a reference to the current version of this type. The version holds the actual schema defined for a specific type version. This is essential for the versioning system of MUSE4Anything, which stores the history of the type schema versions, and allows going back to older type definitions easily. Objects are versioned in the same way, and it is only the object version that holds the actual data. For older object versions, the type versioning is necessary, as they are likely to point to older type versions as well. Editing a type or an object results in a new version of that entity being created and stored in the database. Afterwards, the type's or object's pointer to the current version will be updated to match the newly created version that includes the last changes.

MUSE4Anything has views for all defined types and objects, enabling users to explore the entire ontology, or specific sub-sets of it using filters. The object list for example can be filtered by object type and to also include deleted objects. To analyze the data stored in the generic data repository of MUSE4Anything, additional tooling is required. The Quantum Humanities Analysis (QHAna) tool [KBW+21] is such a tool that offers functionality to analyze categorical data. Therefore, using MUSE4Anything in QHAna could be interesting in the future.

Listing 2.1 Example integer type definition in MUSE4Anything

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "$ref": "#/definitions/root",
    "title": "IntegerType",
    "description": "A description\n",
    "abstract": false,
    "definitions": {
        "root": {
            "title": "IntegerDefinition",
            "description": "An integer is stored here\n",
            "type": ["integer"],
            "minimum": 0,
            "maximum": 99,
            "multipleOf": 3
        }
    }
}
```

2.2 JSON Schema

JSON Schema [WA18] is a declarative language that can be used to describe and validate the structure of JSON (*JavaScript Object Notation*) [Bra17] data, i. e., it is possible to define the way a JSON document should look. JSON Schema itself uses JSON. Among other things, it is possible to define the type of data, e. g., a string or an integer, set requirements, or default values. It is possible to validate data against a JSON schema, based on the restrictions defined in that schema. As of now, there are five JSON Schema drafts available for this validation that support different features and keywords to define the structure of JSON data. Draft 07 [WA18] is the one used in MUSE4Anything. While newer drafts support conditions in schemas, these features are not supported in MUSE4Anything as it would require more development effort. Other possible keywords of JSON Schema are `anyOf` and `oneOf` used to combine schemas. Due to the complexity of automatically creating a user-friendly UI for them, they are currently not supported by MUSE4Anything. There are plans to support `allOf`, and some of the functionalities are implemented already. However, this support is currently disabled on purpose for user-generated types. This is due to the complexity of presenting the combination of generic schemas in a clear way.

Listing 2.1 shows an example type definition with JSON Schema as it is used in MUSE4Anything. This example is used to explain some of the possibilities of JSON Schema. The first keyword is `$schema`, and it defines the JSON Schema version, in this case Draft 07. Next, `$ref` is a reference to `#/definitions/root`, which is essential for making the example a functioning JSON Schema. This reference integrates the definitions from the referenced field, resulting in a valid JSON Schema. The `#/` in that reference indicates that it is a local reference, i. e., it is is the same schema definition. Furthermore, the optional fields `title` and `description` field are defined to set the title of a type and to provide additional context in human language. The `abstract` field is an addition of MUSE4Anything that can be used to define an abstract type which prevents objects being created directly from it. Finally, we see the `definitions` element which specifies the structure of the type. More precisely, it is located in `root`, as indicated by the earlier reference. Again, `title` and `description` are part of the definition of the type. The deciding field is `type`, specified as

`["integer"]` here. JSON Schema allows defining a list of types which is used when the schema should accept various types. MUSE4Anything uses that possibility to enable *nullable* types by adding `"null"` to the list. When a type is nullable, objects of that type can be empty, i. e., not hold any data. The example shows three restrictions that objects of the given integer type need to follow, `maximum`, `minimum`, and `multipleOf`. Looking at the given values, objects of that type need to be integers greater or equal to zero and less or equal to 99. Furthermore, the value needs to be a multiple of three. There also exist exclusive maxima and minima for integers. Other types have different keywords available which are defined in the specification of the draft [WA18]. According to the limitations stated in the type definition, objects of these types can be created. All objects have a `name`, `description`, and `data` field. The look of the `data` field depends on the type definition, and is, therefore, determined by user.

In that context, it is also important to mention that MUSE4Anything uses `$ref` to implement references to other types, called *schema references*. This approach allows reusing existing type definitions in a new type schema. By using the elements of JSON Schema this way, the effort required to define structures that build upon other types can be significantly reduced. Required types are referenced and do not need to be implemented again. These references can either be *local*, i. e., defined within the same schema, or *remote*, pointing to a different type schema.

Using JSON Schema allows improving the interoperability. For example, by defining the structure of JSON data in an API schema with JSON Schema, one can easily ensure that only appropriate data is accepted. Similarly, one can ensure that data fits to the desired table structure in a database. Data verification is what JSON Schema is used for in MUSE4Anything as well. Validating data against the schema definition of a type is essential to ensure data integrity.

2.3 Data Migration

Consider a scenario where the goal is to create a type schema for a person, similar to the example given before. In this case, persons are customers at a movie rental store. Initially, the type is designed to define a person by their first and last name. From this definition, instances are created to represent persons in the system, e. g., *first_name='John', last_name='Doe'*.

Later it becomes apparent, that the type does not include enough information as the age is necessary to account for age-restricted movies. The type schema is updated to incorporate the additional information. However, already existing instances, such as John Doe, are no longer consistent with the modified schema. This inconsistency needs to be addressed, meaning, already existing objects have to be mapped to the newest type version. The process of updating objects to conform to an edited type schema is referred to as *data migration*.

Like any abstraction, types which are models of real world entities, may require modifications to address design errors or changes in the real world entity [Rod95]. When type changes happen, it also means that the objects of that type might need to follow different restrictions. If that is ignored, problems may arise, e. g., when trying to access a newly defined field that is not part of an older, outdated object. It is desired that objects of a type follow the type's most recent schema definition which improves compatibility significantly.

It involves making sure that the change of the schema is valid, as well as defining the required operations to transform objects to an updated version. Finally, these operations also need to be executed on the existing objects. If the migration process runs successfully, no further effort is required. Otherwise, manual intervention might be necessary. An update that is considered invalid, e. g., because it includes an unsupported type conversion, is called *breaking update* [LXLZ13]. MUSE4Anything currently does not automatically migrate objects after type updates. As a result, a significant amount of effort is required to update every object manually.

Schema matching is a part of data migration and describes the process of comparing two schemas trying to find potential matches in their definitions. While adding a new field to a type as in the example above does not require complex matching, it is necessary for other changes, e. g., when the name of attributes change. Going back to the example from above, a more difficult change would be changing the attribute *first_name* to *first_names* to accommodate persons with more than one name. Here, already existing persons should be updated by matching the value of *first_name* to the new *first_names* attribute.

Schema matching has been an active research area for many years, and there exist many approaches to implement schema matching [BLN86]. By analyzing these approaches, their advantages and disadvantages, as well as their architecture, Rahm and Bernstein [RB01] could derive five classification criteria. The first criterion they worked out were *instance-based* and *schema-based* matching approaches. As their names suggests, instance-based algorithms perform updates based on data content, whereas schema-based algorithms do so based on the data schema. Next, the authors look at *element* and *structure* matching. The difference here is whether only elements like attributes of objects are considered, or also more complex combinations of those. Another differentiation is *language* and *constraint*. Schema matching using an approach that includes the language can for example map attributes based on their names, or based on word similarities. Constraint-based approaches are based on relationships between elements. Another criterion is the *matching cardinality*: During the mapping, an element of one schema could be mapped to exactly one or more elements in the other schema. Finally, some algorithms use *additional information*, which could be provided, e. g., in the form of user input, or from earlier schema matching runs.

3 Design

This chapter addresses the current state of data migration in MUSE4Anything, and elaborates on what is necessary to incorporate an automatic migration engine into MUSE4Anything. The requirements for migrating data and the expected functionality of a data migration engine will be highlighted to provide a foundation for the implementation. The chapter also includes three use cases to introduce the user's expectations for the migration process.

3.1 Problem Statement

In its current implementation, MUSE4Anything allows users to manually change type and object definitions. However, it does not automatically update an object to a new type version when the corresponding type definition is changed. As a result, objects retain outdated type versions after an update, causing potential compatibility issues between the old and new type definitions.

As of now, the type version of an object is updated to the most recent version when an object is manually edited. This means existing objects are still displayed using the type version they were created or last edited with. Therefore, unless an object is edited after every type update, intermediate type versions are skipped. This could increase incompatibility issues during updates to newer versions.

While in some cases, the update is trivial, and it is possible to modify the object in one step to conform to the new type schema, this process still requires manual effort. When the new type definition cannot be applied directly, e. g., because the structure of the type has changed significantly, it is necessary to edit the object twice because the new type definition cannot be loaded at first. With the first edit, the object's type version is updated. The new restrictions can only be seen upon a second edit operation, when the updated type definition is loaded and changing the object to the new schema is possible. Being able to handle large amounts of data is an important feature for MUSE4Anything. The manual approach requires a lot of user effort, is prone to errors, and impractical for large data repositories where automation plays a significant role for efficiency and correctness.

After explaining the need for a data migration engine, it is now time to plan the implementation of this extension, and how to integrate it into MUSE4Anything. As mentioned before, data migration in MUSE4Anything must be automated. There are two approaches to initiating the automatic migration process, automatically and manuallly. Starting the migration process automatically after a type update ensures consistency without any user effort. It ensures that objects are kept up-to-date and conform to the current type definitions. However, it also reduces the control of the user over the update process. This could become problematic in the process of undoing a type update due to an input error. Manual effort would be required to undo the migration of objects as well.

Alternatively, users could initiate the migration process manually, giving them more control over the repository. By delaying the migration of data until triggered explicitly, users can review type changes, and hold back an update until it is considered safe. Implementing manual initiation would be a significant improvement over the current state, as data migration is automated and the required effort remains small, regardless of the amount of data. Trusting the user to initiate the migration process could introduce problems if the user forgets to initiate the process. Assuming objects follow new type definitions when they do not, likely causes errors.

To decide which approach to choose, consider that the data migration process consists of two steps. First, it is necessary to validate the type schema change, and afterwards objects need to be updated to conform to the new type schema version. Since the validation of schema update checks for incompatibilities between the schemas, it was considered useful to initiate the migration process automatically to minimize the user effort and ensure data consistency.

Validity of a schema change depends on how the schema's data type changes. MUSE4Anything offers ten types, specifically, *array*, *boolean*, *enumeration*, *integer*, *number*, *object*, *resource reference*, *schema reference*, *string*, and *tuple*. In this context, validity means meeting the following requirements:

R 1.1 Primitive data types (boolean, integer, number, string) *must* be convertible to one another.

R 1.2 Primitive data types *must* be convertible to enums.

R 1.3 While transformations between primitive data types *must* aim to prevent data loss, some updates might still lead to it, e. g., an update from integer to boolean.

R 2.1 Enums *must* be convertible to other enums and the primitive types.

R 2.2 Enums *should* not be convertible to other data types.

R 3.1 Special instances of complex data types (array, tuple, object), e. g., arrays with one element or objects with one property, *should* be convertible to primitive data types and vice versa.

R 3.2 Arrays and tuples *must* be convertible to both arrays and tuples, but not objects.

R 3.3 Objects *must* be convertible to objects, but not arrays or tuples, and their properties *must* be matched deterministically.

R 3.4 If a property of a complex type (array, tuple, object) is changed in a compatible way, the complex type definition *must* also remain valid, e. g., changing the data type of array elements.

R 4.1 Schema references *must* be resolved to migrate data according to the referenced schema.

R 4.2 Resource references *must* be convertible only to resource references, see also Chapter 4.

R 5.1 There *must* be a check to validate the schema update before it is saved to the database.

R 5.2 If the schema update is validated, it *must* be saved to the database. Otherwise, the user *must* be informed about the invalidity.

R 6.1 All objects of a type *should* be migrated after a type update. Individual objects *should not* be migrated, if they cannot be mapped to the new schema, e. g., in the migration of strings to integers.

R 6.2 The migration engine *must* perform migration by creating a new object version.

To enable automatic data migration, it is necessary to check whether a schema change is valid. There are two different levels of modifications to consider in this regard. On the one hand, there are changes within the type restrictions, e. g., the maximum value of an integer could be changed from 64 to 128. These changes do not necessarily require a transformation of the data stored in an object as the type does not change. However, they still need to be monitored and may require action, as they could introduce difficulties in the migration process. In fact, automatic migration might become impossible because of logical contradictions between type versions, e. g., if a minimum of 50 is replaced by a maximum of 40. It would be impossible to update existing objects to the new type version based only on this information. Additional information in the shape of default values could be used to set data for existing objects and enable automatic data migration. However, default values are currently not supported by MUSE4Anything, and therefore, these updates should be considered invalid to avoid problems in the migration of existing objects.

On the other hand, there are changes of the type itself, e. g., a value was stored as an integer before and should now be stored as a boolean. If the type changes, a transformation of data is necessary, which can introduce difficulties and prevent automatic data migration. In these cases, it is not obvious how objects should be updated and therefore additional information would be required to enable automatic migration. For this, consider changing an object definition to a tuple definition. While this transformation could be useful in some scenarios, e. g., changing the structure of an address from an object to a tuple, it is very difficult to implement this correctly, especially finding the matches between object properties and tuple elements. This schema change is also invalid as it does not allow automatic migration without additional information which is not available in the current implementation.

Even if the schema update was considered valid in the first step, it might still happen that some objects cannot be migrated automatically. Going back to the example with integers, a type could introduce a maximum value when there was none before. Some objects might violate that restriction, which would require manual effort to fix. It is useful to introduce an overview on outdated objects to support the user in that process. This will be implemented as a filter that only shows objects whose type version does not fit to their type's current version.

After updating a type schema, all objects should be updated eventually. There are two approaches to perform this data migration. One option is to evaluate the update lazily which means, the migration is initiated once an object is actively accessed. Using this approach, one can avoid seeing outdated data, but it also introduces additional complexity. Every time an object is accessed, there would have to be a check of the type version. Running the migration only on access would also introduce delays when accessing an object. Alternatively, the migration is run for all objects after a new type version was saved. Ideally, the migration is run as a background process to ensure the system remains available. Since a big data repository would make the migration a long-running process, blocking the system for the entire time is not feasible. While this bares the risk of seeing old data when accessing an object, it is still deemed a useful approach to reduce the complexity and maintain availability. For that reason, updating objects in the background is a reasonable choice to realize data migration. In the context on running data analysis on the data repository, the runtime would increase significantly if it would be necessary to wait for the update of every object first. Therefore, the decision to initiate the migration of all objects is initiated automatically after a type update.

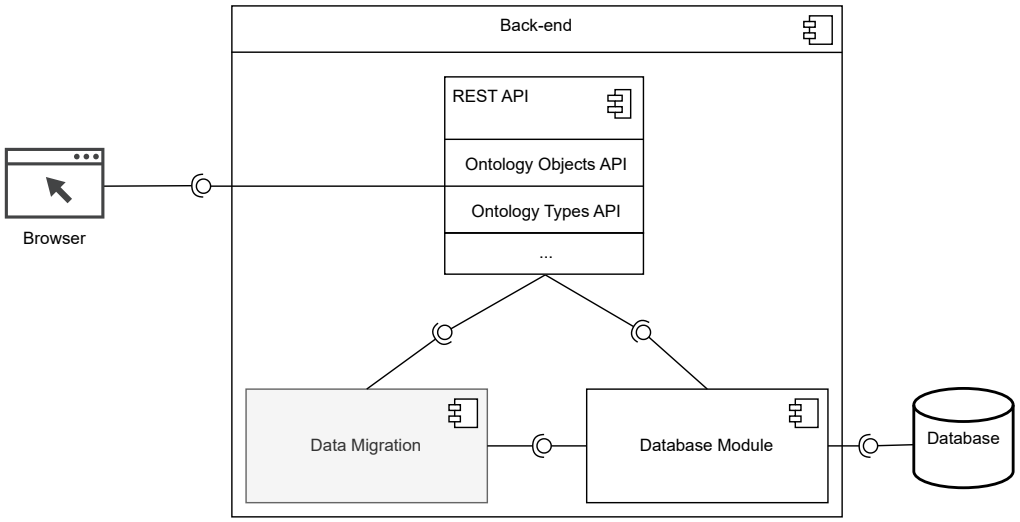


Figure 3.1: Integration of the data migration package (marked in gray) in MUSE4Anything

3.2 Architectural Overview

Figure 3.1 shows the data migration package will be architecturally integrated into the back-end of MUSE4Anything to avoid overloading the front-end with application logic. The external interfaces of the back-end with the front-end and the database are also indicated. The REST API, specifically those parts of the API that handle types and objects, will use the functions of the data migration package to perform the automatic processes of migrating data. As the modification of type schemas is handled through the API, it is useful to call the functions from there. The migration package uses the database module, which internally uses SQLAlchemy, to retrieve the necessary information for updates.

The data migration package encapsulates all required processes for data migration in MUSE4Anything. That way, the development can be realized independently of other packages. It consists of two parts, checking schema compatibility and the actual data migration. A function to check compatibility is used to confirm that a schema update is valid and supported by the migration engine, while the data migration function performs the migration of objects. The package is used when editing a type schema. With the inputs given by the user, a new type version is created that stores a new schema. This is handled in the object type API.

In the current state of the implementation, the method handling type updates accepts the data from a request and creates a new type version. After checking whether the new schema is indeed a valid schema, the new type version is then saved to the database. To enable migrations in MUSE4Anything, it is necessary to edit this method as it is exactly the point where type changes are realized. The adjusted logic of the method represented as a workflow is shown in Figure 3.2, where the required changes are marked as gray nodes. Initially, a validation procedure must be run to check whether a schema update is valid according to the requirements defined from before. A new object version will only be created if a change is considered valid. After saving the new version to the database, the migration of objects can be started. To do so, a background process should be

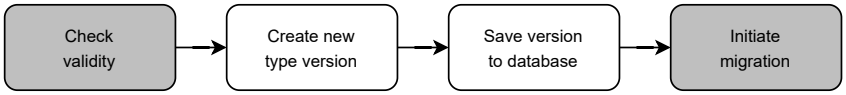


Figure 3.2: Updating a type in the back-end (new steps marked in gray)

initiated that also uses the methods provided by the migration package. Migrating all objects might be a long-running process, especially in large data repositories. Therefore, it is required to perform the migration as a background task that does not block the system.

3.3 Existing Tools

Data migration is not a new problem in data management [BMR11; RB01] and there exist different tools for data migration. A subset of these tools will be presented and compared next. They were chosen because of their well-defined and established foundations, and because they are still actively maintained. It is also necessary to highlight why they are not suitable in the scenario of MUSE4Anything.

First, tools that support users in database migration are presented. *Alembic* [24d] is a tool written by the developers of SQLAlchemy [24c] meant to support developers in the process of data migration when the structure of tables in databases changes. These structural changes happen when the database models in code using SQLAlchemy are changed. To use Alembic in a system, it is necessary to write migration scripts. These scripts define the steps that are necessary to update a database to a new schema. Optionally, it is also possible to define the reversal of the update for a potentially needed downgrade of the database. It also offers automatically generated migration scripts. However, these need to go through a manual revision to ensure correctness of the update. Alembic's automatic generation of migration scripts seems close to fulfilling the requirements for the extension. The idea is to automatically determine the necessary steps to update all objects of a specific type. In contrast to Alembic, however, a solution with as little manual effort in the schema matching process as possible, is required. *Flask-Migrate* [Gri24] is a tool to handle data migrations in Flask [24b] applications and is built upon Alembic. For that reason, Flask-Migrate carries the same problems as Alembic and cannot be used to perform automatic migration in MUSE4Anything.

Atlas [24a] is a new tool that attempts to solve one of the problems of Alembic, i. e., the need for the manual input when creating migration scripts. With Atlas, developers only define the target schema and the tool is able to create migration scripts automatically. Some complex changes might still require manual intervention. Apart from that, it has better support for collaborative efforts and supports various database systems. Fully automatic data migration is also desirable for MUSE4Anything.

However, in the case of MUSE4Anything, there is no change in the structure of the database when a type update happens. MUSE4Anything is constructed in such a way that the type schema is not represented in the database schema, but defined in an entry of the database. A type change is therefore not observable in the structure of the database, but instead it is necessary to compare the schema definitions saved in two type versions. For these two reasons, Alembic and Atlas are not applicable in MUSE4Anything and an alternative is needed.

There are also tools that focus more on comparing JSON Schemas, such as *jsonsubschema* [HSHP21]. This is a tool that enables *subschema* checking for JSON Schema. The authors define a schema A as subschema of schema B if the validity of all objects for A implies their validity for B. The tool runs on the JSON Schema and not on instance data, which enables checking for breaking changes before changing a schema. A subschema check could verify that an API update does not introduce incompatibilities to already existing API interactions. To do so, an update would only be accepted if the updated schema used in the interface is a subschema of the original version. This would ensure that requests of the old format remain valid even after an update, which makes it a relevant tool for data migration in MUSE4Anything. Preventing breaking changes is an essential part of making automatic migration possible.

That being said, breaking changes in MUSE4Anything differ from jsonsubschema, i. e., editing type definitions should be less restrictive than in subschemas. For that, consider an integer definition that defines a maximum value of 150, which is updated to a maximum of 100. While this would not be considered a subschema, it should be a candidate for automatic migration. Another example for a valid change that is not a subschema is adding new properties to an object definition. Even though some objects may not be suitable to be migrated automatically without more information, this is a valid schema update according to the requirements. For better support of these type of changes, default and cut-off values could be introduced in the future to migrate data in a more useful manner. While jsonsubschema provides a similar behavior, it is not directly usable in MUSE4Anything as it introduces too many restrictions. The tool provides a lot of information on comparing JSON Schemas which is useful for data migration.

Other tools that compare two JSON Schemas, are *json-schema-diff* [16] and *json-schema-compare* [Han17]. However, these tools are neither suitable for MUSE4Anything, nor maintained at this point in time. The first tool can be used to list differences between two schema versions, but it does not immediately help with migrating objects. Similarly, the second tool focuses on determining whether two schemas are identical. However, checking for equality is not useful in the case of MUSE4Anything, as type changes are likely to introduce modifications to the structure of the type as well. The need for a custom migration engine in MUSE4Anything is evident, and the implementation should include and build upon concepts from existing tools and algorithms.

3.4 Use Cases

The following three use cases are example scenarios in data migration. The first use case is designed to represent a commonly expected change of a schema. In contrast to that, the second one is made to cover many potential changes at once. While it is an atypical scenario for a user, it allows covering many requirements at once and in a clear structure. The last use case is an invalid schema change that needs to be communicated to the user. All use cases follow the general idea as it is presented in Figure 3.3. The user initiates type changes in the input forms provided by the UI. These modifications need to be validated by the migration engine before saving them. Only if they are valid, will the user be informed about the update and the migration of the objects of that type can be started as a background process. The user perspective only includes editing a type schema in the UI and seeing the changes accepted by the system. If the update is not valid, the user needs to be

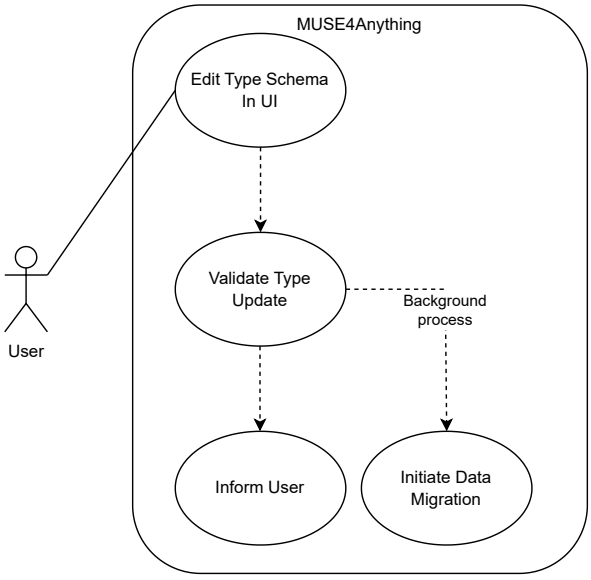


Figure 3.3: Use case diagram for changing a type schema

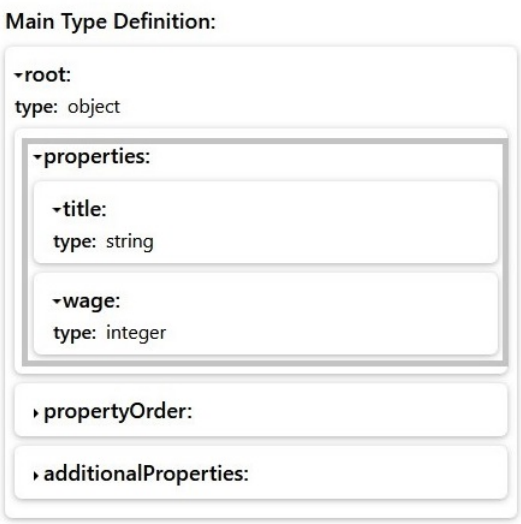


Figure 3.4: Type definition of the *JobType* Schema

informed about it. Apart from that, the required migration processes should be run independently of the user. The first two use cases reference the *JobType* schema which is shown in Figure 3.4 for reference. It is a simple object type with the two properties `title` (string) and `wage` (integer).

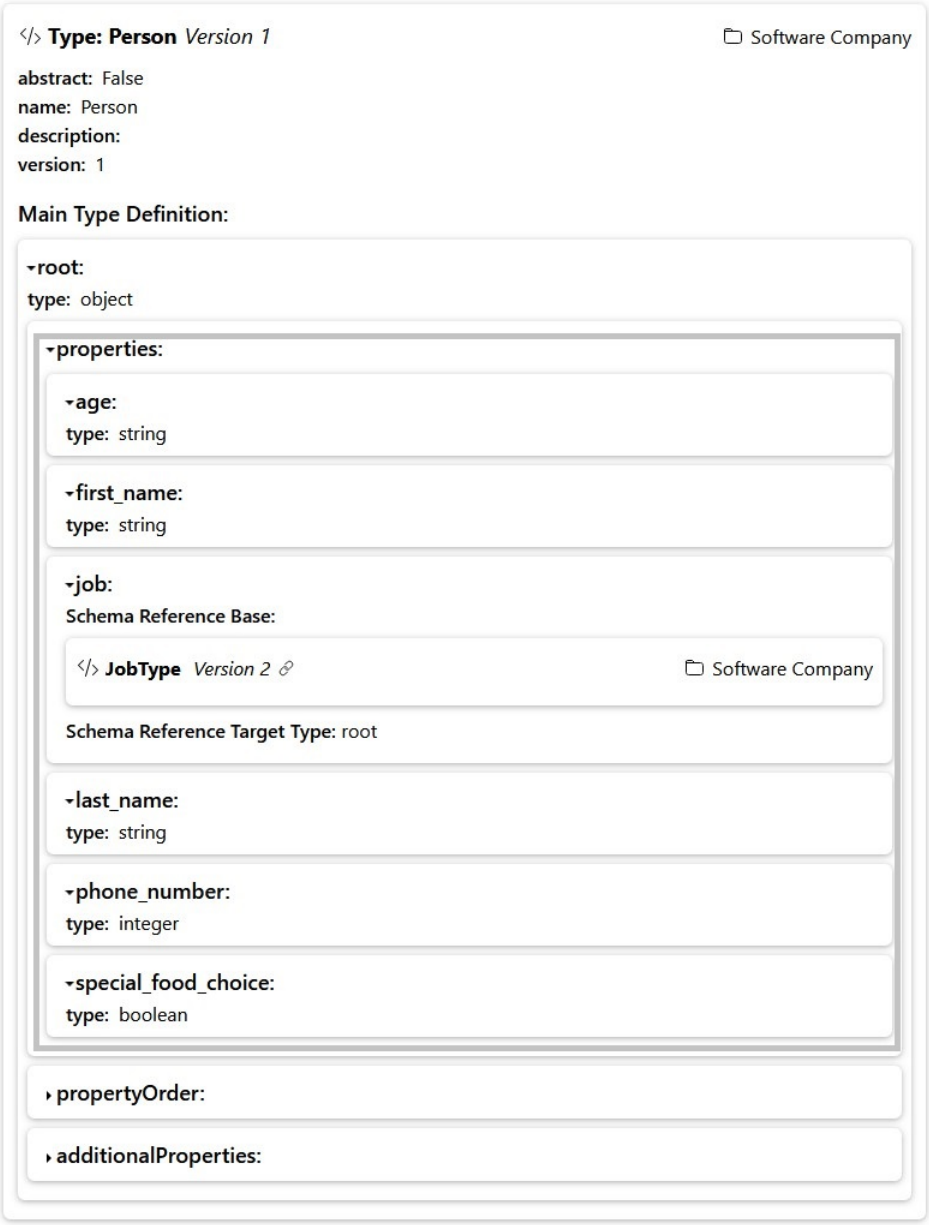


Figure 3.5: Initial version of *Person* schema as shown in MUSE4Anything

3.4.1 Use Case "Update Structure of Employee Data"

Assume that a company uses the *Person* schema shown in Figure 3.5 to manage employee data. *Person* is a JSON Schema object type that defines a list of properties, as shown in the gray box. On the one hand, these properties include general information about a person, such as `last_name`, `first_name`, `age`, and `phone_number`. On the other hand, some relevant employee data is incorporated as well, i. e., `job`, `special_food_choice`, and `department`.

In an annual check, the HR department of the company reviewed the data models and created a list of required changes:

- `age` was mistakenly stored as a string, and will be stored as an integer instead.
- `phone_number` was stored as an integer, and will be stored as a string instead to support better formatting options and international phone numbers in an easily readable way.
- `special_food_choice` was stored as a boolean, and will be stored as an enum to support more options such as vegan, gluten-free, and potentially more.

3.4.2 Use Case "Quality Control Test"

This use case is designed to showcase a large set of requirements for the migration engine. An object base type was chosen because it allows to easily define many properties of different types. Note that properties which store complex types, such as other objects, arrays, or tuples, are referenced by schema references. The initial and updated properties of this type schema are shown in Table 3.1. The following properties are changed:

- `bool_prop` was stored as a boolean, and will be stored as a string.
- `list_prop` was stored as a list of integers and will be stored as a list of strings.
- `number_prop` was stored as a number and will be stored as a boolean and renamed to `changed_prop`, i. e., values should be migrated to the new property definition.
- `string_prop` was stored as a string, and will be stored as an enum.
- `tuple_prop` was and will be stored as a tuple. However, the tuple definition will be changed from `(boolean, integer, string)` to `(boolean, string, string)`.

With these changes, we test a range of conversions. First, there are simple transformations, such as transforming boolean or integer to string. These simpler changes also happen in the migration of complex types, e. g., migrating list or tuple elements to strings. There is also a migration from string to enum. Finally, the property name change should be feasible as no other properties have been added or deleted. In that way, simple and complex, as well as nested schema changes are tested.

Table 3.1: Object properties in original and updated schema

Original Schema	Updated Schema
`bool_prop` (boolean)	`bool_prop` (string)
`int_prop` (integer)	`int_prop` (integer)
`list_prop` (list of integers)	`list_prop` (list of strings)
`number_prop` (number)	`changed_prop` (boolean)
`schema_ref_prop` *(JobType)*	`schema_ref_prop` *(JobType)*
`string_prop` (string)	`string_prop` (enum)
`tuple_prop` (tuple of boolean, integer, string)	`tuple_prop` (tuple of boolean, string, string)

3.4.3 Use Case “Invalid Schema Change”

This use case is used to ensure appropriate handling of invalid schema changes. The user should be informed if an update is not supported by the migration engine, and therefore considered invalid. The use case is made of the attempt to change an array schema to an object schema. As defined in R 3.2, arrays should be convertible to arrays and tuples, but not objects. It is expected that this schema update is prevented and not stored to the database. The user should receive a notification in the UI that indicates invalidity of the change.

4 Implementation

The implementation of the extension for MUSE4Anything follows the test-driven development (TDD) [JS05] method. In TDD, tests, and especially automatic tests, are written before the program logic is implemented. This highlights the importance of tests that validate the expected behavior in the implementation. By having tests of high quality from the beginning, newly introduced errors are more likely to be discovered early. It allows developing source code step-by-step satisfying more cases with every iteration. Using TDD can decrease the density of errors in code [JS05].

As explained in the previous chapter, data migration is implemented in a separate package. Internal details are not reflected in the interface offered by the module, so it can be modified without the risk of breaking the system [Sny86]. As part of the TDD strategy, the package responsible for automatic data migration was developed isolated from other components in MUSE4Anything. The package is implemented independently of the API and database at first. Only once the package performs the required migrations reliably, it is integrated into the rest of the system. Having an encapsulated module that offers a defined interface, makes developing with tests much easier as it can be tested and developed independently of other parts of the system.

An essential part of the implementation is dynamically determining the schema's type, i. e., one of the ten types available in MUSE4Anything. Since different types typically also have different keywords available, it is important to know the type of a schema when migrating data automatically. Determining the type in MUSE4Anything is not a trivial task and was therefore implemented in the migrations package. It is not only the type attribute of a schema that needs to be checked, but often, there are additional attributes that allow the definition of more distinct types. As an example, both tuples and arrays share the same type, i. e., array, but are distinguished by an additional keyword, `arrayType`. While arrays require all elements to be of the same data type, tuples allow storing elements of different types in a list. Even though the information is contained in the schema, having the exact type as a simple string, eases comparisons significantly. Additionally, the schema can include `"null"` in the type definition, making it nullable, i. e., objects of that type can be empty. As this is important information in the migration process, it should be extracted as well for simplified access. Therefore, the provided functionality extracts the type information stored in a schema and returns a tuple of the type as a string and also a boolean that indicates whether the type is nullable.

In the following part of the chapter, the implementation of the data migration engine is presented. The chapter will provide the necessary context to understand the implementation, and elaborate on the design process of the migration engine. It will highlight how data can be migrated, as well as the validation of schema changes, also using the presented use cases from Chapter 3.

Algorithm 4.1 Pseudocode for migrating data

```
function MIGRATE_DATA(data, source_schema, target_schema)
    source_schema ← resolve_references(source_schema)
    target_schema ← resolve_references(target_schema)
    source_type ← extract_type(source_schema)
    target_type ← extract_type(target_schema)
    if target_type == 'boolean' then
        updated_data ← migrate_to_boolean(data, source_type)
    else if target_type == 'enum' then
        updated_data ← migrate_to_enum(data, source_type, target_schema)
    else if target_type == 'integer' then
        updated_data ← migrate_to_integer(data, source_type)
    else if target_type == 'number' then
        updated_data ← migrate_to_number(data, source_type)
    else if target_type == 'string' then
        updated_data ← migrate_to_string(data, source_type, source_schema)
    else if target_type == 'array' then
        updated_data ← migrate_to_array(data, source_type, source_schema, target_schema)
    else if target_type == 'tuple' then
        updated_data ← migrate_to_tuple(data, source_type, source_schema, target_schema)
    else if target_type == 'object' then
        updated_data ← migrate_to_object(data, source_type, source_schema, target_schema)
    end if
    return updated_data
end function
```

4.1 Data Migration

Algorithm 4.1 provides an overview on how the migration functionality was realized. If any of the input schemas contain schema references, they must be resolved before continuing the execution. By doing so, it is possible to skip special treatment of references later, unless a schema is recursive, i. e., contains references to itself. Nested references will be resolved until a schema definition is found. The migration function will continue using the resolved schema, consisting of clearly defined types. After resolving these references in the schema, source and target types are extracted from the schema. Migrating data is oriented at the *visitor pattern* [PJ98], as the appropriate migration is chosen depending on the target type. The source type is given as an argument as the migration procedure needs to know how to interpret the original data. Some migrations may require the entire source or target schema, for better migration results, e. g., matching two tuples requires information about the elements' definitions, and migrating objects relies on the properties defined in the object type schema. When migrating data, it might be necessary to call this method again with a nested schema. To ensure that the system does not get stuck in an endless recursive schema, a depth counter is included. Depending on the target type, the parameters might have to be completed with additional information. In the following subsections, this will be highlighted and explained in more detail for the different types.

		Source Type		
Target Type	Conversion	Boolean	Numeric	String
Boolean	`bool(data)`	*None*	*Data loss*	*Data loss*
Integer	`int(data)`	*None*	*Data loss*	*Limited*
Number	`float(data)`	*None*	*None*	*Limited*
String	`str(data)`	*None*	*None*	*None*

Table 4.1: Consequences of the conversion of primitive types. *None* indicates that the conversion is reliable and works for all inputs. When a conversion causes *data loss*, not all information can be transferred to the new type, e. g., changing a number with decimals to an integer. Some conversions from strings are also *limited* to a sub-set of values, as not all strings are valid numbers.

Primitive Types

Transformations between primitive types (boolean, integer, number, string) are performed with Python's built-in type conversion utilities, as indicated in Table 4.1. Depending on the target type, different functions are used to transform data appropriately. Some migrations have side effects as data loss, while others only work for a subset of values, due to the different value spaces of the types. The conversion will be highlighted for each type next.

Boolean values are limited to `true` and `false`. They are expected to be transformable to other primitive data types. For integers, a common convention is to represent false as `0` and true as `1`. To improve compatibility when migrating to strings, the boolean value false is transformed to the empty string and true to `"true"`. There is no data loss caused by this transformation, as the other types offer an appropriate value space. Furthermore, assuming consistent use of the new value space, data can be reliably transformed back to boolean.

Integer and *Number* can be summarized as numeric types, even though the conversion function differs slightly. In MUSE4Anything, these data types are used to store numeric values, e. g., `3.14`. Further restrictions to the accepted values can be imposed using additional keywords that both these types support. The keywords are `minimum`, `maximum`, `exclusiveMinimum`, `exclusiveMaximum`, and `multipleOf`. Their purpose is self-explanatory, e. g., if multipleOf is set to `3`, only numbers that are multiples of three are accepted. The only difference between integer and number lies in the support of floating point numbers. Numeric types cannot contain another type definition.

Starting from a numeric value, converting it to another type can cause data loss, e. g., in the transformation of number to integer, as decimals would be cut off. Also transforming number to boolean is tied to data loss because many numbers would get mapped to the same boolean True, while only 0 is migrated to False. Therefore, after the migration, it is only possible to state whether a number was non-zero before. Numeric types are typically transformable to number and string without problems, unless the schema introduces new restrictions with the keywords.

String can save text values, e. g., `"Hello World"` or `"42"`. Migrating from string to another primitive data type is often only possible with data loss or limited to a small sub-set of strings. Looking at the two examples, the first one could not be migrated to one of the numeric types, while it would be possible for the second. The migration to boolean is not lossless, as it only determines whether there was a non-empty string before.

In contrast to the other primitive types, string allows a more flexible approach even when migrating to complex types. As a string can represent many things, even arrays or objects can be transformed to strings. A tuple `[True, 42, 1944, False]` can easily be transformed to a string. In this case, the migration would result in `"[True, 42, 1944, False]"`, i. e., exactly the object's data as a string.

Simple Instances of Complex Types

Consider *simple instances* as instances of complex types, such as arrays, tuples, or objects, with only one element or property, e. g., the list `["Hello World!"]`, or the object `{"int": 1944}`. These simple instances are interesting for the automatic data migration to primitive types. This could happen in the case of an object model that captures too many details. Deleting unnecessary properties in the first step, and then transforming the remaining property to a primitive data type in the next, allows redefining a schema. Migrating a complex type to boolean is handled differently, as it evaluates to `true` if there is an object, otherwise to `false`. This is the functionality of Python's conversion function which also is able to handle arrays and objects defined in MUSE4Anything. Note that for object schemas, the check if such a migration is valid depends on the schema only. For arrays and tuples it depends on the instances. The conversion from array or tuple to a primitive type was implemented to migrate only those instances that are simple instances. Therefore, additional effort may be required if some instances do not suit the new data type definition. This approach was chosen to avoid data loss, as it is possible to change back to the old type definition. Unless the migration involves data loss, it is easily possible to recreate the old object.

It is also interesting to highlight the other direction, i. e., from a primitive data type to a simple object. As an example, assume a value is currently stored as an integer. Over time, it becomes apparent, that the integer model is not sufficient, and it should be replaced by an object definition. If the schema is changed to an object type that initially only includes the integer value, it is possible to migrate the old data to the new format. In additional steps, more properties can be added to the schema to improve the model.

For both directions, the appropriate migration function must be called which takes the value stored in the initial data object and transforms it into the expected target type. For these transformations, the same limitations as elaborated above apply, e. g., when a string is saved in the property of an object and not all strings can successfully be migrated to integers. If a primitive type is transformed to a complex one, it is necessary to migrate the data to the expected data type, and create a list or object with the updated data.

Enumeration Type

Enum in MUSE4Anything can define booleans, numbers, strings, and `null` as accepted values. An example enumeration is provided in Listing 4.1 to show how they are stored in MUSE4Anything. This example accepts the string `"Hello World!"`, the boolean `true`, the number `1234.56`, and `null` as

Listing 4.1 Example enumeration definition in MUSE4Anything

```
{
"enum": ["Hello World!", true, 1234.56, null]
}
```

Listing 4.2 Example array definition and object in MUSE4Anything

```
// Example array definition
{
    "type": ["array"],
    "arrayType": "array",
    "items": {
        "type": ["boolean"],
    }
}
```

```
// Example array object
[
    true,
    false,
    true
]
```

values. Evidently, an enumeration in MUSE4Anything does not allow defining nested schemas either. Therefore, migrating to an enumeration means checking whether the data input conforms to one of the allowed values. For the given example, the string `"Hello World!"` is accepted and would be migrated to a new object version, whereas `"Hello Luna!"` would not be migrated. Numeric values also need to be exact matches. In this example, `1234` would not be accepted by the enumeration to ensure no accidental false transformations.

Array Type

Array defines the type of elements as nested schemas in the items keyword of the schema, see Listing 4.2. There is also an example for a valid array for the given schema. Note that the type of the elements can be one of the primitive types like boolean, but it can also be a complex schema, such as another array, referenced with a schema reference. Therefore, in the case of migrating to an array, a recursive call of the migration function is necessary that uses the nested schema definitions. In the provided example array, the nested schema is `"type": ["boolean"]`, i. e., a simple boolean value. Assuming the migration from an array that has boolean values to one that has string values, the recursive call is designed to match every element stored in the array to the new data type by accessing the nested schemas. Similarly, in the case of migrating to a tuple, there is a recursive call of the migration function, only that the target type of an element is defined differently for tuples.

Tuple Type

Listing 4.3 gives an example *Tuple* type schema and a possible instance. Tuples are also arrays, but distinguished from classical arrays by the `arrayType` keyword. In contrast to arrays, tuples accept different data types for each element. Therefore, `items` is defined as a list for tuples. In the given example, data must be a list of a boolean, a nullable integer, and a string. It is also possible to define `additionalItems`. In the example this means that the first three elements of data must conform to the given definition in items, but afterwards arbitrarily many booleans might follow.

Listing 4.3 Example tuple definition and object in MUSE4Anything

```
// Example tuple definition
{
    "type": ["array"],
    "arrayType": "tuple",
    "items": [
        {"type": ["boolean"]},
        {"type": ["integer", "null"]},
        {"type": ["string"]}
    ],
    "additionalItems": {
        "type": ["boolean"]
    }
}
```

```
// Example tuple object
[
    true,
    42,
    "Hello World!"
]
```

Algorithm 4.2 Pseudocode for migrating from tuple to tuple

```
function MIGRATE_TUPLES(data, source_schema, target_schema)
    source_tuple ← source_schema['items']
    target_tuple ← target_schema['items']
    i ← 0
    while i < length(data) do
        source_element ← source_tuple[i]
        target_element ← target_tuple[i]
        data[i] ← migrate_data(data[i], source_element, target_element)
        i ← i + 1
    end while
    return data
end function
```

In general, the migration process of tuples is similar to the one of arrays. However, the first n elements can have different types and must thus be handled as defined in `items`. A simplified version of matching two tuple schemas is provided in Algorithm 4.2, which only covers the case of tuples of equal length. Because tuples have a list of item types, it is necessary to iterate through the two lists in parallel and migrate each element of data according to the definitions in the lists. In every iteration, another element of data is updated, and, finally, the migrated data can be returned.

Object Type

Object needs a distinct migration approach as well. For context, an example object schema and instance is provided in Listing 4.4. Objects define a number of properties that can be of any type, e. g., integer or schema reference. To create an object, these properties must be set to fitting values, as seen in the example.

For the migration from an object schema to another object schema, it is necessary to compare the defined properties. This happens on a lexical level, i. e., properties with the exact same name and spelling are mapped to each other. The different properties have nested schemas. Therefore, the data stored in those properties can be migrated using those schemas in a recursive call of the

Listing 4.4 Example object definition and instance in MUSE4Anything

```
// Example object definition
{
    "type": ["object"],
    "properties": {
        "age": {"type": ["integer"]},
        "name": {"type": ["string"]},
        "happy": {"type": ["boolean"]}
    }
}
```

```
// Example object instance
{
    "age": 42,
    "name": "John Doe",
    "happy": true
}
```

migration function. That way, even if those nested schemas are complex schemas, the migration process can be executed normally. Changing the name of a property may be desired due to errors when developing a model. These changes are supported by deleting the old property and adding the new property with the corrected name in the same update operation [Zic91]. This is limited to one name change for every type update. By moving the deleted property to the new one, data can be migrated in this case. To avoid this behavior, in the case of adding a new, unrelated property to an object schema and deleting another property, those changes must be applied in two separate steps. However, allowing name changes of properties in the migration process can improve the quality of the data models and was therefore considered useful.

Resource Reference Type

Resource References are a MUSE4Anything-specific feature. They allow defining references to taxonomies or types that are defined in the repository. Objects of these resource references are pointers to resources in the repository, e. g., an already existing taxonomy, or an object that stores an integer. An example definition of a resource reference schema for a type schema and a possible object of that type are given in Listing 4.5. As the example shows, resource references are also defined as objects, however they are distinguished from other objects by the `customType` keyword. Furthermore, they define the required properties `referenceType` and `referenceKey`, which are used to indicate that objects of the resource reference schema need to point to an `ont-object` and define the `referenceKey` appropriately. The schema itself also defines values for those two keywords. However, for the type definition, they do not reference an object, but a type using its `namespaceId` and `typeId`. That way, it is possible to reference an existing type schema.

The instance then defines a pointer to an object in the repository by specifying `referenceType` and `referenceKey`. Note, that for the object, the key includes `objectId`. This allows to reference an object that is already saved in the data repository of MUSE4Anything.

Resource references can only be updated from and to other resource references. Additionally, it is not possible to switch from a taxonomy reference to a type reference, and vice versa, as they are not compatible. Taxonomies have to be set when defining a resource reference, and they cannot be changed in updates afterwards. Types do not have to be defined, i. e., it is possible to accept any type instead of a specific type. Therefore, in this case, some changes are important to support. The type definition can change from no limitations of the object type to a specific object type, and the other way around. Going from the general to the specific case should be supported as some objects may still be valid after the update. However, some objects may require manual effort to adapt to the new type restrictions. The other way around, i. e., from a specific object type to any

Listing 4.5 Example resource reference definition and object in MUSE4Anything

```
// Example resource reference definition
{
    "type": ["object"],
    "customType": "resourceReference",
    "required": [
        "referenceType",
        "referenceKey"
    ],
    "properties": {
        "referenceType": {
            "const": "ont-object"
        },
        "referenceKey": {
            "additionalProperties": {
                "singleLine": true,
                "type": "string"
            },
            "type": "object"
        }
    },
    "referenceType": "ont-type",
    "referenceKey": {
        "namespaceId": "2",
        "typeId": "14"
    }
}
```

```
// Example resource reference object
{
    "referenceType": "ont-object",
    "referenceKey": {
        "namespaceId": "2",
        "objectId": "42"
    }
}
```

object type, does not impose new limitations and must be supported as well. In the case of changing from a specific object type reference to another specific object type, no objects can be migrated automatically. Therefore, this change is considered invalid and is prevented.

4.2 Schema Validation

MUSE4Anything already allows editing type definitions. To ensure that the automatic migration can be run successfully, a mechanism is implemented to validate changes before they are stored in the migration package. The method walks through the original type schema *source* and the edited schema *target* and matches related parts. Walking through the schema and acting according to the type is also modeled after the visitor pattern [PJ98]. Depending on the type defined in the source and target schemas, recursively checking nested schemas might be necessary, e. g., matching two lists with elements of different data types. While the primitive data types and enumerations typically do not need special treatment when matching schemas, data types like arrays, tuples, and objects often do.

Schema references, i. e., pointers to other type definitions, can be resolved with a method already provided in MUSE4Anything, which will be elaborated on later. After resolving, the validation can continue with the referenced definition. By using schema references, it is possible to construct very deeply nested schemas. For this reason, it is necessary to observe the depth of the schema definition, and stop the execution for extremely nested schemas.

From \ To	Bool	Int	Num	Str	Enum	Array	Tuple	Obj	Res Ref
Bool	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Limited*	*Limited*	*Limited*	*No*
Int	*Lossy*	*Yes*	*Yes*	*Yes*	*Yes*	*Limited*	*Limited*	*Limited*	*No*
Num	*Lossy*	*Lossy*	*Yes*	*Yes*	*Limited*	*Limited*	*Limited*	*Limited*	*No*
Str	*Lossy*	*Limited*	*Limited*	*Yes*	*Yes*	*Limited*	*Limited*	*Limited*	*No*
Enum	*Lossy*	*Limited*	*Limited*	*Yes*	*Yes*	*No*	*No*	*No*	*No*
Array	*Limited*	*Limited*	*Limited*	*Yes*	*No*	*Yes*	*Yes*	*No*	*No*
Tuple	*Limited*	*Limited*	*Limited*	*Yes*	*No*	*Yes*	*Yes*	*No*	*No*
Obj	*Limited*	*Limited*	*Limited*	*Yes*	*No*	*No*	*No*	*Yes*	*No*
Res Ref	*No*	*No*	*No*	*No*	*No*	*No*	*No*	*No*	*Yes*

Table 4.2: Validity of type conversions in automatic migration. *Yes* indicates a conversion is possible without problems, while *no* states it is not possible at all. *Lossy* conversions are generally supported, but might introduce a loss of information during the transformation, e. g., migrating 3.14 to the integer 3. A *limited* conversion is only valid for a sub-set of values, e. g., only simple instances of complex types can be migrated to primitive types.

Table 4.2 presents the type conversions allowed by the validation function. While some transformations are considered valid without limitations, others are limited to a small sub-set of cases, or bear the risk of data loss. Finally, some conversions are considered invalid due to the large effort necessary to implement them. As mentioned in Chapter 3, primitive data types, i. e., boolean, integer, number, and string should be easily transformable within each other. However, transforming integer to boolean is very likely to cause data loss. Furthermore, conversions between primitive types and very simple definitions of complex types should also be supported. This includes for example arrays with a length of one, or objects with just one property. In an array of length one, it can be attempted to migrate the single element to the primitive type. Similarly, a single object property can be migrated to the appropriate primitive type. The other way around is supported analogously, e. g., if a boolean schema is updated to be an object. In this case, only if the object schema defines exactly one property, it is attempted to migrate the former boolean to that property. It is not possible to change a list or tuple type to an object type, and vice versa, as that introduces a lot of complexity in matching properties to elements. If a type update is not considered valid for automatic migration, one option is to split the transformation into several parts. For instance, consider changing an object type with many properties to a primitive integer type. In the first step, unnecessary properties would be removed from the object schema, and in the next step it would be possible to change the definition to an integer. The migration engine can handle transformations from objects with just one property to a primitive data type. Therefore, it can execute the process and update objects, unless they violate some restriction. Two type versions are created and objects are migrated twice.

As an example for matching complex types, two arrays shall be matched. In Listing 4.6, the source and target schemas of an array definition is provided. The array elements of the source schema are defined as integers and for the target schema theye will be strings. As shown in the example, the

Listing 4.6 Example source and target array schema

```
// Source array schema
{
    "arrayType": "array",
    "items": {
        "type": ["integer"]
    },
    "type": ["array"]
}
```

```
// Target array schema
{
    "arrayType": "array",
    "items": {
        "type": ["string"]
    },
    "type": ["array"]
}
```

Listing 4.7 Contradictory update of integer schema

```
// Source integer schema
{
    "type": ["integer"],
    "maximum": 10
}
```

```
// Target integer schema
{
    "type": ["integer"],
    "minimum": 11
}
```

type of array elements is defined in `schema["items"]` which also defines the `type` keyword of JSON Schema. Consequently, the definition of items is also a JSON Schema, and therefore it can be used in a recursive call of the validation function, now as the new source or target schema. In this simple case of matching two arrays, it means checking whether the element types are compatible. For other cases of matching complex types with each other, the idea is similar. The structure of nested schemas resembles a tree-like structure that can be traversed. This approach allows following the nested schemas up to the point of reaching a simple conversion, which often involves a primitive data type where no more recursive call is necessary. Schemas that contain references to themselves require special attention. It is necessary to keep track of visited schemas to recognize those recursive schemas.

Finally, it is also important to look at potential contradictions in the type's specific keywords in the course of schema validation. Listing 4.7 gives a fitting example for an integer update. On the first look, this type update seems valid as it does not include an invalid data type change. However, a closer look at the keywords `maximum` and `minimum` reveals a contradiction that would prevent automatic migration. Since all existing objects can only store integers smaller than ten, none of them would meet the requirement from the target schema. Similarly, strings, arrays, and tuples can define restrictions to length and item counts. These values need to be compared to prevent updates that would require big manual effort afterwards, as no objects could be migrated automatically.

4.3 Integration

As pointed out in the last chapter, the validation method is embedded in the API call of MUSE4Anything as indicated in Figure 3.2. When a user edits a type and attempts to save it, a method in the API is called. Before the implementation of the migration engine, a new type version was created and stored to the database directly. Now, the changes must first be validated by the validation function. The modifications of the schema will only be saved once they have been validated. The validation was implemented as described above. After the update has been approved,

the regular process of creating and storing a new type version continues. When the new version has successfully been saved, the data migration process can begin and access the new type schema in the database.

With the implementation of data migration in MUSE4Anything covered, and the process of automatic migration explained, it is now time to look into some details of the implementation that were necessary to set up a reliable migration engine. There have been some references to libraries that were used, and special cases, such as resolving schema references, which will be presented next.

4.3.1 Celery Background Task

Celery [SUN+16], which is a distributed task queue, was used to implement the automatic migration in a background process. Celery defines workers, which are responsible for executing tasks which are stored by the broker. The broker is also responsible for the reliable communication between the workers. In the case of MUSE4Anything, the broker is a redis [1] docker container. With celery being set up, functions written in Python can be defined as celery tasks using the `CELERY.task` decorator, where `CELERY` is an instance of a celery application. In the process of realizing automatic data migration in MUSE4Anything, one background task was designed that handles migrating an object to a new version.

Instead of calling that task once with a list of all objects of a type, it is called for every object separately, which eases error management. Celery provides *groups* to call a task for each element of an iterable. Depending on the amount of workers, tasks specified in a group are executed in parallel, improving the performance of the migration engine. Using celery, it is possible to initiate the data migration as a number of background processes that do not block the system. The task that handles the migration of data retrieves the current version of an object from a given ID, and updates the data stored in it using the migration package. The update needs to be validated, meaning the new data will be checked for the new type schema, to ensure it does not validate any newly set limitations. Finally, a new object version with the updated data is stored in the database and the background process terminates.

4.3.2 Reference Resolving

In the above-mentioned task that calls the migration function of the new package, another adaptation is necessary. As elaborated before, type schemas can contain schema references to other type schemas. This is helpful for combining schemas, e. g., defining an array of persons when the person type already is defined. These references can be remote, which means they are pointing to a different type definition in the MUSE4Anything repository, or local, i. e., in the same type schema. To perform automatic data migration, both kinds of references need to be resolved. This means, the referenced schema must be retrieved, and then used as the respective source or target schema.

[1] https://redis.io/

Listing 4.8 Example definition of a local reference in MUSE4Anything

```
"definitions": {
    "root": {
        "type": ["object"],
        "properties": {
            // Local reference
            "age": {"$ref": "#/definitions/1"}
        }
    },
    // Referenced type schema
    "1": {
        "type": ["integer"],
        "maximum": 99
    }
}
```

MUSE4Anything provides functionality to resolve remote references. However, it requires a request context, which at first does not exist in the app context for background tasks. Flask [24b] offers a work-around by creating a `test_request_context`, that pushes a fake request context on the given app. The request context is required when resolving remote references, as this includes a url validity check. That context is not available to the migration function as it is an isolated package. For this reason, the work-around with the fake request context is used.

Local references also need extra steps to be resolved correctly. Consider the schema definitions shown in Listing 4.8. In `root`, where the main type schema is stored, an object with just one property `age` is defined. This property is a local reference to `definitions/1`, where an integer with a maximum value of 99 is declared. The recursive call uses the data stored in the `age` property of the object to continue with the migration. However, one cannot access the parent schema containing the other schema definitions from the nested property schema. To be able to resolve the local reference from a nested schema, it is necessary to introduce two more parameters to the method signature. The source and target *root schemas*, i. e., the complete original schemas, must be passed along in every step of the recursion. That way, even with nested schemas, the local references can be resolved successfully by accessing the root schema instead of the nested, partial schema.

With that, the implementation of the migration engine in MUSE4Anything is covered. The different steps necessary to realize automatic data migration have been explained. First, it is necessary to validate the schema update, to ensure data can be migrated automatically in the next step. The migration of data is dependent on its type, and therefore, it is necessary to dynamically compare source and target types and resolve potential references. Using celery, it is possible to execute this process in the background and finally store the changes to the database.

5 Evaluation

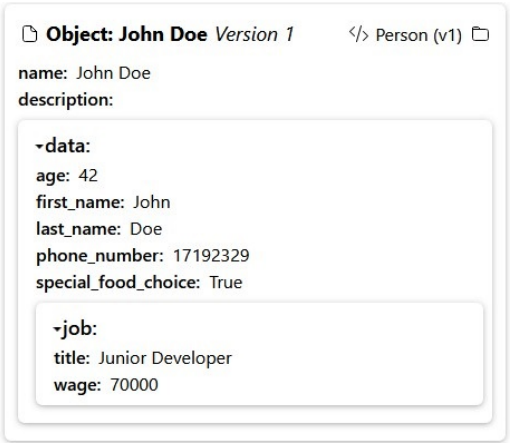
The subject of this chapter is evaluating the implementation of the data migration engine. Besides the automatic tests, that are used to verify the migration module's expected functionality, the defined use cases from Chapter 3 will be used to evaluate the migration process. This will showcase how the requirements given by the use cases are fulfilled.

5.1 Automatic Tests

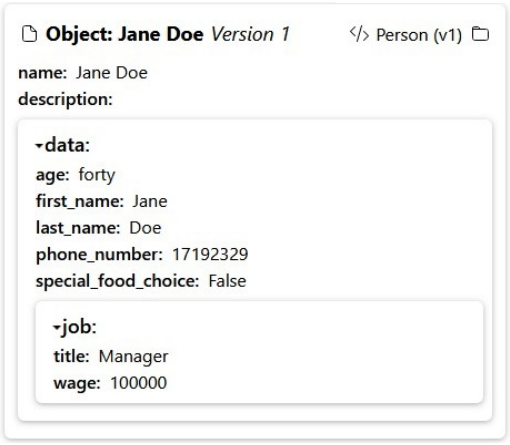
During development, automatic unit tests, developed with Python's testing framework *unittest*, were used to evaluate the functionality of the migration module independently of the other components. According to test-driven development [JS05], the test cases need to cover all migration operations that should be fulfilled and define the correct requirements. With the help of 139 test cases, the expected behavior in different scenarios was defined. Following the idea of TDD, the program was extended incrementally to implement the required behavior. The tests are organized by target type and include migration tests from various source schemas. Using these automatic tests, it is possible to detect breaking changes quickly during development and before deployment, which is specific to TDD. An example test function migrating an array to a string is shown in Listing 5.1. The automatic test cases were all designed in the same structure as the given example. The setup of a test case includes defining a source and a target schema, and data that conforms to the source schema. The two schemas define certain types, here, the transformation is from an array of integers to a string. The idea is to migrate the data according to the schema change. The migrated data is then evaluated against the expected value.

Listing 5.1 Example test function

```
def test_from_array_to_str(self):
    # Setup
    source_schema = {
        "arrayType": "array",
        "items": {"type": ["integer"]},
        "type": ["array"],
    }
    target_schema = {
        "type": ["string"]
    }
    data = [2, 9, 44]
    # Migration
    updated_data = migrate_data(data, source_schema, target_schema)
    # Evaluation
    self.assertEqual("[2, 9, 44]", updated_data)
```



(a) First version of *John Doe*



(b) First version of *Jane Doe*

Figure 5.1: First versions of *Person* objects

Testing the mechanism of reference resolving requires initializing the flask application within the test cases. As discussed in the previous chapter, resolving references requires both an application context and a request context. Consequently, these test cases can only be executed once these contexts are appropriately set up. To do so, unittest's `setUpClass` functionality was used. It allows setting up resources that are shared across all test methods in a class before the execution of the test. Therefore, it is useful to instantiate an application that can be used to test reference resolving.

5.2 Use Cases

Besides automatic tests, the migration was also evaluated with the use cases introduced Chapter 3. They are inspired by how users would interact with the system, and therefore provide a practical way to evaluate the implementation of a system. The desired changes are briefly mentioned, the modified schemas are presented and the automatic migration of objects is evaluated.

5.2.1 Use Case "Update Structure of Employee Data"

Figure 5.1 provides two example objects of the *Person* type schema that was introduced in Chapter 3. The important difference lies in the `age` property which is defined as a string. While the age of *John Doe* was saved as a string that can be interpreted as a number easily, the age of *Jane Doe* was not. In this use case, *John Doe* should be migrated successfully, whereas *Jane Doe* cannot be migrated due to the limitations of Python's conversion function.

The updated schema for *Person* is shown in Figure 5.2, according to the decisions presented in Chapter 3. The screenshot shows that `age` is now defined as an integer (string before), whereas `phone_number` is a string (integer before). Furthermore, `special_food_choice` was changed to an enumeration (boolean before). Due to limitations of the migration engine, it is necessary to include the original boolean values in the enumeration. When migrating to an enumeration, only values that are specifically defined, are accepted. This is done to avoid accidental data loss, e. g., by cutting

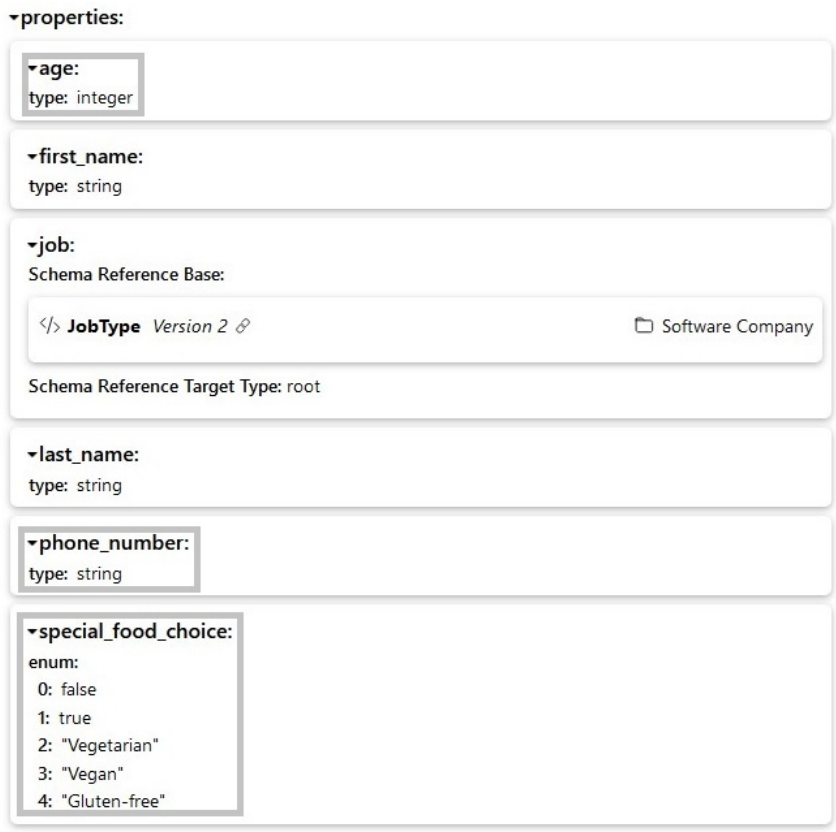


Figure 5.2: Updated version of *Person* schema as shown in MUSE4Anything. Changed properties are marked in gray.

off decimals of the original data entry. Even though it would be possible in this case, to map `True` to `Vegetarian`, defining a general rule in MUSE4Anything is difficult. The approach chosen here ensures objects can be migrated automatically.

After the update, the two objects are checked. As shown in Figure 5.3, *John Doe* was updated to *version 2* and the type version points to the correct and updated *Person (v2)* type. While the screenshots show that no data was lost, in this case it is not obvious how the data type changed. To highlight the changes, also the updated JSON data from the database is given in Listing 5.2. It shows that `age` is now an integer and `phone_number` is a string that can contain non-numeric characters. For the enumeration defined `special_food_choice`, `true` is an accepted value.

The *Jane Doe* object was not migrated, i. e., no new object version was created, as `forty` is not easily transformable to an integer. However, it is possible to view only outdated objects after a type update using a filter in the list of objects. This filter provides a clear overview on objects that need to be manually migrated.

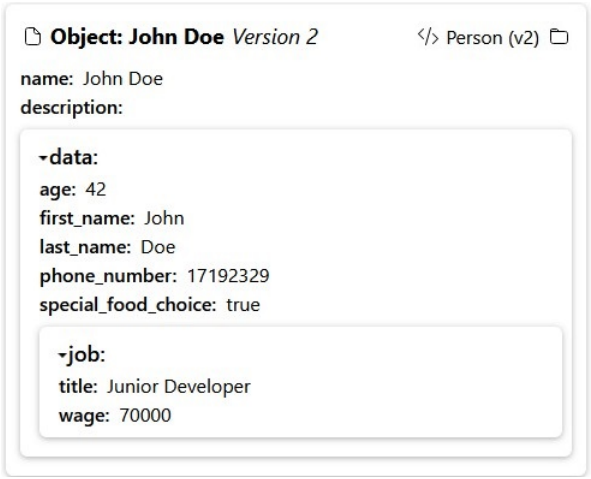


Figure 5.3: Updated version of *John Doe*

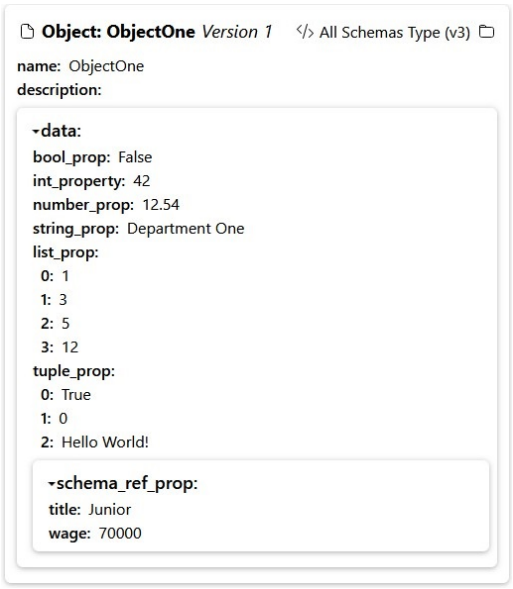
Listing 5.2 Updated JSON data for *John Doe*

```
{
    "age": 42,
    "first_name": "John",
    "last_name": "Doe",
    "phone_number": "17192329",
    "special_food_choice": true,
    "job": {
        "title": "Junior Developer",
        "wage": 70000
    }
}
```

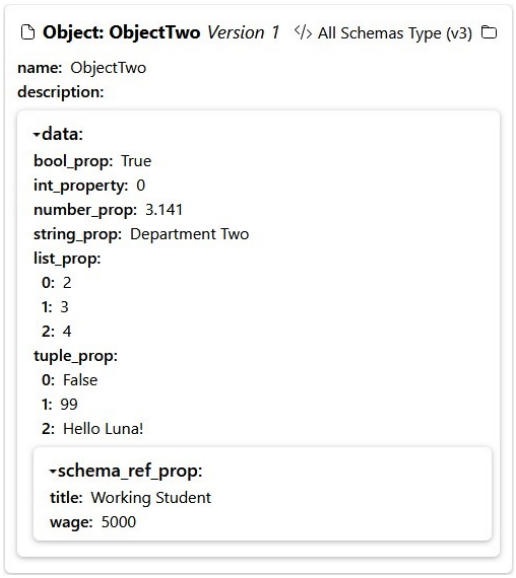
5.2.2 Use Case "Quality Control Test"

Two example objects conforming to the defined *All Schemas Type* schema are shown in Figure 5.4. The data is set to the presented values, which vary across the two objects. One of the numbers which will be transformed to a boolean is set to a value greater zero, the other one is zero to see the different outcomes of migrating to a boolean. In this use case, both objects are expected to be migrated to the new type version.

The updated schema for the *All Schemas Type* is shown in Figure 5.5, according to the decisions presented in Chapter 2. The changes were done in the UI of MUSE4Anything which offers editing forms for types. The `bool_prop` is now defined as a string, while the `int_property` is still an integer. The referenced schema for `list_prop` was also updated to a new version that now accepts strings as elements and no longer interprets them as integers. The `number_prop` was renamed to `changed_prop`, i. e., the original property was deleted and the new one was added in the same step. Under the new property name, the type has been changed from number to boolean. The reference to *JobType* has not been modified. An enumeration of strings was introduced for the `string_prop`. The valid strings for that enumeration are shown, and it is easy to update these permitted values in future changes. Finally, the tuple property was also updated to a new version of the tuple schema. Since the tuple is



(a) First version of *ObjectOne*



(b) First version of *ObjectTwo*

Figure 5.4: First versions of *All Schemas Type* objects

Listing 5.3 Updated JSON data for *All Schemas Type* objects

```
// Updated ObjectOne
{
    "bool_prop": "False",
    "changed_prop": true,
    "int_property": 42,
    "string_prop": "Department One",
    "list_prop": ["1", "3", "5", "12"],
    "tuple_prop": [true, "0", "Hello World!"],
    "schema_ref_prop": {
        "title": "Junior",
        "wage": 70000
    }
},
```

```
// Updated ObjectTwo
{
    "bool_prop": "True",
    "changed_prop": true,
    "int_property": 0,
    "string_prop": "Department Two",
    "list_prop": ["2", "3", "4",],
    "tuple_prop": [false, "99", "Hello Luna!"],
    "schema_ref_prop": {
        "title": "Working Student",
        "wage": 5000
    }
}
```

defined by a remote reference to *TupleType*, it could be that it has been updated independently. These changes are now integrated into the *All Schemas Type*. In the run of a set of updates, the second element of a tuple must now be interpreted as a string. The other elements remain untouched.

After the update, the two objects are checked. Listing 5.3 shows the JSON data for both objects after they were migrated. The migration was mostly lossless, except for the mapping of `42` to `true`. This is due to the smaller value space of booleans compared to integers. The JSON data shows that `bool_prop` is now a string, `changed_prop` is a boolean, and `string_prop` is a value of the enumeration defined in the schema. Furthermore, the list now has string elements. While the referenced *JobType* has not changed, the *TupleType* definition has. The second element is now a string and accepts more varied input than the integer from the old tuple schema.

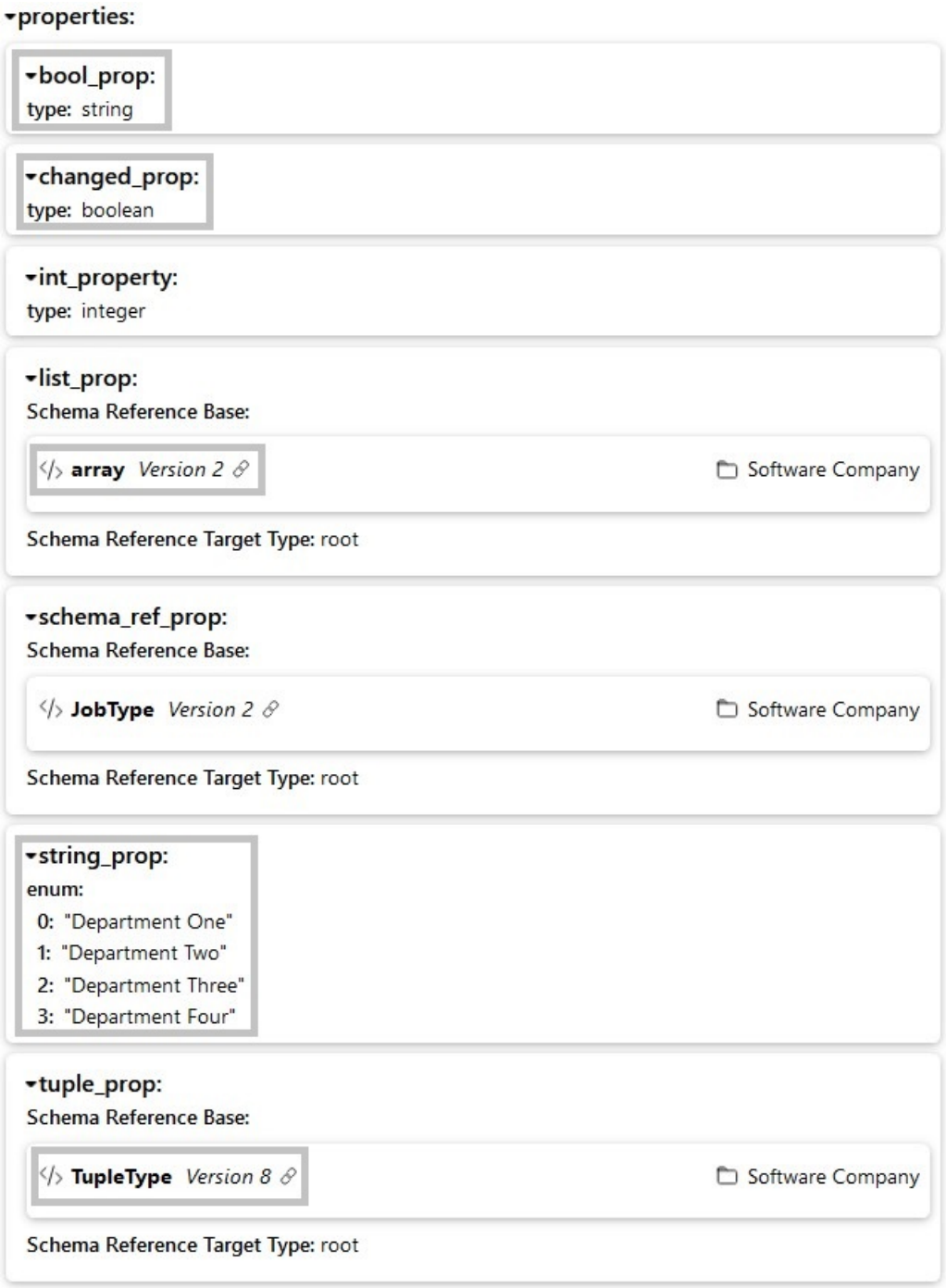


Figure 5.5: Updated properties of *All Schemas Type* schema as shown in MUSE4Anything. Changed properties are marked in gray.

5.2.3 Use Case “Invalid Schema Change”

Figure 5.6 shows the notification in MUSE4Anything for an invalid type update. In this case, the attempt was to change an array definition to an object definition. As stated in R 3.2, migrating arrays to objects is not desired. Since the type version is not saved to the database, there is also no migration of objects. Hence, the expected behavior is achieved and the invalid schema update prevented.

5.3 Result of the Evaluation

Based on the results of the use cases and the automatic tests, the requirements from Chapter 3 can be evaluated. R 1.1, R 1.2, and R 1.3, which are related to the migration of primitive data types (boolean, integer, number, string) are fulfilled. The conversion of primitive types is tested with automatic tests, and data loss is limited to cases of changing to a type with a smaller value space than the source. The use cases also tested the migration from primitive types to an enum definition. R 2.1, R 2.2, and R 3.1, i. e., the migration from an enum to a different data type, and simple instances of complex types, were not covered by the use cases but are part of the automatic tests. However, the migration of more complex type definitions as required by R 3.2, R 3.3 and R 3.4, are all covered by the use cases as well. There were migrations between array and tuple schemas, and object definitions were updated as well. The migration of the elements of those compound types is often based on the migration of primitive types, which is achieved by using the nested schemas in the migration function call.

Furthermore, schema references were used in both the use cases and automatic tests to ensure that the migration works as expected by R 4.1. After resolving those references, the migration function can access the schema and perform the appropriate transformations. The migration of resource references is implemented as stated in R 4.2 which was tested manually.

While the first two use cases provided valid schema updates, the third use case gave an invalid one. MUSE4Anything accepted the valid updates and prevented the invalid update from being stored to the database. Therefore, R 5.1 and R 5.2 are satisfied. Finally, it was shown that the migration engine is able to migrate all objects after a type update, unless specific objects have values that are not convertible to the target type, e. g., migrating `forty` to an integer. New object versions were created when the data stored in objects could be migrated successfully, i. e., also R 6.1 and R 6.2 are met.

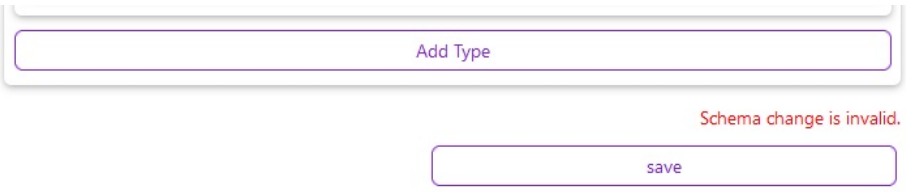


Figure 5.6: Notification of invalid schema change

The evaluation with automatic test cases and the execution of the use cases, shows that data can now be automatically migrated in MUSE4Anything. The automatic tests check the correct functionality of the migration module, and can be used to define expected behavior. If changes in the requirements happen, these tests should be adapted accordingly to reflect those changes. Afterwards, it is possible to change the implementation of the migration engine if necessary.

By testing the data migration with the help of the use cases, it was possible to examine the behavior of the migration engine in the running system. The test runs have shown successful migrations of a small selection of supported combinations, but also included migration failures and an invalid schema change. Covering all of them, would be infeasible for manual tests and would not provide more context. It was considered more useful to use automatic tests to cover all combinations and present a subset of features in more detail here.

The migration engine does not yet support every transformation perfectly, e. g., mapping boolean values to specific string elements of an enumeration. However, these limitations apply only to a sub-set of changes, and often can only be reliably solved by including the user into the migration process. Here, the focus was on implementing an automatic migration engine that can handle a wide range of changes. However, improving the nature of these mappings with user input could be interesting for future enhancements. Default values would be an option to manage this, however there can only be one default value regardless of the different original values present in the property. Therefore, false and true would both be mapped to the same default value. Implementing useful mappings of values in an old schema to values in a new schema requires knowledge of the specific schemas, but could increase the quality of the migration process. In conclusion, the presented migration engine for MUSE4Anything meets the requirements and can solve the use cases, and is still extensible through code changes for potential improvements in the future. Test-driven development helps to provide extensibility, as it is always possible to adapt test cases to represent new behavior, which helps in implementing the necessary changes.

6 Related Work

There exists a lot of research on algorithms for automatic data migration after schema changes. Many approaches are focused on migrating data in databases where the schema is defined by database specialists and not, like in MUSE4Anything, user-defined. This chapter provides an overview of other work on schema matching and data migration, and presents similarities and differences. Data migration has been a research subject for a long time [BLN86]. In 2001, an extensive research work on automatic schema matching [RB01] attempts to find a classification for available algorithms. Using this classification, the approach of MUSE4Anything would be classified as a mix of *instance-based* and *schema-based*. It also has elements of *language-based* matching, when mapping properties of the object type in MUSE4Anything, as here names of properties are compared. The survey was picked up by the authors again ten years later for an updated version and an overview on new tools in the research area of schema matching [BMR11].

Madhavan et al. [MBR01] developed *Cupid* to tackle the problem of schema matching. They aim to achieve more reliable schema matching by combining various matching techniques. To compute a similarity measure between schemas, the tool employs linguistic approaches and considers both the elements of a schema and the schema as a whole, including references and constraints, as well as other techniques. Those similarity values are then used to calculate the likeliness of matches and compute a schema match. Cupid interprets schemas as graphs such that the algorithm can represent references in its model. With this more elaborate approach, it is possible to find complex matches between schemas. Finding more complex matches as it is done in Cupid, could be useful in MUSE4Anything to match properties of an object type definition not only on a lexical level.

ARTEMIS [CD99] is another schema integration approach based on the calculation of *affinities*. Affinity describes how much two elements in different schemas are related semantically. To calculate the affinity of two attributes, ARTEMIS also considers name and structure of the elements in source and target schema. ARTEMIS is used as part of a database mediation system called Mediator environment for Multiple Information Sources (MOMIS) [BBC+00; BCV99; RB01], that is used to create a global schema from a set of independent base schemas. Similarly to Cupid, ARTEMIS also focuses on finding matches between schema elements which is an essential part of data migration.

In contrast to Cupid and ARTEMIS, the migration algorithm of MUSE4Anything is simpler, but fulfills the requirements of automatic data migration. Further enhancements, such as a more elaborate matching of object properties, could provide useful in the future. However, the deterministic nature of the algorithm ensures predictability of the migration, and avoids false mappings. Therefore, this implementation was preferred over a complex matching algorithm, that might introduce unpredictable behavior.

Another schema match approach is *SEMantic INTegrator (SEMINT)* [LC00]. It takes advantage of neural networks to improve the search for matches between schema versions. In the end, attributes of different schemas are mapped to one another depending on constraint-based and content-based evaluations that the neural network uses. This approach can introduce performance issues with

large schemas due to the reliance on neural networks [CHR98]. It is a flexible approach, but it does not support name-based matching and is not deterministic. Similarly, *Learning Source Description (LSD)* [DDL00] uses machine-learning techniques in the process of matching new data sources to an existing, global schema. In that regard, SEMINT and LSD differ from MUSE4Anything and other presented approaches, as they do not investigate schema changes due to updates or similar processes, but due to integration. LSD builds on the two phases learning and classification. In the first phase, *learners* get to know the global schema and in the classification phase, the acquired knowledge is used to classify new schemas appropriately. Both approaches are not applicable in the case of MUSE4Anything, because they are not deterministic, and SEMINT does not support name-based matching.

The Stanford University also developed a schema matching tool called *Semantic Knowledge Articulation Tool (SKAT)* [MWJ99] which is also used in the ontology matching tool *ONION* [MWK00]. SKAT is built on four steps. Initially, the user provides some indications for the matching. With these suggestions, the tool then outputs an initial set of matches. These matches then need to be evaluated by the expert user with the three options *approved*, *rejected*, and *irrelevant*. Then, SKAT generates an improved matching using these answers. This approach could also be helpful for future enhancements of MUSE4Anything. The user could be handed information about the consequences of the changes which first need to be approved in order to avoid accidental data loss. SKAT also uses various techniques to find matches, such as structure-based and term-based rules.

COMA 3.0 [MRA+11] is another schema and ontology matching tool developed by the University of Leipzig. It is the newest version of COMA [ADMR05; DR02], and was released as a community edition. It offers schema matching with a graphical user interface and can be used for database schemas or XML message formats. As the name *Combining Match Algorithms* suggests, COMA combines various matching techniques to achieve a better matching result. Among other, it uses *context-dependent*, *fragment-based*, and *reuse-oriented* matching. The first is used to handle relations and correspondences of elements that are shared. Besides that, COMA employs a divide-and-conquer strategy, breaking down large problem statements into smaller, easier to solve parts. Finally, by using the result of earlier matches, it is possible to compose a match from previous matches, and improve the quality of the migration result. COMA is useful for schema matching, however the problem of migrating objects in MUSE4Anything would remain. As that would only solve a small part of the migration problem in MUSE4Anything, the need for a new approach persists.

QuickMig is an approach for semi-automatic schema matching for data migration proposed by Drumm et al. [DSDR07]. It was developed in the context of migrating legacy data to new systems. Automating this process gains importance with the ever-increasing amount of data available today as humans usually are slower than algorithms and make more mistakes. The developers of QuickMig do not only focus on the schema, but also include several other factors in the migration process, by including example instances from both, legacy and target system, and questionnaires. These questionnaires about the relevant parts of a schema need to be answered by the user before initiating the migration proces to incorporate the user's expert knowledge about the domain. UI was not the focus here, thus everything with extensive user interaction would not be feasible for this work.

The authors describe a five-step process to perform semi-automatic data migration which is depicted in Figure 6.1. The first step is to collect information about the source system from the questionnaires, which need to be filled out by someone who has knowledge about the given source system. This information reduces the complexity of realizing the migration process significantly. Next, a user

creates instances from the target system manually in the source system. These instances can then be used by matching algorithms that are based on data instances to find correct matches, without a focus on the schema. QuickMig incorporates various matching algorithms to achieve a more reliable matching. The import step describes loading the source schemas and the manually created sample instances into the QuickMig system, such that it can execute the matching algorithms in the next step. Finally, it is necessary to review the created mapping, and to initiate the migration process if the generated mapping is accepted [DSDR07].

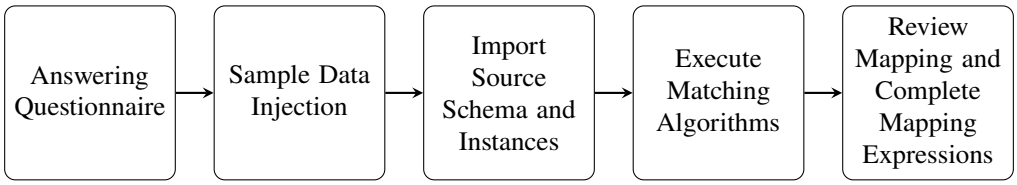


Figure 6.1: Five steps of the QuickMig migration process [DSDR07]

Similarly to QuickMig, the data migration engine in MUSE4Anything provides a more automatic approach to data migration than handling updates manually. However, there are no questionnaires or example instances used in MUSE4Anything. Even though they could provide some improvements in the final migration result, questionnaires were not included as they would increase the manual effort. In contrast to QuickMig, the migration engine in MUSE4Anything is a fully automatic data migration engine that only requires manual effort when data could not be matched to the new schema automatically. If possible, these problems are indicated before running the migration, to avoid a huge amount of effort necessary in manually making sure the data conforms to the new set of rules.

Ferrandina et al. [FMZ94] highlight that the advantage of a lazy implementation of updates is not blocking the system for a long-running update process. Instead, a lazy implementation defers the transformation of data until the required resources for the migration are available, or it is required to see updated data. Ferrandina et al. implement lazy updates as follows: First, there is a check whether the schema update is valid. If it is, the system can be used normally. While an object is required to immediately *logically* follow the new schema, the *physical* object is transformed only once accessed. The challenges of this approach are that it is necessary to remember those schema changes and to check the version every time an object is accessed. In MUSE4Anything, a background task to update objects is started immediately after the initial check of the schema update. With that, the running system is not locked for the entire update process, similarly to what Ferrandina et al. [FMZ94] suggest. However, objects are not updated automatically when accessed, i. e., the user might see outdated data. For that reason, all objects of a changed type need to be migrated immediately after the update. A problem that might occur in long-running update processes due to many objects is, that an object may still be outdated at the point of access.

Zicari [Zic91], who was co-author of the above-mentioned paper, also offers a look at the basic operations during type updates. He sorts the required tasks into *simple tasks*, e. g., adding or deleting attributes of a type, and *complex tasks*, e. g., changing the name or the type of an attribute. Complex tasks are typically expressed by a combination of simple tasks. This was also used to implement the migration algorithm in MUSE4Anything, e. g., when migrating an object schema with property name change. The original value is stored in the new property, before deleting the old property.

Studying related work, tools, and different approaches has been helpful in designing the migration engine for MUSE4Anything. The presented ideas and approaches were considered in the process of implementing this new tool. Based on the findings in related work, it is possible to improve the implementation of data migration in MUSE4Anything in the future. Extending the matching of object properties, and allowing user input for better migration results, are two possible approaches to improve the user experience and to handle more varied changes to the schema. With that, the migration engine could manage more changes automatically, and the quality of the migrated data would also increase.

7 Summary and Outlook

The beginning of this thesis establishes the foundations of implementing assisted data migrations, and explains the necessity of integrating this functionality into MUSE4Anything in Chapter 2. Afterwards, the design of the implementation and requirements of automatic data migration in MUSE4Anything were presented in Chapter 3. Three representative use cases were introduced to validate requirements and to emphasize the importance of automatic data migration. There were both general requirements that should always be satisfied, and specific requirements that are only relevant to specific use cases. The first use case was defined to represent a likely change scenario, while the second was designed to include many potential changes at once. The last use case coveres an invalid schema change, to highlight the error management. This allows covering many application scenarios of automatic data migration. A short overview of existing tools showed that they are not usable in the MUSE4Anything repository, and a new implementation is necessary.

Chapter 4 presents the implementation of the migration engine detail. It is realized as an extension of the type editing process, and split into two parts. First, a schema change needs to be validated, to ensure only those transformations stated in the requirements are possible. After validating the schema update, a new type version is saved to the database, and the migration of objects is initiated as an asynchronous background task. By applying schema changes to data stored in objects, the migration process ensures that new object versions conform to the updated schema and are saved to the database. The migration engine applies changes recursively, ensuring nested structures, such as lists of lists, are processed correctly. A separate process is started for each object, allowing concurrent object updates. The migration functionality is implemented in the backend of MUSE4Anything where it is accessed by the API when type changes are saved. The implementation details, including the handling of references, type-specific migration logic, and recursion for nested schemas, are discussed in depth. Using test-driven development ensured the migration engine meets all defined requirements.

In Chapter 5, the system was evaluated using both automated test cases and the predefined use cases, demonstrating successful and unsuccessful migrations, as well as an invalid schema change. The engine enables MUSE4Anything to perform the automatic migration of objects impacted by schema changes, according to the requirements. Objects are migrated to the new type version, unless the data cannot be transformed to conform to the new schema. Objects that cannot be migrated, are displayed to the user using a filter in the list of objects. The evaluation of the migration engine also identified areas for improvement, as some schema changes require additional effort, such as including original values in an enumeration.

Comparing this approach with related work in Chapter 6, emphasized the unique challenges of MUSE4Anything's schema management and the need for a custom solution to meet the defined requirements. The findings also identified potential extensions, such as enhanced automatic mapping of object properties, support for default values, and user-configurable migration options presented in the user interface.

An enhanced mapping of object properties could enable more elaborate schema changes to be validated for automatic migration. In that regard, it could also be helpful to consider questionnaires and dummy data as it is done in QuickMig [DSDR07]. The user could provide information about the expected mapping and give example migration objects that could then be used to migrate the other outdated objects automatically. With these additions, the result of data migration could be further improved, and the manual effort reduced.

Introducing default values could prove useful when defining new properties. When properties cannot be migrated to the new schema, e. g., after introducing a new maximum value, an option to cut off integer values at that new maximum, would allow migrating more objects automatically. However, these features require a clear representation in the UI, and further studies on which features offer a meaningful benefit to the migration compared to the added complexity.

User-configurable options in the migration process could include the choice between alternative conversion strategies, e. g., using array length instead of array elements when migrating to integers. Currently, the transformation between primitive types is solely based on Python's built-in functionalities. However, it could be useful to allow different approaches, e. g., in the case of matching boolean to string, or also matching an array to integer. In the first example, different keywords could be included that also evaluate to false. For the second example, several ideas exist: Migrating only arrays of length one as it is now, always choosing the first element of an array, or using the length of the array. All these approaches are valid. Allowing the user to choose from a drop-down menu which approach they want, would provide greater flexibility and control to the user, and enhance the migration engine further.

Furthermore, it is important to improve the error management. That includes more detailed feedback when a schema match is not valid, as well as information about failed object migration. Here, the focus was on implementing the migration engine, and not on the UI development. For unsuccessful migrations, it would also improve the user experience to introduce partially updated objects, e. g., in the case of changing an array of strings to an array of numbers. The partially updated object would store the migrated values of all array elements that could be migrated, and the original values of those elements that could not be migrated due to the limitations of the conversion function. However, this would require a new database table that is able to deal with these complications during data migration and the combination of different type schemas. Implementing these additions could further optimize the assisted data migration process in MUSE4Anything, minimizing the need for manual intervention.

Bibliography

[16] *JSON Schema Diff Validator*. Atlassian Pty Ltd, 2016. URL: https://bitbucket.org/atlassian/json-schema-diff/src/master/ (cit. on p. 30).

[20] *Aurelia*. Blue Spire Inc., 2020. URL: https://aurelia.io/ (cit. on p. 20).

[24a] *Atlas*. Ariga, 2024. URL: https://atlasgo.io/ (cit. on p. 29).

[24b] *Flask*. Pallets, 2024. URL: https://flask.palletsprojects.com/en/stable/ (cit. on pp. 20, 29, 46).

[24c] *SQLAlchemy*. SQLAlchemy, 2024. URL: https://www.sqlalchemy.org/ (cit. on pp. 20, 29).

[24d] *SQLAlchemy Alembic*. SQLAlchemy, 2024. URL: https://alembic.sqlalchemy.org/en/latest/ (cit. on p. 29).

[ADMR05] D. Aumueller, H.-H. Do, S. Massmann, E. Rahm. "Schema and ontology matching with COMA++". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: Association for Computing Machinery, 2005, pp. 906–908. DOI: 10.1145/1066157.1066283 (cit. on p. 56).

[Bar18] J. Barzen. "Wenn Kostüme sprechen – Musterforschung in den Digital Humanities am Beispiel vestimentärer Kommunikation im Film". PhD thesis. University of Cologne, 2018. URL: http://kups.ub.uni-koeln.de/id/eprint/9134 (cit. on pp. 17, 19).

[BBC+00] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, M. Vincini. "Information Integration: the MOMIS Project Demonstration". In: *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00* (Oct. 2000) (cit. on p. 55).

[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* 32 (Nov. 2016), pp. 323–328. DOI: 10.1007/s00450-016-0336-1 (cit. on pp. 17, 19).

[BCV99] S. Bergamaschi, S. Castano, M. Vincini. "Semantic integration of semistructured and structured data sources". In: *ACM SIGMOD Record* 28.1 (Mar. 1999), pp. 54–59. ISSN: 0163-5808. DOI: 10.1145/309844.309897 (cit. on p. 55).

[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 (cit. on pp. 17, 19).

[BLN86] C. Batini, M. Lenzerini, S. B. Navathe. "A comparative analysis of methodologies for database schema integration". In: *ACM Comput. Surv.* 18.4 (Dec. 1986), pp. 323–364. ISSN: 0360-0300. DOI: 10.1145/27633.27634 (cit. on pp. 23, 55).

[BLSW12] J. Barzen, F. Leymann, D. Schumm, M. Wieland. “Ein Ansatz zur Unterstützung des Kostümmanagements im Film auf Basis einer Mustersprache”. In: *Proceedings Modellierung 2012*. GI, Mar. 2012, pp. 1–16 (cit. on p. 19).

[BMR11] P. Bernstein, J. Madhavan, E. Rahm. “Generic Schema Matching, Ten Years Later”. In: *PVLDB* 4 (Aug. 2011), pp. 695–701. DOI: 10.14778/3402707.3402710 (cit. on pp. 17, 29, 55).

[Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259, Internet Engineering Task Force. 2017. URL: https://www.rfc-editor.org/rfc/rfc8259 (cit. on p. 21).

[Büh17] F. Bühler. *Design und Implementierung eines Storage-Backends für MUSE4Music*. 2017. DOI: 10.18419/opus-9566. URL: http://elib.uni-stuttgart.de/handle/11682/9583 (cit. on pp. 17, 19).

[Büh21] F. Bühler. “MUSE4Anything”. MA thesis. University of Stuttgart, 2021. DOI: 10.18419/opus-11410. URL: http://elib.uni-stuttgart.de/handle/11682/11427 (cit. on pp. 17, 19).

[CD99] S. Castano, V. De Antonellis. “A schema analysis and reconciliation tool environment for heterogeneous databases”. In: *Proceedings. IDEAS’99. International Database Engineering and Applications Symposium (Cat. No.PR00265)*. 1999, pp. 53–62. DOI: 10.1109/IDEAS.1999.787251 (cit. on p. 55).

[CHR98] C. Clifton, E. Housman, A. Rosenthal. “Experience with a Combined Approach to Attribute-Matching Across Heterogeneous Databases”. In: *Data Mining and Reverse Engineering: Searching for Semantics*. Ed. by S. Spaccapietra, F. Maryanski. Boston, MA: Springer US, 1998, pp. 428–451. DOI: 10.1007/978-0-387-35300-5_18 (cit. on p. 56).

[DDL00] A. Doan, P. M. Domingos, A. Y. Levy. “Learning Source Description for Data Integration”. In: *Proceedings of the Third International Workshop on the Web and Databases*. Ed. by D. Suciu, G. Vossen. Jan. 2000, pp. 81–86 (cit. on p. 56).

[DR02] H.-H. Do, E. Rahm. “COMA: a system for flexible combination of schema matching approaches”. In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB ’02. Hong Kong, China: VLDB Endowment, 2002, pp. 610–621 (cit. on p. 56).

[DSDR07] C. Drumm, M. Schmitt, H. Do, E. Rahm. “QuickMig - Automatic schema matching for data migration projects”. In: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. Nov. 2007, pp. 107–116. DOI: 10.1145/1321440.1321458 (cit. on pp. 56, 57, 60).

[EBH17] L. Eusterbrock, J. Barzen, F. Hentschel. *Eine Ontologie symphonischer Musik des 19. Jahrhunderts*. Tech. rep. Universität Stuttgart – Fakultät Informatik, Elektrotechnik und Informationstechnik, June 2017. URL: https://www2.informatik.uni-stuttgart.de/bibliothek/ftp/ncstrl.ustuttgart_fi/TR-2017-02/TR-2017-02.pdf (cit. on pp. 17, 19).

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

[FMZ94] F. Ferrandina, T. Meyer, R. V. Zicari. "Implementing Lazy Database Updates for an Object Database System". In: *Very Large Data Bases Conference*. 1994. URL: https://api.semanticscholar.org/CorpusID:8710381 (cit. on p. 57).

[Fur14] F. Furrer. "Eine kurze Geschichte der Ontologie". In: *Informatik-Spektrum* 37 (Aug. 2014), pp. 308–317. DOI: 10.1007/s00287-012-0642-3 (cit. on p. 19).

[Gri24] M. Grinberg. *Flask-Migrate*. 2024. URL: https://github.com/miguelgrinberg/flask-migrate (cit. on p. 29).

[Han17] M. Hansen. *JSON Schema Compare*. 2017. URL: https://github.com/mokkabonna/json-schema-compare (cit. on p. 30).

[HSHP21] A. Habib, A. Shinnar, M. Hirzel, M. Pradel. "Finding data compatibility bugs with JSON subschema checking". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 620–632. ISBN: 9781450384599. DOI: 10.1145/3460319.3464796 (cit. on pp. 17, 30).

[JS05] D. Janzen, H. Saiedian. "Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9 (2005), pp. 43–50. DOI: 10.1109/MC.2005.314 (cit. on pp. 35, 47).

[KBW+21] N. Krebs, F. Bühler, F. Weik, M. Salm, V. Pisano. *QHANA*. 2021. URL: https://github.com/UST-QuAntiL/qhana-backend (cit. on p. 20).

[LC00] W.-S. Li, C. Clifton. "SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks". In: *Data & Knowledge Engineering* 33.1 (2000), pp. 49–84. ISSN: 0169-023X. DOI: 10.1016/S0169-023X(99)00044-0 (cit. on p. 55).

[LXLZ13] J. Li, Y. Xiong, X. Liu, L. Zhang. "How Does Web Service API Evolution Affect Clients?" In: *Proceedings of the 2013 IEEE 20th International Conference on Web Services*. ICWS '13. IEEE Computer Society, 2013, pp. 300–307. ISBN: 9780769550251. DOI: 10.1109/ICWS.2013.48 (cit. on p. 23).

[MBR01] J. Madhavan, P. Bernstein, E. Rahm. "Generic Schema Matching with Cupid". In: *Proc 27th VLDB Conference* (July 2001) (cit. on p. 55).

[MRA+11] S. Massmann, S. Raunich, D. Aumüller, P. Arnold, E. Rahm. "Evolution of the COMA Match System". In: *Proceedings of the 6th International Conference on Ontology Matching - Volume 814*. OM'11. Bonn, Germany: CEUR-WS.org, 2011, pp. 49–60 (cit. on p. 56).

[MWJ99] P. Mitra, G. Wiederhold, J. Jannink. "Semi-automatic Integration of Knowledge Sources". In: *2nd International Conference on Information Fusion (FUSION 1999)*. July 1999 (cit. on p. 56).

[MWK00] P. Mitra, G. Wiederhold, M. Kersten. "A Graph-Oriented Model for Articulation of Ontology Interdependencies". In: *Proceedings of the 7th International Conference on Extending Database Technology (EDBT 2000)*. Vol. 1777. Lecture Notes in Computer Science. Springer, Apr. 2000. DOI: 10.1007/3-540-46439-5_6 (cit. on p. 56).

[PJ98] J. Palsberg, C. Jay. “The essence of the Visitor pattern”. In: *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac ’98) (Cat. No.98CB 36241)*. 1998, pp. 9–15. DOI: 10.1109/CMPSAC.1998.716629 (cit. on pp. 36, 42).

[RB01] E. Rahm, P. Bernstein. “A Survey of Approaches to Automatic Schema Matching.” In: *VLDB J.* 10 (Dec. 2001), pp. 334–350. DOI: 10.1007/s007780100057 (cit. on pp. 17, 23, 29, 55).

[Rod95] J. F. Roddick. “A survey of schema versioning issues for database systems”. In: *Information and Software Technology* 37.7 (1995), pp. 383–393. ISSN: 0950-5849. DOI: https://doi.org/10.1016/0950-5849(95)91494-K (cit. on p. 22).

[SBLE12] D. Schumm, J. Barzen, F. Leymann, L. Ellrich. “A Pattern Language for Costumes in Films”. In: *Proceedings of the 17th European Conference on Pattern Languages of Programs (EuroPLoP 2012)*. Ed. by C. Kohls, A. Fiesser. New York, NY, USA: ACM, July 2012, pp. 1–25. ISBN: 978-1-4503-2943-9. DOI: 10.1145/2602928.2603083 (cit. on p. 19).

[Sny86] A. Snyder. “Encapsulation and inheritance in object-oriented programming languages”. In: *SIGPLAN Not.* 21.11 (June 1986), pp. 38–45. ISSN: 0362-1340. DOI: 10.1145/960112.28702 (cit. on p. 35).

[SUN+16] A. Solem, A. S. Uddin, T. Nosrati, M. Bressolle-Chataigner, I. C. Maries, P. Maslanka. *Celery*. 2016. URL: https://github.com/celery/celery (cit. on p. 45).

[WA18] A. Wright, H. Andrews. *JSON Schema Draft 07*. 2018. URL: https://json-schema.org/draft-07/json-schema-release-notes (cit. on pp. 21, 22).

[Zic91] R. Zicari. “A framework for schema updates in an object-oriented database system”. In: *Proceedings. Seventh International Conference on Data Engineering*. 1991, pp. 2–13. DOI: 10.1109/ICDE.1991.131447 (cit. on pp. 41, 57).

All links were last followed on January 15, 2025.

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