FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Data Management Plans Migration for the DS Wizard Tool
Student:	Bc. Josef Doležal
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

- Acquaint yourself with the Data Stewardship Wizard (DSW) project with the focus to current solution of Knowledge Model (KM) migrations.
- Acquaint yourself with the Haskell programming language.
- Design and implement a solution for migrating data management plans created in DSW under specific KMs.
- Test the solution and demonstrate it on a non-trivial case study.
- Document your work.

This topic is offered by the #CCMi research group in collaboration with the GO-FAIR initiative.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 25, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Data Management Plans Migration for the DS Wizard Tool

Bc. Josef Doležal

Department of Software Engineering
Supervisor: Ing. Robert Pergl, Ph.D.

May 6, 2019

Acknowledgements

In this section, I would like to thank everyone who helped me throughout writing this thesis.

Firstly my supervisor, Ing. Robert Pergl, Ph.D. for his willingness, valuable advice and introduction to the data stewardship domain.

Thank everyone from the DataStewardship Wizard team, namely Ing. Jan Slifka, Ing. Vojtěch Knaisl and Ing. Marek Suchánek, for their help with the implementation part of the thesis.

I would also like to thank my family and friends for their support and help.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 6, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Doležal, Josef. *Data Management Plans Migration for the DS Wizard Tool*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019. Also available from: ⟨`https://github.com/josefdolezal/fit-mi-dip`⟩.

Abstrakt

Tato práce se zabývá návrhem a implementací nástroje pro migraci plánů řízení dat. Na základě požadavků a výstupů analýzy byl navržen modul aplikace integrovatelný do existujícího systému Data Stewardship Wizard. Práce reflektuje standardní postup návrhu software a obsahuje tak analýzu, návrh, realizaci a testování. Výstupem implementační části jsou moduly pro serverovou a klientskou část aplikace, umožňující migrovat existující plány na novější verze Knowledge Modelu.

Klíčová slova data stewardship, FAIR, otevřená věda, haskell, elm, funkcionální programování

Abstract

This thesis deals with the analysis and implementation of the data management plans migration tool. The application module for the existing system Data Stewardship Wizard is designed, based on the requirements and analysis. The thesis reflects the standard software development process and includes analysis, design, realization, and testing. The output of the implementation part is modules for both server and client application, allowing to migrate existing plans to a newer version of a Knowledge Model.

Keywords data stewardship, FAIR, open-science, haskell, elm, functional programming

Contents

List of Figures

List of Tables

Introduction

From the beginning of the ages, science has served to expand the collective consciousness of our society. It was, however, a privilege of experts in the industry and their research results were not accessible to the general public. With the rise of technologies of the twenty-first century, science is undergoing significant changes.

Science is no longer meant to serve a small group of scientists. Instead, we all use it in different forms in our daily lives. Today, strategic decisions in the commercial business are almost exclusively based on data analysis. Public participation in the processing of researched data is therefore almost unavoidable.

Making science accessible is however necessary not only for the general public. It also serves the future researchers who would like to start their research on the existing data. The Open Science Movement is focusing on making science researches accessible and reproducible.

In 2019, near to 1.4 billion euro will be funded into science and research from taxpayers' money in the Czech Republic itself[1]. In addition to that, the European Union will invest more than 97 billion euro under the Horizon Europe project for research and innovation in the years 2021-2027[2].

It is primarily the responsibility for the public funds, that should make researches funded in such manner accessible (both by humans and machines), available free of charge, reusable and reproducible. The format of publications using scientific articles, used during the whole twentieth century, is therefore not sufficient anymore. Such format of results is often not machine-processable or open (charged using paywalls), data are not understandable, and research conclusions are not reproducible.

The goal of the Open Science Movement is to change such an approach to science. By Open Science, we can understand an umbrella term for a systematic change in how researchers work, collaborate, share ideas and makes their researches more accessible and reusable. The guiding principle for Open Science, therefore, is making research data and its related tools fulfill the FAIR

principles.

According to these principles, all research data should meet four basic requirements: Findable, Accessible, Interoperable, Reusable (FAIR). Data should be findable for both humans and machines, accessible using open and universal protocols, interoperable with the use of formal language and vocabulary, and reusable with clear and accessible license and with relevant attributes.

As FAIR only covers theoretical principles, there is no strict implementation. One of the existing implementations is Data Stewardship Wizard (DSW). This system is developed in cooperation with ELIXIR[1] (specifically by ELIXIR CZ and EXILIR NL nodes) organization.

Goals of the thesis

This thesis extends the thesis of Ing. Vojtěch Knaisl which dealt with the topic of developing a migration tool for Knowledge Model (KM) in DSW system.

The main goal is to create a migration tool for Data Management Plans (DMP) in DSW. This tool will help researchers migrate existing plans based on an older version of knowledge model (or its customization) to a new version which includes updated information and changes.

The first chapter sums up the current state of the system and introduces the reader to the system architecture.

After that, in the second chapter, I will define formal system requirements together with use cases and user scenarios.

In the following chapter, Design, I will describe the conceptual design of the proposed solution together wireframes of the application user interface.

Then, in chapter Realization, I describe the implementation part of the conceptual design and its integration into the existing system.

The last chapter, Testing, describes which technique I used to verify whether the tool is working correctly, as designed.

[1] ELIXIR is a multinational group of scientists associating scientific resources such as databases, software tools, and educational materials across Europe.

Chapter 1

State-of-the-art

This chapter deals with the analysis of the current state of the DSW application. In the first section, the reader is briefly introduced to the architecture of the application and is acquainted with parts which have been already done.

After the introduction to the system design, I summarize the need for the migration tool for data management plans. Further, I analyze modules of each part of the system in detail. The reader will be acquainted with current solution design and application deployment.

1.1 Data Stewardship Wizard

As has been already stated, the DSW application is being developed in cooperation of ELIXIR CZ and ELIXIR NL research groups. The development is currently split into multiple independent parts.

The whole system is available as an open-source[2]. That said, the source code, documentation, and the development process are publicly accessible and open for contribution.

The most significant part of the project is the server application called DSW-Server. This application implements all the functionality and business logic of the system, persists data and provides an open Application Programming Interface (API). Using standard terminology, we can refer to it as a backend.

Since the server application does not provide any User Interface (UI) and only exposes an API using REST standard), there is a complementary web application called DSW-Client (or, in DSW terminology, called just *client* for short) which does just that. Using standard terminology, such application is referred to as a frontend.

[2]Application repositories are available at GitHub under the organization page at https://github.com/ds-wizard.

1.1.1 DSW-Server

The backend part of the application is written using the Haskell programming language. The primary goal of this portal is to help researchers create data management plans for their experiments. To be able to create a management plan, at least one knowledge model must be created. At the time of writing, the DSW is shipped without any prebuild knowledge model by default. The maintaing team, however, provides the *core* knowledge model separately to help data stewards setup initial data faster[3].

Public knowledge models (such as *core*) are represented using JavaScript Object Notation (JSON) and stored in a file on a public server. Data stewards then use these models to create questionaries. The role of researchers is to fill in the questionaries with meta information about their research.

Filled-in questionaries are used by data stewards to create management plans. This flow mostly covers system features.

We, however, have silent assumptions in the flow described above. The portal also has to support the following features in addition to the mentioned business logic:

- User management (roles, authorization, and authentication),
- Knowledge model (and its customizations) administration including creation, editing, and versioning,
- Data persistence (database connection, file system integration),
- REST API.

All of the mentioned functions are split into independent Haskell modules. The REST API layer is built using Scotty framework[3].

1.1.2 DSW-Client

DSW-Client is a frontend application written in Elm programming language[4]. It provides a web-based user interface for the server application.

The implementation was done by Ing. Jan Slifka as a part of his master's thesis in 2018[4].

The application is built using Elm Architecture. The logic of such an application is split into three separated parts:

[3]Scotty is an open-source web framework written in Haskell, inspired by Ruby's Sinatra. It is a lightweight alternative to frameworks like Yesod or Spock. The source code and more information is available on the project homepage: https://github.com/scotty-web/scotty.

[4]Elm is a type-safe programming language used for web applications frontend. The source code written in Elm is transpiled into JavaScript and interpreted using a web browser. Elm website: https://elm-lang.org.

Model – represents application state,

Update – allows updating the state,

View – interprets state using HTML.

This architecture has a similar approach to state modification as famous state container called Redux. Redux is also used for frontend applications, most commonly together with the React view library.

In oppose to React and Redux, Elm language was built with an architecture pattern in mind from the beginning. According to the official website[5], Elm is up to two times faster than the same application written in React.

The application is fully rendered locally in the user's web browser. That makes the server and client applications entirely independent of each other. Development of both parts is, therefore, break up into two separate repositories.

1.2 System architecture

As was briefly described in previous sections, the system consists of two independent parts.

In the following section, I will describe the architecture of the server-side part of the project in deep. After that, the next section is dedicated to the front end application.

1.3 Server-side application

As mentioned earlier, the server part of the application is written using Haskell programming language using Scotty web framework. The application exposes public Representational State Transfer (REST) API over HTTP protocol.

To achieve modularity and lose coupling, the server application is split into multiple modules. Those modules are grouped into logical partitions based on their purpose. The most significant parts are:

- Handler layer,
- Service and Data Transfer Object (DTO) layers,
- Model and Data Access Object (DAO) layers.

1.3.1 Handler layer

A handler is responsible for processing API requests. It directly interacts with Scotty framework, transforms incoming data into strongly typed objects and orchestrates other layers to evaluate response.

Handler groups together multiple endpoints. Those endpoints are uniquely registered to a specific Uniform Resource Locator (URL) and HyperText Transfer Protocol (HTTP) method. In terms of the server application, we refer to such pair as a *route*.

```haskell
-- Registering HTTP method with its route
createEndpoints :: BaseContext -> ScottyT Text BaseContextM ()
createEndpoints context = do
    get  "/questionnaires"          getQuestionnairesA
    post "/questionnaires"          postQuestionnairesA
    put  "/questionnaires/:qtnUuid" putQuestionnaireA

-- Questionnaire detail endpoint
getQuestionnaireA :: Endpoint
getQuestionnaireA =
    checkPermission "QTN_PERM" $
        qtnUuid <- param "qtnUuid"
        eitherDto <- getQuestionnaireDetailById qtnUuid
        case eitherDto of
            Right dto -> json dto
            Left error -> sendError error
```

Listing 1.1: Routes definition and route endpoint (simplified)

Once the correct handler for the requested route is selected, it starts processing data from the request. Such data might be URL (or query) parameters and/or request body. Those data are transformed into language primitives (such as `Int` or `String`) or complex objects called DTO (explained later). In case of invalid or malformed data, the request is aborted immediately with appropriate error information in response.

In addition to that, some routes may also require several permissions in order to be executed. Those routes need requests to be authenticated using Javascript Web Token (JWT) technology and user tied with given token must be granted such permissions.

If the request is validated successfully, required data are passed into a service layer where the actual business logic happens.

1.3.2 Service layer

Services are responsible for application logic. This means that its public interface is exposed to other layers using DTO. Private functions and internal dependencies (such as data persistence) are thus *implementation detail* of the layer itself and do not affect its interface.

In DSW, service layer takes care of basic Create, Read, Update, Delete (CRUD) operations over persisted entities. This includes application objects, for example, `User`, `KnowledgeModel` or `Questionnaire`.

In addition to data persistence, this layer also manages application configuration, knowledge model migrations, DTO mapping and maintains data consistency. To keep the public interface simple, services are usually composed using other services or DAO.

Services make operations based on given input data (objects identifiers, DTO, ...). Operations itself then converts DTO objects into internal representation (persisted object) and do computations on it. As a result of all computations, the DTO object is returned from service.

1.3.3 DTO

While describing the Service layer earlier in this chapter, there were multiple references to objects called DTO. These objects' goal is to create a framework-independent interface between layers. DTOs are plain objects[5] representing standard model objects, usually in many ways.

Model objects may be complex structures containing a significant amount of data. Such complexity may or may not be appropriate for some service operations.

While presenting a list of data (populated by DTO) to the user, it is not necessary to fetch all nested objects recursively. Instead, simplified objects may be used. Additional data may then be requested on demand. On the other hand, when the user wants to see the detail of some list item, the DTO object should contain enough data to fulfill the user's expectation. For such usecase, there would be two different objects, and the service layer would choose the one which best fits user expectation.

In DSW this approach is used to display either list or detail of questionnaires, knowledge models or knowledge model customizations. Conversion between model and DTO objects is done using specific services.

1.3.4 Authentication

In section 1.3.1 about HTTP requests handling, I briefly discussed endpoint authentication using JWT. In this section, the reader will be acquainted with how these tokens work.

Even though some endpoints (such as login or registration) do not require the user to be authenticated, the vast majority of the application is based on managing private data. Therefore, the user needs to be authenticated using

[5] By plain objects, we usually mean structures, which are not restricted by any framework and are built only constructs available in the language itself or by other plain objects (using composition). The same pattern is available in other languages. In Java programming language, this concept is referred to as Plain Old Java Object (POJO)[6].

```haskell
-- Define DTO
data QuestionnaireDetailDTO = QuestionnaireDetailDTO
    { uuid           : String
    , knowledgeModel : KnowledgeModelDTO
    , replies        : [ReplyDTO]
    }

-- Convert Model object to DTO
toQuestionnaireDetailDTO :: Questionnaire
                         -> QuestionnaireDetailDTO
toQuestionnaireDetailDTO model =
    QuestionnaireDetailDTO
        { uuid = model ^. uuid
        , knowledgeModel = model ^. knowledgeModel
        , replies = model ^. replies
        }
```

Listing 1.2: DTO definition and transformation from the Model object

username and password before he or she can access these data. Similarly to data persistence, also authentication may be done in many ways. There is basic authorization[6], API keys, JWT and many more.

The DSW application uses JWT tokens, which stands out by having the ability to be verified for issuer authenticity. Thanks to this, the issuer (in this case the server application) can embed custom payload into token and be sure those data will not be modified by an unauthorized person.

To become authorized, the user first has to log in to the application using username and password. If those credentials are correct, the server will issue an access token and send it back to the user. For all request requiring authorization, the user has to send issued token in the request header. The server will read the payload, verify token integrity and check the user's permissions. If the token is valid, the appropriate handler is called.

All JWT tokens have the same structure (pictured in figure 1.1). The token consists of three parts: `Header`, `Payload`, and `Signature`[8].

The header contains information about the token type and hashing algorithm used to generate the signature. The payload is an arbitrary JSON object containing publicly visible data (such as user identifier). Token signature is created by hashing header and payload (both encoded using the Base64 algo-

[6]Base64 is an HTTP authentication used to encode user's credentials into request header directly. Such aan pproach is considered as insecure as the credentials are easily captured by an unauthorized person[7].

Figure 1.1: JWT token structure example

rithm) by hashing algorithm specified in the header and also encoded using Base64.

These three parts are combined using period (.) character and returned as a `String`.

1.3.5 Authorization

In modern application, authenticating users is not enough. We might want a system to support a wide range of users hierarchy, capabilities and responsibilities. In DSW this is solved using combinations of roles and permissions.

Each user has one of the following roles:

- Administrator,
- Data Steward,
- Researcher.

Once the user entity is created, it has assigned a role and a default set of permissions. Having two levels of authorization is a vital system feature as the user's account might gain or lose privileges over its lifetime.

Permissions are resource specific, and for each resource, we might distinguish between multiple permissions (for example read and write permissions). Default permissions are system-wide configurable using a configuration file (which is loaded at startup time).

In the default environment, *Administrator* has all possible permissions including management of the users, organizations, and content. The *Data Steward* – the role in the hierarchy just under the Administrator – has similar permissions but is unable to manage organizations and other users. The *Researcher* can see knowledge models and public questionnaires but is able only to modify the content he created.

As mentioned earlier in section 1.3.1, permissions are checked in handler before any business logic happens. Each handler has defined which permission is required (for example `Read questionnaire`). Once the user makes a request, handler checks authentication state and search user's permissions

(demonstrated in code example 1.1). If user the has granted required permissions, request processing continues. Otherwise, the request is aborted immediately.

1.3.6 DAO

DAO is a way to access persisted data using a simplified interface. In general, application data may be persisted in many ways. The most common approach to this is using a database[9]. However, even database persistence may be implemented in different ways.

On the market, there are several options available. There are various kinds of databases: Graph databases, Relational databases, Document databases, and many others – moreover, all of those offered as free to use as well as business solutions.

In DSW, the Mongo DB was chosen as a database for persisting data. Mongo DB is a NoSQL document database (explained later), which allows to storing data using nested structures.

The goal of DAO is to encapsulate such technical detail from other application layers. The underlying database may change over time, but as long as the data model stays the same, the only affected layer will be DAO.

In Haskell, such layer is implemented using independent modules where each module manages one resource (Mongo DB collection). The module public interface is implemented using free functions. Those functions offer high-level API such as `findAll`, `findById` and similar for all CRUD operations.

```haskell
-- Finds questionnaire by its ID
findQuestionnaireById :: String
                      -> Context (Either AppError Questionnaire)
findQuestionnaireById id = do
    let action = findOne $ select [ "id" =: id ] collection
    maybeQuestionnaireS <- runDB action
    return . deserializeMaybeEntity $ maybeQuestionnaireS

findQuestionnaires :: Context (Either AppError [Questionnaire])
findQuestionnaires = ...
```

Listing 1.3: DAO module example for Questionnaire entity

1.3.7 Mongo DB

Mongo DB was chosen as a persistent layer in the early development of DSW. As stated earlier in section 1.3.6, Mongo DB is a document database.

By document database, we understand system which can store hierarchical tree structures (so-called *documents*) composed using scalars (`String`, `Integer`, ...), hashable maps, arrays or nested documents[10]. These documents are identified using unique internal identifiers and grouped into collections.

On collections, we can run standard CRUD queries to manipulate with stored data. Mongo DB provides an API for querying objects using notation based on JSON. This notation allows querying on nested objects, aggregations, relations or regular expressions. Since JSON notation may be unnecessarily verbose, documents are internally stored in Binary JSON (BSON) representation[11].

```
1 // Find Questionnaire with id '729fb982'
2 db.questionnaire.findOne(
3     { "id": "729fb982" }
4 )
```

Listing 1.4: Mongo DB query for finding questionnaire using its identifier

In oppose to standard relational databases, collections and documents do not have an explicit schema. Therefore there might be stored almost any kind of data in collection at one time.

Another notable difference is in data normalization. Standard databases (based on Structured Query Language (SQL)) use data decomposition and normalization to achieve better performance and great organization of tables and columns. In the case of Mongo DB, related documents are usually tightly coupled together (using composition) and duplicated instead of relations using foreign keys.

As mentioned before, the data are internally stored in binary format which is not directly usable in Haskell. Therefore the database driver exposes two special Typeclasses[7] for encoding and decoding binary formatted data. Such typeclasses must be implemented by all types which will be stored in database.

In addition to coding typeclasses, the driver also exposes a type-safe interface to build CRUD queries. These queries are created using simple Domain Specific Language (DSL) as shown in example 1.3.

1.4 Frontend application

In this section, I will in detail discuss the approach of development of the frontend (client) application. As stated earlier in section 1.1.2, this part of

[7] More information about Haskell Typeclasses are available at https://www.haskell.org/tutorial/classes.html

```haskell
instance ToBSON Questionnaire where
    toBSON model =
        [ "uuid"           BSON.=: model ^. uuid
        , "knowledgeModel" BSON.=: model ^. knowledgeModel
        , "replies"        BSON.=: model ^. replies
        ]

instance FromBSON Questionnaire where
    fromBSON doc = do
        uuid           <- BSON.lookup "uuid" doc
        knowledgeModel <- BSON.lookup "knowledgeModel" doc
        replies        <- BSON.lookup "replies" doc
        return Questionnaire { .. }
```

Listing 1.5: Example of Typeclasses used to transform the Model object into binary representation

the application is written using functional language Elm.

Even though Haskell and Elm have similar syntax, the application structure is entirely different. Elm application entry point is a module called `Main`. This module's responsibility is to initialize the application in the browser window and set its state based on the current browser URL.

The application view is a function of the state. Every time the state changes, the view function is called with the latest state value and returns corresponding UI elements. The resulting view is then passed to Elm runtime.

The runtime will compare the given view with current Document Object Model (DOM)[8] and applies only appropriate modifications. Since rendering the whole DOM may be slow[12], Elm may group multiple changes and render them at once. This means the browser does not have to render the whole tree so often and the user experience may increase.

For an extensive application as DSW is, it is not possible to keep the architecture that simple. Instead, for each screen new module with the same architecture is created. In the rest of this thesis, I will refer to such modules as "subapplicatons" for clarity. This means that state update, view functions, and messages are created from scratch for each screen. To accomplish interoperation between subapplications and standard Elm architecture, the composition is used.

In this case, the composition means that every nested application screen state is managed by its superior screen. A similar idea is used for `update`

[8]HTML DOM is a tree of web document (page) objects. It is used to create dynamic HTML modifiable by JavaScript. More information about DOM is available at W3C documentation: https://www.w3schools.com/js/js_htmldom.asp.

function where superior screen calls nested's screen update function.

In the time of writing, the client is composed of six subapplications, namely: `Users`, `Questionnaires`, `KMEditor`, `KnowledgeModels`, `Organization`, and `Public`. In the following sections, I will shortly introduce the reader to each of these.

1.4.1 Users module

This module is primarily used by the system administrator. The administrator can list registered users, manage their profiles and roles.

For each user, the administrator may update the user's profile information (including email) and change password. The user also may be deleted or deactivated in order not to be able to use application furthermore.

The rest of the users are only allowed to update their profiles without being able to deactivate or delete it.

1.4.2 Questionnaires module

Questionnaires module has two parts available to all user roles. As each of those parts contains complex logic (such as network calls and state management), it is also separated into individual subapplications.

The first part is a list of existing questionnaires with the ability to create a new one. Each user can see either public questionnaires or private questionnaires he or she created. By selecting a questionnaire, user can see questionnaire detail.

The second part is a possibility to fill a questionnaire and generate the data management plan. User fills the questionnaire by answering a set of questions from the knowledge model which was selected during the process of creating the questionnaire. Some of the questions have assigned FAIR metrics. By selecting an answer to a question, its metrics may positively or negatively affect the overall questionnaire.

Since researched project state may change over time, the user can change its phase on the questionnaire detail page. As some questions are only desired in a specific project phase, this action will also probably affect set questions which need to be answered.

Questions are designed to be infinitely nested and composed using other questions. Nested questions are in terms of DSW called `Item Templates`. In order to answer such question, the user is required to create item answer and reply to all nested questions in it.

Another type of nested question is follow-up questions for answer item (from the single choice list). The application is designed to make follow-up questions optional for items which do not require it. Since questionnaires may quickly become complicated and hard to orientate in, follow-up questions are not visible until the appropriate answer is selected.

1.4.3 KMEditor module

In section 1.5 about knowledge models migration, I will describe how editing knowledge models is designed and implemented. In this section, I will describe how the UI of customizing knowledge models work.

`KMEditor module` is mainly designed for data stewards (even though it is available to other roles too). Stewards are allowed to list existing knowledge models customizations or create a new one. We can understand customization as a new branch of existing (or even new) knowledge model which contains changes from the superior knowledge model.

Together with listing existing customizations, the user can also create a new one. When creating a new one, the user is asked to either select superior knowledge model or start from scratch. In both options, a new screen is displayed with details of customization.

The editor allows managing the whole knowledge model structure mentioned earlier: Chapters, Questions, Answers, References, and Experts. All of these may be added, modified or deleted. Changes are synchronized to the server in a batch once the user decides to save the current version of the knowledge model explicitly.

Nodes overview is displayed in a hierarchical tree view where modified nodes are highlighted using appropriate color. Navigation in the knowledge model is done using either tree view on the left side of the screen or from the selected node overview on the right side.

Node overview displays configuration specific to each node type together with its nested nodes.

During the time of writing this thesis, new capabilities for the editor are currently in development by its maintainers. The first one allows to tag questions. It helps to categorize questions into logical groups by them. Later, while creating a new questionnaire, data steward can pick only questions having specified tags, which are relevant for the questionnaire.

The second one is a knowledge model preview. Currently, there is no way to quickly preview how questionnaire built on edited knowledge model will look like. To do that, the user is required to save and publish the knowledge model and then create a questionnaire to see the UI. In the new version, users will be able to see an overview of final then questionnaire right in the editor without a need to publish unfinished customization.

The editor UI is shown in figure 1.2.

1.4.4 Knowledge models module

This module partially groups features from both `Questionnaires` and `KMEditor` modules. The main difference is, that this screen directly operates with knowledge models, instead of customizations. This, however, is not a significant difference because all published customizations are listed in knowledge models

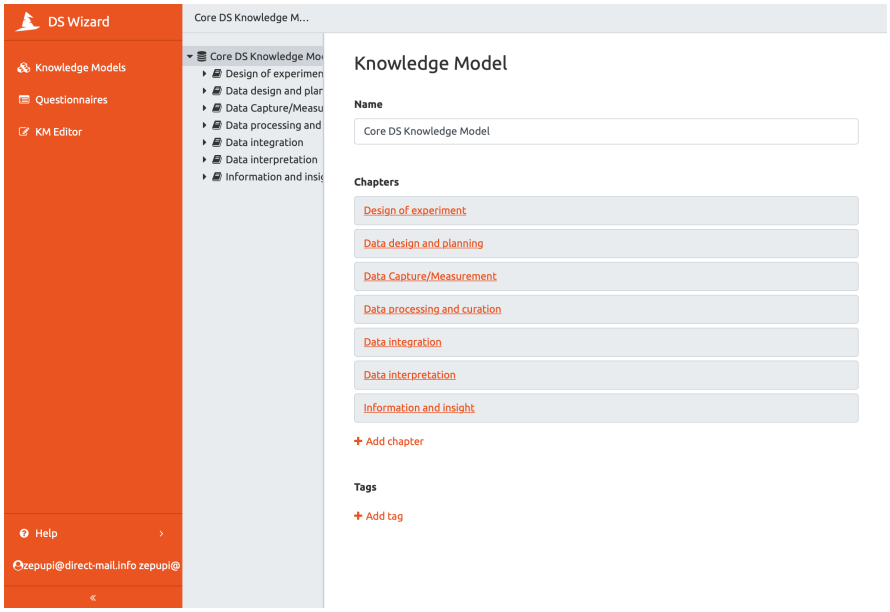

Figure 1.2: The knowledge model editor

too. For the rest of the features, all users may see knowledge models with its detail, create a questionnaire or new customization.

The last feature is related to system interoperability. Users are allowed to both import new knowledge models or export an existing one in JSON format.

This is an essential feature as base knowledge may, in theory, be created by anyone – either by individuals or enterprises. Such models then could be shared publicly and used in an arbitrary instance of DSW.

Even today, importing models is a very useful feature. The default installation package of DSW is currently shipped without any content. However, the initial base model (described later in section 1.5) maintained by Mr. Rob Hooft is shared publicly and can be easily imported.

1.4.5 Organization module

Organization management is a simple module which allows the administrator to manage necessary information (currently only name and identifier) of an organization operating the given instance of the application. This information is mostly used to identify knowledge models created by the organization and to add metadata to exported data management plans.

1.4.6 Public module

Public module groups all subapplications which are accessible without being logged in to the system. Those modules are:

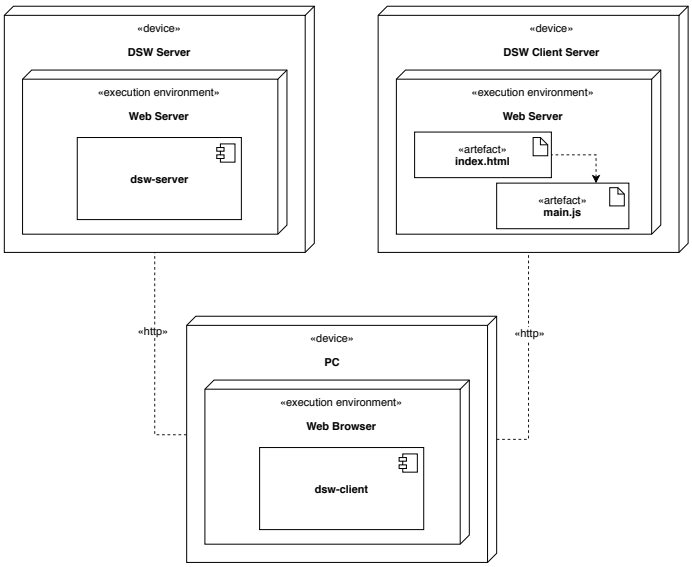

Figure 1.3: The system architecture

- Login,
- Registration,
- Public questionnaire.

The login module is predominantly self-explanatory. It is and landing page for not-logged-in users who are prompted to enter credentials. After a successful login, the server will issue a JWT access token which will authorize the user to see private pages.

Registration module is similarly straightforward. It is designed for users who are not signed up to the system yet and wants to use it. After successful registration, the user is required to confirm his email address and activate the account. Once the profile is activated, the user able to log in to the system.

Last part is a public questionnaire. This serves as an application demo which allows unauthenticated researchers to fill a prepared questionnaire. Since the public questionnaire is for demonstration purpose only, its data answers are not stored anywhere and will be forgotten once the user leaves the screen.

1.5 Knowledge model migrations

Initially, the DSW portal was created as a user-friendly alternative to knowledge model management which was previously built using JSON notation.

This notation was initially created and maintained by Rob Hooft[9] and is still used as core knowledge model in DSW.

As knowledge models may change over time, it is required to keep track of all possible versions created in the past. In terms of DSW, such process of modifying knowledge model is called migration. The migration process was designed and implemented by Ing. Vojtěch Knaisl as a part of his master thesis[13].

Migration consists of two parts: modification and upgrade. In the modification part, data steward may do CRUD operations on all knowledge model nodes (this includes chapters, questions, answers, references, and experts). Once the modifications are done, data steward publishes a new version of the knowledge model, so the modifications are available to other stewards.

The second part, upgrade, is done on knowledge model customizations. These customizations may be for example localizations, which needs to reflect changes in its parent knowledge model. During the upgrade, the system asks data steward step by step for all knowledge model changes coming from parent knowledge model. Since modifications on a single node may be done on both customizations and parent knowledge model, conflicts may occur.

In case of conflict, the user is asked to solve it manually. There are two options for the user to solve a conflict. User may either accept incoming change (which will discard customization changes) or reject the incoming change (which will in oppose prefer customization changes). Changes which do not cause conflict are merged automatically without user interaction.

Once all conflicts are resolved, data steward is asked to finalize the migration by publishing new a version of customization.

On top of knowledge models, researchers create and fill questionnaires which are later used to create data management plans. Questionnaires are currently tightly coupled to knowledge model they were initially created on. This, however, means, that once a new version of the knowledge model is released, a new questionnaire has to be created and filled from scratch.

As a result of this master's thesis, researchers will be able to migrate their questonnaires to newer version similarly as data stewards can migrate knowledge models.

1.5.1 Event-driven architecture

In previous section 1.5, I discussed the idea behind having multiple versions of the same data source at once. In this section, I would like to describe the technical details of that idea briefly.

In system representation, knowledge models are nothing more than a sequence of modification events. Those events are strictly defined and must

[9]Rob Hooft is a manager of Netherland node of ELIXIR group (ELIXIR NL). Mr. Hooft created the original knowledge model data source which is currently used in DSW as an optional component.

always be performed in the order they initially were created. The single event represents one specific modification. This modification may be "Add chapter", "Add question" or "Remove answer". Together, the system supports exactly twenty events where each of these events contains additional metadata. In Haskell, such structure is represented using multiple constructors of the same data type (example 1.6).

```haskell
-- Knowledge model modifications events enumeration
data Event
    = AddKnowledgeModelEvent' AddKnowledgeModelEvent
    | AddChapterEvent' AddChapterEvent
    | DeleteChapterEvent' DeleteChapterEvent
    | AddQuestionEvent' AddQuestionEvent
    | ...

-- New knowledge model meta data
data AddKnowledgeModelEvent = AddKnowledgeModelEvent
  { _addKnowledgeModelEventUuid :: UUID
  , _addKnowledgeModelEventKmUuid :: UUID
  , _addKnowledgeModelEventName :: String
  }
```

Listing 1.6: Event representation in DSW

Such an approach is generally known as *event-driven architecture*[14]. One of the main benefits for DSW is that by repeatedly applying the same events again, we always get the same result.

This works well with Mongo DB as a persistence layer. Instead of compiling (by applying events) knowledge models and storing the result in the database multiple times (for each customization, questionnaire, ...), only events are stored. When user requests compiled knowledge model, it is always compiled on demand and presented using DTO. Thanks to that, the user always has the latest possible version without problems with the inconsistency which would not be possible over time otherwise.

Another great feature of a chain of events is that it can be easily manipulated. For example, merging two customizations of knowledge model may be quickly done by applying one set of events at the end of the other chain. It also allows the user to *cherry pick* events and apply only a subset of incoming changes. Those properties are greatly utilized in the knowledge model migrations mentioned earlier.

1.6 Deployment

In previous sections, I described the architecture of the DSW distributed system. This section is dedicated to the production environment of this application.

Since both parts of the system are independent, deployment of the whole application is done in two steps. In one step, the server side application is deployed. As discussed earlier, Haskell application does not need application server (the requests are handled directly by the application) but depends on the persistence layer. To make the deployed application lose coupled, two Docker containers[10] are used to decouple application itself from the database. To interconnect the containers, Docker compose is used to make a private network between them. From the user perspective, the application behaves as a monolith which internally encapsulates its complexity.

In the second step, the client application is deployed similarly. Since Elm application runs in the user's internet browser, there is an additional container which handles HTTP requests and servers the application to the user. This container may be an arbitrary HTTP server. For the purpose of DSW, the Nginx HTTP server was chosen. Nginx is configured to return an empty HTTP document which links compiled Elm application. The actual logic (including routing) is handled locally in the browser.

The deployment schema is shown in figure 1.4.

[10]Docker is a development and deployment tool using containers to package up an application (together with its dependencies) and deploy it as a single package. Containers utilize hosts OS kernel instead of creating virtual a machine and virtual OS [15].

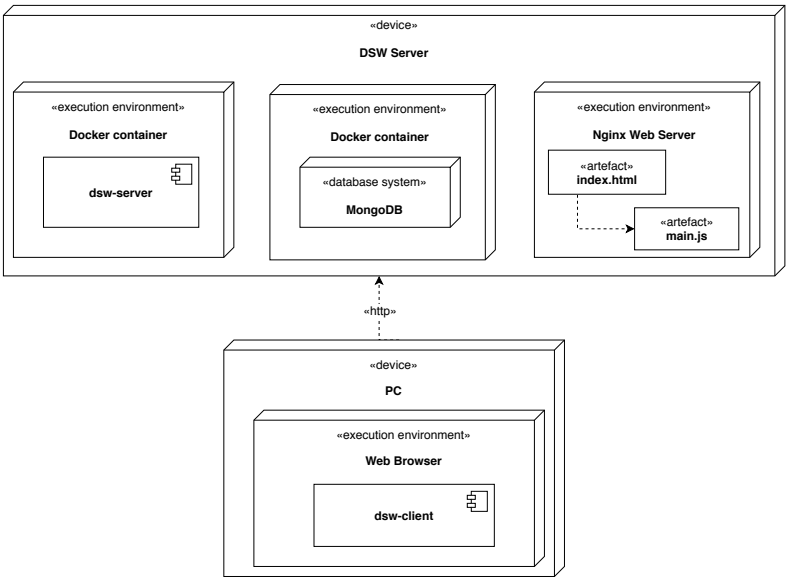

Figure 1.4: The application deployment schema

Chapter 2

Analysis

In this chapter, the reader will be acquainted with general requirements emphasized on the solution.

Firstly, I will identify the functional and non-functional requirements for questionnaire migrations and specify each requirement further.

Next, identified requirements are extended and defined further with specific use cases.

Then, I will describe individual scenarios for given use cases in detail.

Lastly, I will describe the domain model of the application, identify basic entities for such model and defined relationship between them.

2.1 Requirements

In this section, I will define functional and non-functional requirements for the DSW portal.

2.1.1 Functional requirements

This section will introduce a list of identified functional requirements to the reader. There are the following requirements:

FR1 – User is notified about available upgrade

User is visually notified when a newer version of the knowledge model is available for the given questionnaire.

FR2 – Create questionnaire migration

This requirement enables the user to create new questionnaire migration from current knowledge model to its newer version.

FR3 – Explore questionnaire on new knowledge model

User can see a preview of the migrated questionnaire before the questionnaire is fully migrated. This includes a preview of all questionnaire nodes together with the user's replies from previous knowledge model version.

FR4 – Guide user through the list of changes

User is guided by the system through changes relevant to the migrated questionnaire.

FR5 – Resolve question change

Question change may be marked as resolved, so the user does not have to remember which changes need to be reviewed before the questionnaire is migrated.

FR6 – Display node difference between versions

Questionnaire nodes are shown in both the old version and new version with a visual representation of textual changes (removed, changed or added characters).

FR7 – Mark question change for review

Allows user to mark questions which need to be carefully reviewed once the migration is done. This enables the user to adjust open-ended answers or choose a different choose different answer in the hierarchical tree.

FR8 – Migrate questionnaire

User is allowed to finish migration and upgrade questionnaire knowledge model version.

FR9 – Cancel questionnaire migration

Questionnaire migration which was not finalized by the user may be canceled without affecting the original questionnaire.

2.1.2 Non-functional requirements

Now, I would like to discuss the non-functional requirements of the system. These requirements specify entitlements for system and further define the functional requirements.

NR1 – User authentication

User is only allowed to create, modify, finish and cancel migration of questionnaires belonging to him. The system administrator is allowed to migrate an arbitrary questionnaire. Unauthenticated users are not authorized to access the migration feature at all.

NR2 – Migrated question consistency

The question may be either unmarked, marked as resolved or marked for later review. Other states are illegal.

NR3 – Preserve existing replies

If the user's reply to a question is compatible with the new knowledge model (for example, the question type did not change), it must be preserved.

NR4 – State synchronization

Question state is synchronized automatically without user interaction.

NR5 – Migration interoperability

Questionnaire migrations are integrated into the existing system and are compatible with its domain model.

2.2 Use cases

Based on identified requirements from the previous section, I would like to further discuss application use cases covering these requirements. The use cases are initiated by authenticated users without the need for increased permissions. All identified use cases together with short description are listed below.

UC1 – Initiate questionnaire migration

Enables user to start the questionnaire migration process in case he is notified about new knowledge model version availability.

UC2 – Cancel questionnaire migration

Enables user to discard process of the questionnaire migration without affecting the state of the initial questionnaire or its replies.

UC3 – Display migration change context

Enables user to navigate through the new knowledge model version in the context of filled in questionnaire replies.

UC4 – Change migrated question state

Enables user to modify changed question state to one of the following states:

- Initial state – changed question without state modification,

- Needs review – a change needs to be reviewed after the migration is finalized,
- Resolved – a change does not affect the user's previous answer.

UC5 – Migrate questionnaire

Enables user to migrate questionnaire to the new version of the knowledge model.

2.2.1 Requirements coverage

Requirements coverage ensures that all functional requirements identified in section 2.1.1 are covered by defined use cases. Table 2.1 shows how all identified requirements are covered by use cases.

functional requirement / use case	FR1	FR2	FR3	FR4	FR5	FR6	FR7	FR8	FR9
UC1	•	•							
UC2									•
UC3			•	•		•			
UC4					•		•		
UC5								•	

Table 2.1: Coverage of functional requirements by use cases

Figure 2.1 shows the use cases diagram together with relationships between them. Since questionnaire migration feature is integrated into the system only for authenticated users, the diagram contains exactly one actor. The analysis of non-functional requirements shows that all authenticated users (regardless of role) are allowed to manage migrations. Therefore only one role is used.

2.3 Scenarios

In previous sections, I described which requirements are relevant to questionnaire migrations. Requirements were validated against use cases. In this section, the reader will be further acquainted with use case execution demonstrated on scenarios.

Scenarios are presented in a step-by-step form where each step represents interaction with the system.

2.3.1 Scenarios actors

Before describing particular scenarios, I will briefly introduce actors participating in these scenarios. Scenarios have the following actors:

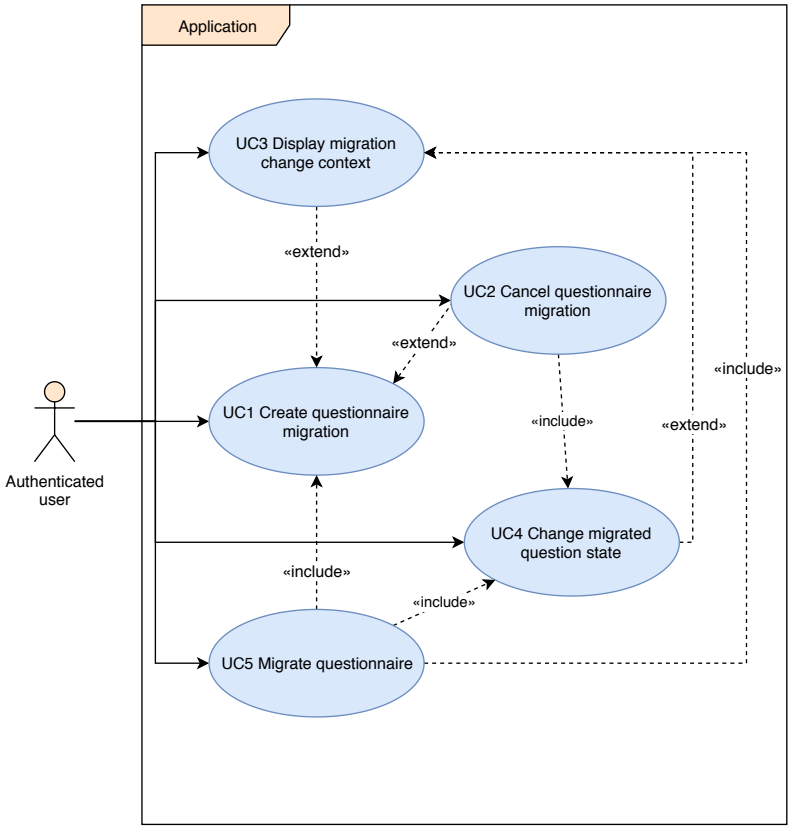

Figure 2.1: Use cases relationship diagram

- **User**

  An actor interacting with them *system* in the role of an arbitrary authenticated user.

- **Application**

  An actor who represents both client and server side of the system. It acts as a black box from the client perspective. Its tasks may internally consist of communication between these two parts.

2.3.2 Scenarios description

This section contains descriptions for individual scenarios. Roles described in the previous section will be highlighted using the *italic* text style.

All use cases start in an implicit state where the user is authenticated and located on a screen with a list of questionnaires.

The scenario for UC1 – Initiate questionnaire migration

1. The *user* is notified about available migration.
2. The *user* selects an upgrade option offered by the *application*.
3. The *application* prompts the `user` to select which of the available versions of the knowledge model wants to migrate to.
4. The *user* selects the desired version.
5. The *user* confirms the selection.
6. The *application* creates questionnaire migration and shows it to the *user*

Output state: The migration process is started and the *user* sees a preview of a first change.

The scenario for UC2 – Cancel questionnaire migration

Input state: The *user* created a migration as described in the UC1.

1. The *application* notifies the user that there is a migration in progress on one of the *user's* questionnaire.
2. The *user* selects the option "Cancel migration" displayed along with other questionnaire actions.
3. The *application* will discard all data related to the questionnaire.
4. The *user* is notified that there is an upgrade available for the questionnaire.

Output state: The *user* reverted questionnaire to the initial state. The *user* is at the same state as at the beginning of the scenario for UC1.

The scenario for UC3 – Display migration change context

Input state: The *user* created questionnaire migration according to the scenario for UC1.

1. The *application* displays detail of a first change in the migration.
2. The *user* expands the hierarchical structure overview.
3. The *application* displays a detailed hierarchy structure and highlights currently displayed change.
4. The *user* navigates through to the hierarchy to see the context of the change.

The scenario for UC4 – Change migrated question state

Input state: The *user* created questionnaire migration according to the scenario for UC1.

1. The *user* selects the option "Continue" migration next to other questionnaire actions.
2. The *application* displays the questionnaire migration process focused on the first change.
3. The *user* navigates through a list of changes until he sees the first change of either question or its answer.
4. The *application* offers to either resolve change or mark it for later review.
   (a) The *user* selects the *resolve change option*.
   (b) The *user* selects the *review change later* option.
5. The *application* saves the state and notifies the user about the new state.

Output state: The *application* stored the updated state so the *user* will see the new state next time he uses the application. Such state may be undone.

The scenario for UC5 – Finalize questionnaire migration

Input state: The *user* updated the questionnaire migration state according to UC4 several times.

1. The *user* selects the option "Continue migration" next to other questionnaire actions.
2. The *application* displays the questionnaire migration process focused on the first change.
3. The *user* selects "Apply migration" action.
4. The *application* creates a new copy of the questionnaire on the new version of knowledge model.
5. The *application* displays a list of questionnaires with the original questionnaire (build on the old version of the knowledge model) and a new questionnaire (build on the version).

Input state: The *user* has one more questionnaire in the list of questionnaires.

2.4 Domain model

After the analysis of requirements, use cases and their scenarios, I will further introduce the domain model. By domain model, we understand a conceptual model of the domain that incorporates both behavior and data[16].

Based on the previous analysis, I identified the following entities:

- QuestionnaireMigration,
- DiffEvent,
- Questionnaire,
- KnowledgeModel,
- QuestionFlag,
- Chapter,
- Question,
- Reply.

Figure 2.2 shows all identified entities and relationships between them. Some of the entities already exist in the DSW; my analysis was therefore also based on Ing. Vojtěch Knaisl's master's thesis[13] previously mentioned in chapter 1. Such entities are explicitly marked in the following list of entities.

2.4.1 QuestionnaireMigration entity

This entity represents the migration itself. It is composed using all entities which are required to create the migration context. The requirements demand that there must be at most one migration for each questionnaire in one moment.

Migration is created by the user when he wants to use a newer version of the knowledge model in his questionnaire.

2.4.2 DiffEvent entity

This entity represents an event which occurred during the knowledge model customization. Events are further described in section 1.5.1. Data stewards create events while making customization to the knowledge model in `KMEditor` module (discussed in 1.4.3).

The list of events is used to create a set of changes which are presented to the user.

This entity is already part of the application.

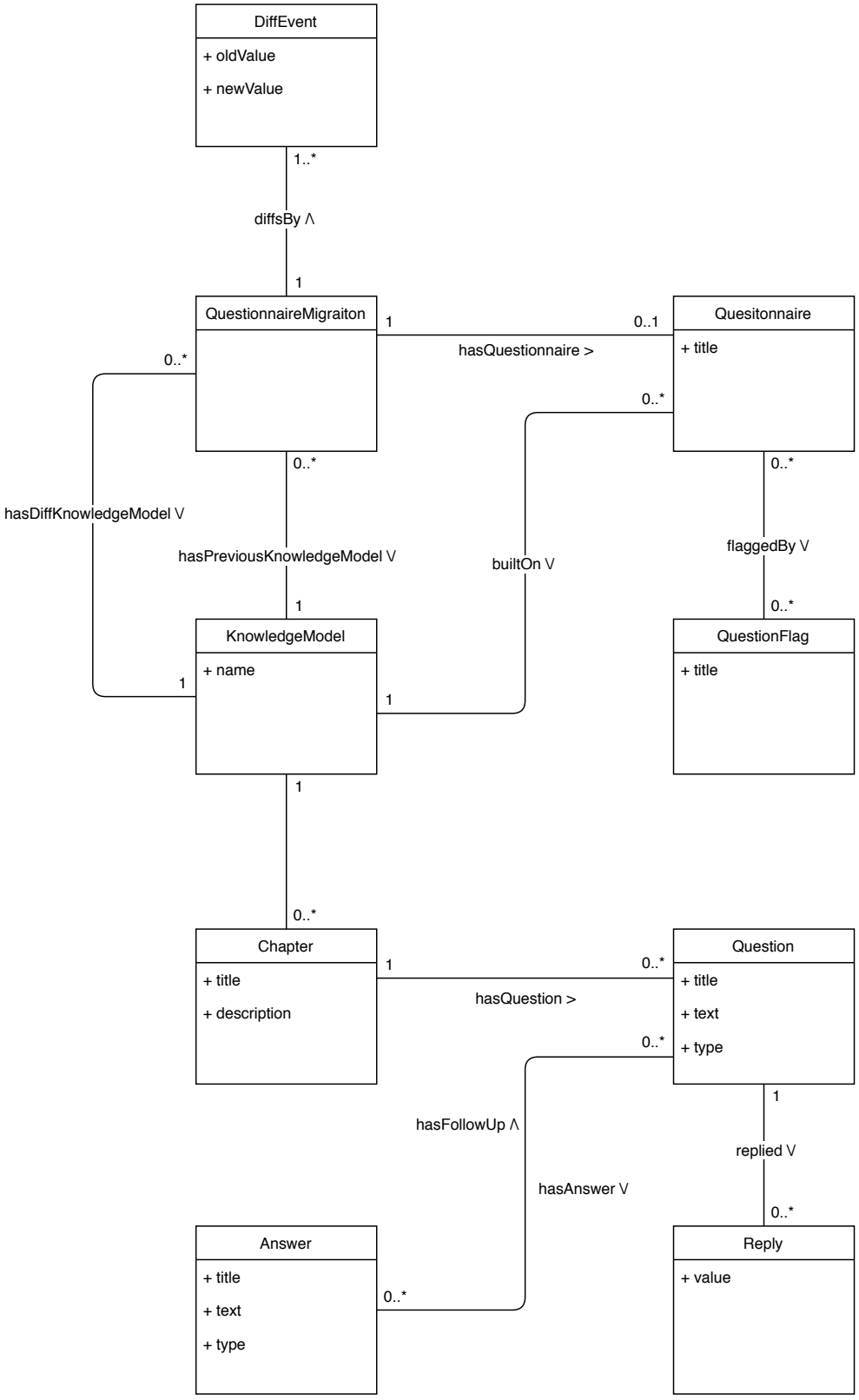

Figure 2.2: Domain model with relationships between entities

2.4.3 Questionnaire entity

The `Questionnaire` entity represents currently migrated questionnaire. It is created by an arbitrary user in the `Questionnaires` module (described in 1.4.2).

`QuestionnaireMigration` module uses questionnaires to make a preview of changes in the context of the user answers.

2.4.4 KnowledgeModel entity

Knowledge models contain the hierarchical structure of chapters, questions, and answers. In the migration, it is used for comparisons between new and previous versions of the knowledge model.

There is a unique structure called `DiffKnowledgeModel` used in the migration process. It has the same structure as the migrated knowledge model but also contains nodes, which were deleted – this enables the user to explore the context of nodes which will no longer be available after the migration is finished.

The `KnowledgeModel` entity is already part of the application.

2.4.5 QuestionFlag entity

Flags are used to represent question state. Since flags are not only used during migration but also in the questionnaire preview, flags are related to the questionnaire instead of migration.

2.4.6 Chapter entity

Chapters are used to group questions into logical parts. In addition to relation to questions, chapters contains a title and a description text.

This entity is already part of the application.

2.4.7 Question entity

Questions are in questionnaires used to collect data from the users. This entity is created by the application when creating (building) knowledge model from a list of events.

This entity is already part of the application.

2.4.8 Answer entity

This entity is used to represent an answer replied by user to a question. There are more types of question such as open-ended, single choice or structured (composed from other questions).

Single choice answers may also have follow-up questions which need to be replied to too.

The answer entity is already part of the application.

2.4.9 Reply entity

Reply entity represents the selected answer for questionnaire question. It is created by the user when filling up the questionnaire in `Questionnaires` module.

Non-functional requirements demand to preserve the reply if it is compatible with newer knowledge model.

This entity is already part of the application.

Chapter 3

Design

In this chapter, I will summarize the design of the implementation solution. Firstly, I will introduce the reader into the UI of the application. The application design will be presented using wireframes and further extended by actual screen designs in chapter 4.

3.1 User Interface

The user interface is a base building block for a modern application. The better the user's experience from the application is, the more likely the application will be successfull[17].

As a first step, while designing the application, it is common to create wireframes.

3.1.1 Wireframes

Wireframes are used to help understand the whole scope of the application while investing a minimum amount of time and effort into the actual design. More than that, wireframes help to verify that all use cases are covered with a possibility to correct or add missing scenarios or features quickly.

Before the actual design of the application is created, wireframes are often used to validate system or feature design by users. Such validation is called user testing and helps companies to effectively adjust the behavior and look of the application before it is fully designed or developed.

Figures 3.1 up to figure 3.6 shows wireframes developed for the migration tool.

Initiating the migration

The migration will be started from a list of existing questionnaires. If the questionnaire is based on the older of the knowledge model, there is a label

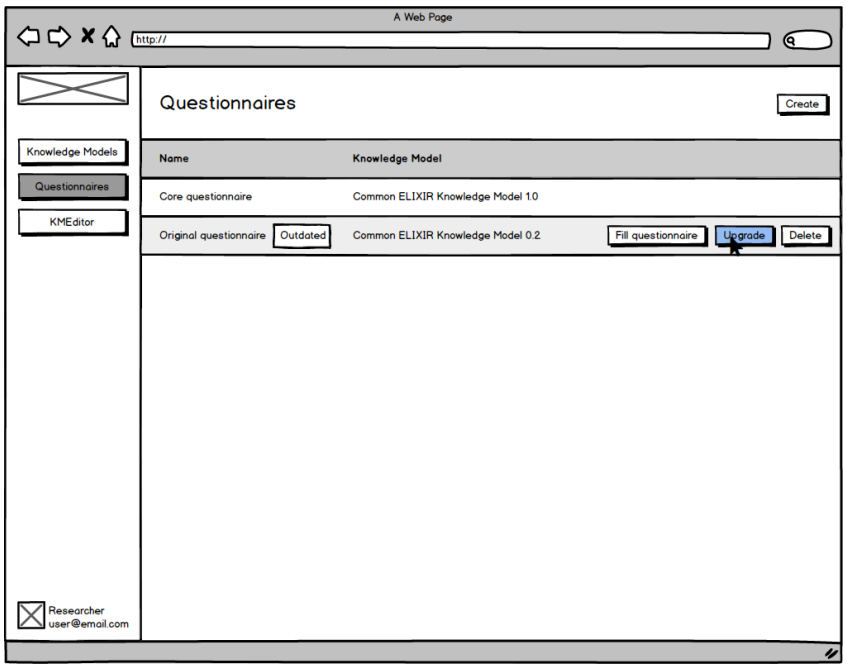

Figure 3.1: List of questionnaires with notification about available upgrade

notifying the user about available migration (figure 3.1).

The upgrade option is visible with the rest of the actions only when the questionnaire is focused (hovered by a mouse pointer). By selecting the upgrade option, the modal window is presented (figure 3.2) to the user asking to select to which version he wants to upgrade the questionnaire. It is intentional not to preselect none of the available versions (a especially not the newest one) as the user might want to upgrade to the newer version but not the latest one.

Migration overview

Once the user selected to which version he wants to migrate the questionnaire to, the migration process is started, and its overview is displayed. On the screen, the user sees three main panels (ignoring the base navigation panel on the left side – figure 3.3):

- Structure overview,
- Old questionnaire overview,
- New questionnaire overview.

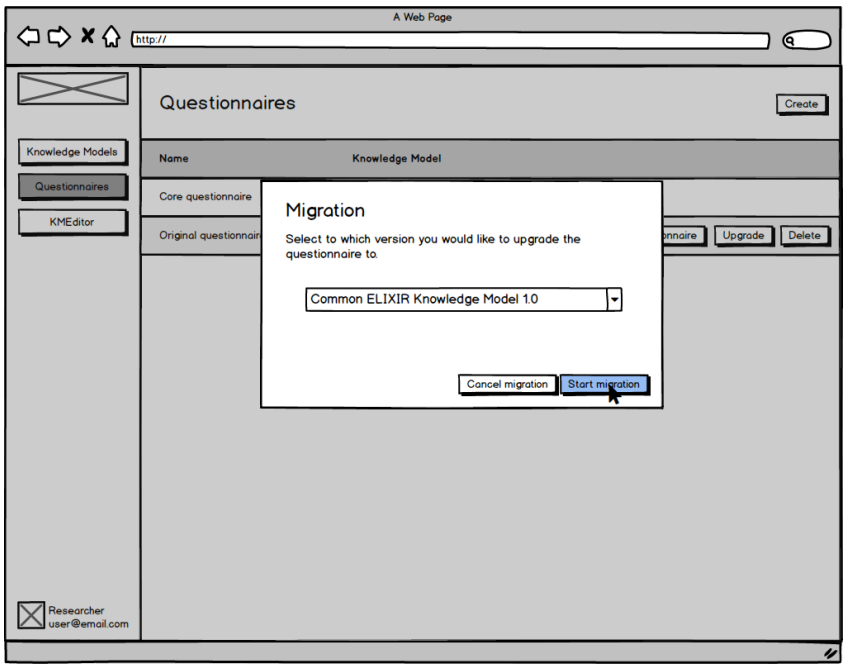

Figure 3.2: Modal dialog asking the user to select a newer version of the knowledge model

The structure panel is used to show the whole hierarchy of the questionnaire – all possible chapters, questions, and answers. By default, only chapters are visible so the user will not get confused by a significant amount of nodes.

The old questionnaire overview displays the questionnaire version from before the migration. This overview helps the user to see the difference between migrated versions. All textual changes are displayed using appropriate colors to make the difference highly visual. When a question node is selected, the user can see all possible answers with the highlighted user's reply. If the question is of type `Items template`, only the number of user's replies is visible.

The overview of the new questionnaire displays the questionnaire version from after the migration is finalized. The view has the same structure as the panel with an old questionnaire state, with one significant difference: when the user's reply is no longer applicable to the new version, the user is notified. This might be a meaningful change for the user. Therefore, he is allowed to mark question for a later review.

Once the question is marked,the user can no longer mark question until the current mark is not removed (figure 3.4).

To make it easier to explore all changes, athe pplication provides quick navigation between them using *previous* and *next* buttons (figure 3.3).

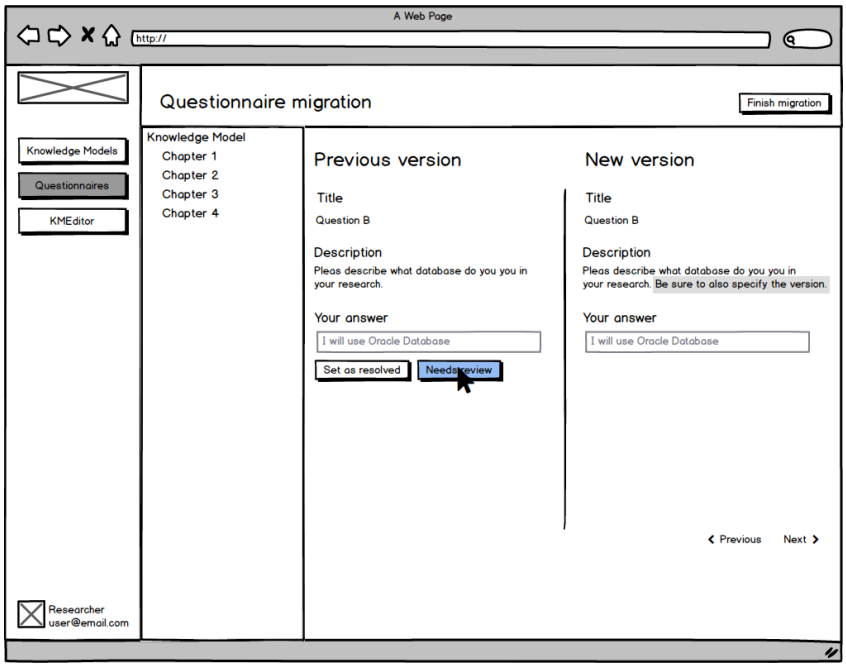

Figure 3.3: Questionnaire migration detail (focused on question difference)

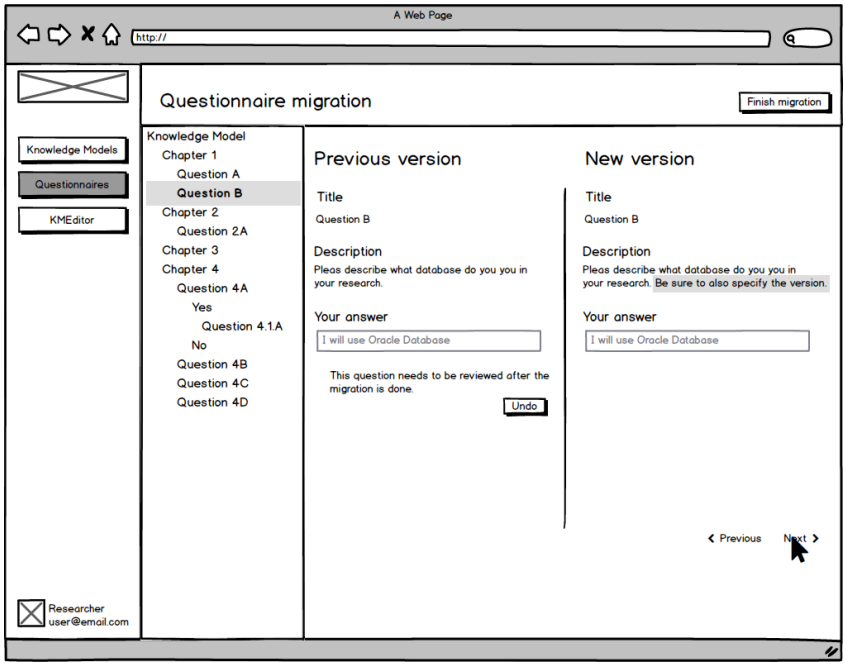

Figure 3.4: Updated state of the migrated question with expanded context

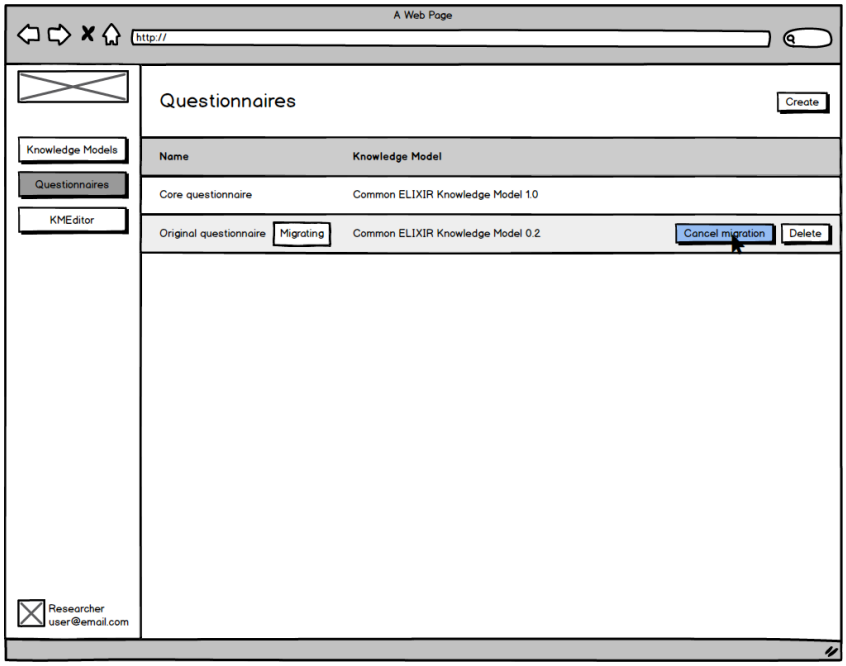

Figure 3.5: Questionnaire in the process of migration

Canceling the migration

During the migration process, the user is not able to fill the migrated questionnaire. There might be an exceptional case where the user either wants to update replies or generally cancel the migration. This can be done from the questionnaires list as shown in figure 3.5

Finalizing the migration

Once the user reviews all the changes which will be applied to the questionnaire, he can finalize the migration. By finishing the migration, a new questionnaire on the new knowledge model version will be available along with the original one. This allows the user to compare replies after the migration is done.

Use case coverage

Table 3.1 shows which use cases are covered by which wireframe. This table makes sure that all identified use cases were implemented into the final solution.

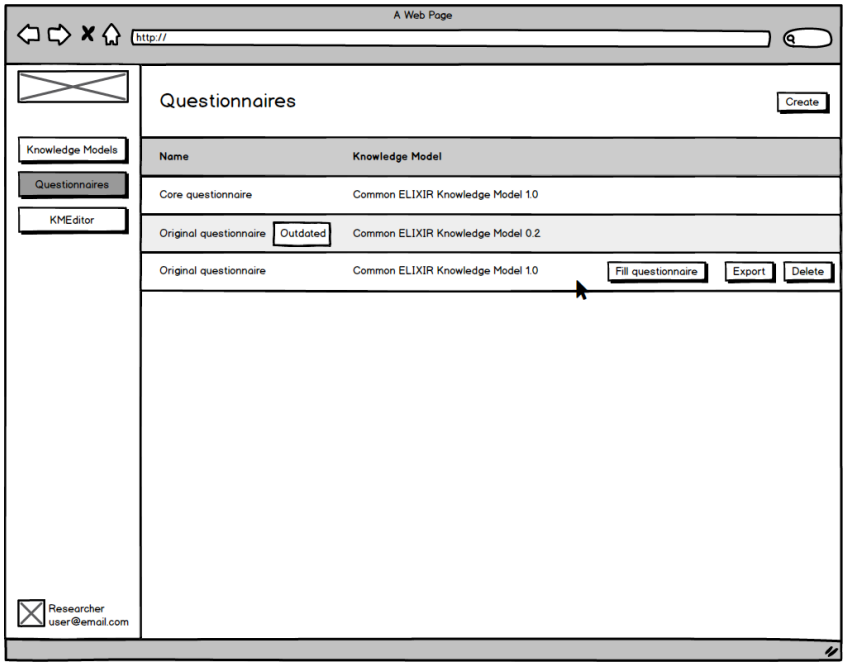

Figure 3.6: Migrated questionnaire together with its original version

use case / wireframe	UC1	UC2	UC3	UC4	UC5
WF1	•				
WF2	•				
WF3				•	•
WF4			•		•
WF5		•			
WF6					•

Table 3.1: Wireframe covering of identified use cases

3.2 Class diagram

In this section, I will introduce the class diagram for both the *DSW Server* and the *DSW Client* applications. This diagram is partially based on the current state and outcomes of the analysis in chapter 2.

The diagram is split into two parts – one for the server side and one for the client side. Main entities are then further described for better understanding of the design.

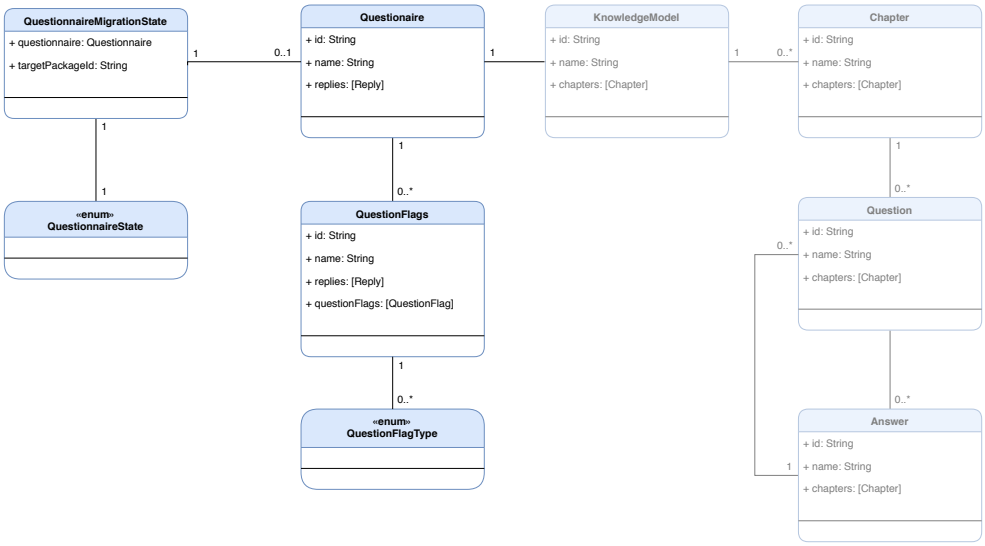

Figure 3.7: Server class diagram

3.2.1 Class diagram in the context of functional programming

In functional programming, the generally used term `class` is either not used at all or (in case of Haskell, for example) has an entirely different meaning. This, though, does not mean that complex data structures are not used.

Both Haskell and Elm programming languages have the concept of structured data called either `data` (in Haskell) or `type` (in Elm). These structures, however, do not encapsulate internal state and lack of methods.

To maintain consistency with terms used in the context of software engineering, I decided to use generally know terms such as *class diagram*. For such terms, the reader should always refer to this section to avoid misunderstanding.

3.2.2 Server class diagram

Firstly I would like to introduce a class diagram for the server application. The diagram is shown in figure 3.7. Entities on the left side the diagram are either newly introduced or existing entities which needed to be updated. On the right side of the diagram, there are entities (partially transparent) which were used in the solution but were not modified, thus serves only for completeness of the diagram.

QuestionnaireMigrationState

This entity is used designed to hold the state of the migration. It contains a deep copy of the migrated questionnaire and an identifier of the target knowledge model.

Initially, this entity was design to also contain a deep copy of both an old and new version of the knowledge model. The maintaining team of the DSW, however, removed knowledge model caching feature, so the knowledge models are always compiled from scratch[18].

Therefore, all needed knowledge models are referred to only using unique identifiers.

The state of the migrated question was added to the questionnaire itself as the state needs to be also preserved once the migration is completed. The deep copy of the questionnaire is then used to create a new copy of the questionnaire without modifying the original version.

QuestionnaireState

Questionnaire state is a simple enumeration of three primary cases:

- `Default`,
- `Migrating`,
- `Outdated`.

The `Default` state represents cases, where the questionnaire is based on the latest version of the knowledge model (and no migration is therefore available). Once the user creates a migration for the questionnaire, it is moved into `Migrating` state and preserved in such a state until the migration is either finalized or canceled.

The last possible state represents a questionnaire, which is based on an older version of the knowledge model and can be migrated to the newer one.

QuestionFlags

Question flags represent a state of migrated questions. Questionnaire holds a collection of `QuestionFlags` object, which is composed of question unique identifier and a set of question flags (represented by `QuestionFlagType.`).

Currently, at most one flag is allowed to be added to each question. `QuestionFlags` is however designed to keep an arbitrary number of flags.

This makes such feature future proof as more flag types may be easily added later. On the other hand, such behavior requires more sophisticated management of the flags because of integration constraints.

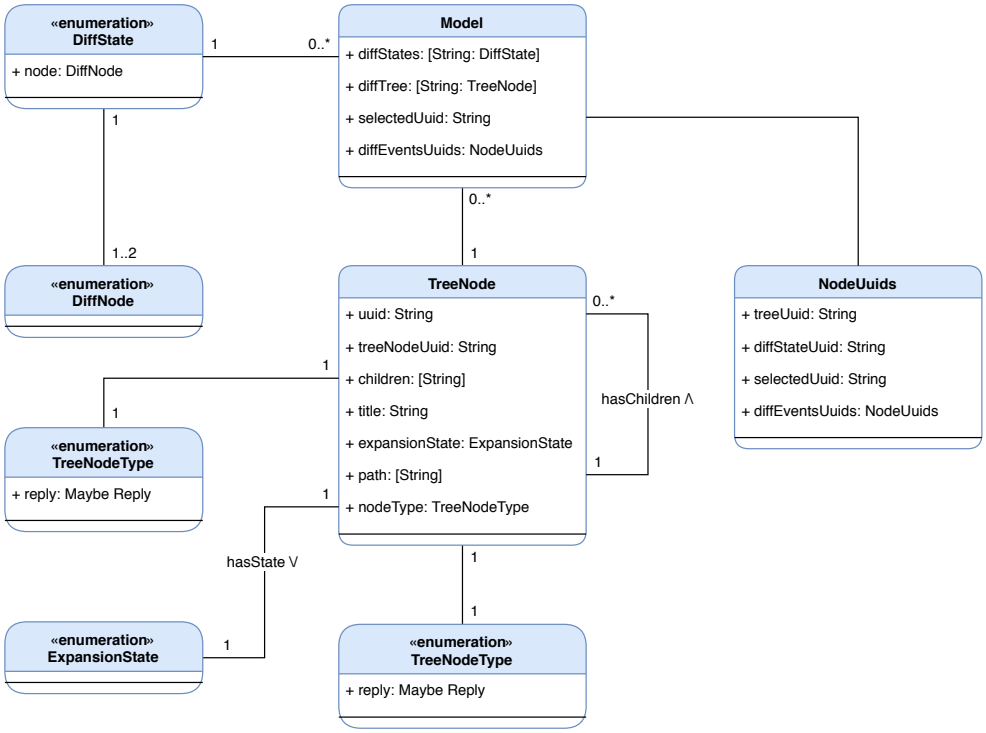

Figure 3.8: Client class diagram with the relationship between entities

QuestionFlagType

This enumeration represents all possible types of migrated question state. Currently, only two cases are implemented: `NeedsReview` and `Resolved`.

These cases correspond to states identified in functional requirements in section 2.1.1. As described in the previous section, these cases are currently mutually exclusive. Therefore additional business logic needs to keep data consistent.

3.2.3 Client class diagram

In this section, I will discuss the design of the class diagram of the client application. For simplicity, the diagram only shows entities which were newly added as existing entities do not have a significant impact on the design. The diagram is shown in figure 3.8 and is further described in the following sections.

Model

The `Model` entity is a base structure keeping the whole state of the questionnaire migration subapplication. The name was chosen according to Elm

architecture described in section 1.4.

This entity is used as a *source of truth* for the rest of the subapplication, and whenever it changes, the whole view structure must be recreated from scratch.

TreeNode

This entity represents a single row in the questionnaire structure overview (the left panel on figure 3.4). The `Model` entity keeps tree nodes in a lookup table so it can be quickly looked-up by its identifier.

Because the hierarchy view needs to display some additional nodes (not presented in the questionnaire directly), it uses the *node path* to represent the node location in the tree.

The path is a unique identifier which is made of node identifier composed with its all predecessor's identifiers. This allows having duplicate nodes in the tree in a deterministic way (by adding custom nodes to the path).

All nodes contain its expansion state, path, list of successors' identifiers and its type.

Children nodes are rendered using repeated lookups in the table of nodes mentioned above.

TreeNodeType

The node type entity is used to differentiate between questionnaire node types. This allows rendering each node with a sligthly different design, so it is visually recognizable to the user.

ExpansionState

The expansion state entity is used to tell the rendering function whether or not it should also render all of the node successors. This state may change every time the user toggles the expansion indicator.

DiffState

A `DiffState` entity represents a difference state of each node of the questionnaire. It is implemented as an enumeration with the following cases:

1. Unchanged,
2. Created,
3. Modified,
4. Deleted.

The `Unchanged` represents a node, which was not edited between migrated versions of the knowledge model.

The next type, `Created` is a constructor only accepting one node which represents the created node. When constructing such node, all texts in it are marked with the `Added` state so it can later visually br represented to the user. In this case, only the node in the right panel will be displayed.

The `Modified` case accepts two nodes: the old version and new version. Its texts are differentiated by letters, which allows to mark deleted and added letters. The old version contains original texts with marked deleted letters. The new version, on the other hand, displays the current version of texts with marked added letters.

Lastly, the `Deleted` node is the opposite of the `Created` node mentioned above. It only displays the node in the left panel with all texts marked as deleted.

This entity is directly created from the knowledge model customization event. Because customization events are not structured hierarchically, the difference state cannot be directly mapped to the tree node and does not contain a unique path.

DiffNode

This enumeration entity is used to created `DiffState` entity mentioned in the section above. It contains constructor for each questionnaire node type. The constructors are

- KnowledgeModelDiffNode,
- ChapterDiffNode,
- QuestionDiffNode,
- AnswerDiffNode.

Each of these constructors further accepts data structure specialized for each node type so it can be adequately rendered.

NodeUuids

This structure is used to map identifiers between composed path (used in `TreeNode`) and unique node identifier from the knowledge model (used in `DiffState`). It is an pair containing path as a one component and node identifier as secong. Thanks to that, there is a quick way to find difference state for all tree nodes and vice versa.

3.3 Server Interface

In this section, I will discuss what interface is needed to be provided by the server to enable all required features. The interface is described in the form of REST API endpoints. All endpoints are further defined in individual sections.

3.3.1 Create questionnaire migration

This endpoint is used to create a new questionnaire migration. The migration is created for the questionnaire, whose identifier is passed as a URL parameter.

The other parameter, `targetPackageId` is passed in the request body and represents the identifier of the newer version of the knowledge model.

- **URL**
  - `/questionnaires/:qtnUuid/migrations`
- **Method**
  - `POST`
- **URL Parameters**
  - `qtnUuid` – Migrated questionnaire unique identifier

3.3.2 Get existing migration detail

Endpoint used to retrieve information about existing migration. The migration is searched for by migrated questionnaire identifier, which is passed in as a `URL` parameter.

Returns an error code if there is no migration for the given questionnaire.

- **URL**
  - `/questionnaires/:qtnUuid/migrations`
- **Method**
  - `GET`
- **URL Parameters**
  - `qtnUuid` – Migrated questionnaire unique identifier

3.3.3 Delete existing migration

Cancels an existing migration by migrated questionnaire identifier passed in as a `URL` parameter.

- **URL**
  - `/questionnaires/:qtnUuid/migrations`
- **Method**
  - `DELETE`
- **URL Parameters**
  - `qtnUuid` – Migrated questionnaire unique identifier

3.3.4 Update migrated question state

Allows updating migrated question state by setting it into `NeedsReview` or `Resolved` state. The questionnaire is identified by an identifier passed in as a `URL` parameter.

The new state is passed in as a structure composed from question identifier and collection of question flags. When no flag is provided, the question is set to the default state.

- **URL**
  - `/questionnaires/:qtnUuid/migrations/resolveQuestionEvent`
- **Method**
  - `PUT`
- **URL Parameters**
  - `qtnUuid` – Migrated questionnaire unique identifier

3.3.5 Finalize migration

This endpoint enables the user to finalize existing migration and migrate the questionnaire from an older version of the knowledge model to a newer one. By calling this endpoint, a new copy of the questionnaire (with a new identifier) is created and saved along with the original version.

- **URL**
  - `/questionnaires/:qtnUuid/migrations`
- **Method**

  - `PUT`

- **URL Parameters**

  - `qtnUuid` – Migrated questionnaire unique identifier

3.4 Questionnaire structure

Questionnaire structure represents a view panel where the user can preview all nodes and answer paths. This structure is created from knowledge model nodes combined with the user's replies.

3.4.1 Creating the structure

The hierarchical structure is created using a lookup table keyed by string identifiers. The final lookup table is generated by recursively passing through a knowledge model and its successors.

Each successor is then also recursively inserting its successors until it founds a leaf (either question, answer or reply) of the tree. Nodes are added into the lookup table by its unique path in the tree. The path is given by chaining sequence of predecessors' identifiers combined using period (.) character. This helps to distinguish nodes with the identifier inserted into multiple locations in the tree while maintaining compatibility with the current implementation of replies identification.

3.4.2 Meta nodes

To accomplish a clear structure of the user's replies, some questions require to insert meta nodes which do not exist in the tree. Such a strategy is used for the template items question type, where the user is required to reply by filling structure of nested questions.

```
Common ELIXIR Knowledge Model
└─ Chapter - Design of experiment
   └─ Question - What database will you use?
      ├─ Q1 - What is the name of the database?
      ├─ Q2 - What version of the database?
      └─ Q3 - Is it an open-source database?
         ├─ Answer - Yes
         └─ Answer - No
```

Figure 3.9: Item template question type example

In example 3.9, the researcher might want to use multiple databases and therefore would reply by filling the template multiple times. This would create

a `Reply` entity for each question by inserting an index tag of the item into nodes paths. Knowledge models, however, do not support such paths and replies cannot be directly applied to it because of that.

Item templates replies

As mentioned in the introduction to this section, knowledge model must be combined with the user's to render meta items and answers correctly. For database question example mentioned above, the final rendered tree would look like the one on figure 3.10 for two replied items.

```
Common ELIXIR Knowledge Model
└─ Chapter - Design of experiment
   └─ Question - What database will you use?
      ├─ Item 1
      │  ├─ Q1 - What is the name of the database?
      │  ├─ Q2 - What version of the database?
      │  └─ Q3 - Is it an open-source database?
      │     ├─ Answer - Yes
      │     └─ Answer - No
      └─ Item 2
         ├─ Q1 - What is the name of the database?
         ├─ Q2 - What version of the database?
         └─ Q3 - Is it an open-source database?
            ├─ Answer - Yes
            └─ Answer - No
```

Figure 3.10: Tree representation for item template replies

By selecting any of the node nested in the item template, the user will be able to see the appropriate answer. There might be however the case, where the user did not reply by any item template, and therefore there is no meta node which would render nested questions. In such a case, the tree would have a *flat* structure as in figure 3.9.

3.5 Questionnaire overview

Questionnaire overview panel is used to visualize the difference between nodes from different questionnaire versions. This difference is modeled over previous and target knowledge model versions. Because each node is build using different data, its overview must be adjusted too.

3.5.1 Knowledge model difference

The only editable property of the knowledge model is its title. Therefore, the difference only shows to the user how its title's text changed (according to 3.2.3).

To achieve simplicity for the overview, nested nodes are not shown in this panel and are only visible in questionnaire structure as described in 3.4.

3.5.2 Chapter difference

Chapters are composed of title and detailed description (called `text`). As both of those properties might have changed, the user can see the character-by-character difference of these texts.

Similarly to knowledge models difference, nested nodes (questions) are left out for the screen simplicity.

3.5.3 Question difference

Question node contains information about its title and description (also called `text`). Similarly to chapters, both of these properties are displayed using character-by-character difference.

Because the overview should show most of the available questionnaire context to help the user get orientated, questions answers are displayed too. The displayed information is based on the question type. The description of how the answer differs for each question type follows.

Value question type

This type represents an open-ended question. While filling the questionnaire, the user is allowed to input an arbitrary text.

To achieve the highest similarity, the user's reply is rendered as a read-only value under the question as it would be in the questionnaire.

Options question type

Options represent a single-choice question. User replies to this question by selecting one of the offered answers.

Under the question difference, all answers are shown as read-only single-choice options. The options are rendered in the exact state they appear in the knowledge model. If the user selected a specific answer while filling the questionnaire, it would be marked.

Item templates question type

This type represents a question type described in section 3.4.2. Because item template may contain complex nested structures which would be confusing

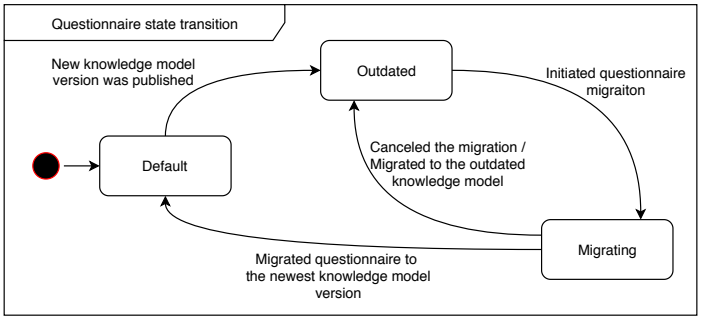

Figure 3.11: The state transition diagram of the migrated questionnaire

in this context, it was decided not to show it. Instead, the user can see information about how many answers he replied and is referenced to navigate to the left panel to see the nested questions.

3.5.4 Answer difference

The answer represents a single-choice option answer for the question. This node contains a label and advice texts. These texts are differenced as other texts in the overview panel.

Follow-up questions nested in the answer are not displayed directly in the overview and are only visible in the questionnaire structure.

3.5.5 Tags difference

In the client application analysis in chapter 2, I briefly introduced a new feature called *tags* used to group questions into logical parts. As this feature is not entirely done at the time of the writing, question tags changes are currently ignored.

3.5.6 Incompatible question answer

One of the non-functional requirements identified in section 2.1.2 demands the question replies to be preserved after the migration is finalized. This is, however, only possible if the question type was not changed.

If the question type was modified, the user is noticed that his answer is not applicable in the newer version and is recommended to mark question for later review.

3.6 Questionnaire states

To summarize how the questionnaire states may change, I prepared a state transition diagram shown in figure 3.11.

Chapter 4

Realization

In this chapter, I will describe the implementation details of the migration tool. Firstly, I will introduce tools and technologies for both client and server side used to implement the migrator. Then, I will go further and describe how the main parts on both platforms are implemented. Lastly, I will acquaint the reader with the final application design.

4.1 Used technologies

Now, I would like to introduce the reader to the technologies used to build the migration tool. As was already stated in the non-functional requirements in section 4.1, the tool needs to be fully compatible and interoperable with the current application.

4.1.1 Server application

I described the tools currently used in the DSW in the analysis in chapter 2. Here, I will briefly highlight the main parts.

Haskell programming language

The server part of the application is fully implemented in the Haskell programming language. Haskell is a purely functional programming language. It means that all functions are *pure* and all data are immutable[19]. By pure functions, we understand every function which is free of side effects. Thanks to side effects elimination, functions become more straightforward and more comfortable to reason about – each function will for the same input return the same output every time.

Because the entire server application was already built in Haskell, there was not much space for choosing a programming language.

The migration tool could be built as an individual service (often called microservice [20]) in an arbitrary programming language. This would, however, introduce unnecessary complexity in development itself, but also deployment and application management.

Because building microservice would mean to rebuild the vast majority of the existing application (an significant part of the model layer and the API layer), I decided to implement the migrator as a new module which is part of the existing code base.

Integrated Development Environment

Integrated Development Environment (IDE) is a software application integrating numerous tools for helping faster development[21]. Such application helps with code syntax highlighting, compiling, testing or even deploying the developed application.

To name a few, applications like Atom, Visual Studio Code or IntelliJ IDEA support development in Haskell[11].

For a project as extensive as DSW is, neither of those applications was working correctly. All of the mentioned suffered by lousy performance, invalid symbols recognition, and invalid error reporting.

After consultation with team members of the DSW maintainers, I decided to turn off all advanced language support and used only syntax highlighting in IntelliJ IDEA. Such disadvantage had unfortunately significant impact on the development time and orientation in the project.

Scotty web framework

The communication between the server and the client application is done using the REST API. The API interface is built on top of the *Scotty web framework*.

Scotty is a framework written in Haskell which allows to create type-safe API routing and provides convenient helper functions to parse HTTP requests.

Most of the work with integrating Scotty with DSW was already done when I joined the project. My only interaction with the framework was to register all supported routes for the migration tool and convert data between internal representation and public JSON.

4.1.2 Client application

Similarly to the server tooling, used technologies were already decided by the DSW maintainers at the beginning of the project. In the application analysis in chapter 2, I described in depth the client side application architecture and

[11] Haskell is usually not supported out of the box by IDEs. Instead, a plugin with language support and advanced features needs to be installed. Those plugins are usually based on either `ghc-mod` or `Intero` libraries.

its base modules. In this section, I would like to summarize tools and used technologies briefly.

Elm programming language

The entire client application is currently written in the Elm programming language (described in 1.4). Elm is a functional language with similar syntax to Haskell. Its base in building frontend application lays in unidirectional architecture called `Elm architecture`.

The term unidirectional describes how data are passed through the application. In Elm, the data are always passed one way, through its base architectural pattern represented by the `model`, `update` and `view`. The application state is represented by the `model`, changed using `update` function and then presented in `view`.

The great advantage of using Elm is that its functional approach may be shared and discussed with the server-side team because of syntax similarity.

Even though both languages are functional, there are few differencies[22]. I would like to point out the two most significant.

First one is lack of *Typeclasses* in Elm. Such disadvantage makes it harder to create generic constraints which are widely used in Haskell. This makes Elm `core` library more verbose as functions such as `map` must be implemented for each type individually.

On the other hand, Elm's record syntax is much more powerful than Haskell's. In Elm, all records may be automatically used as free functions (taking an object as argument and returning record value) out of the box. Records, thus, may be used in function composition, in combination with functions like `Maybe.map`. This makes the code easily readable and maintainable.

Haskell offers a similar feature by using `lenses` language extension; it, however, requires records to be structured in a specific way which makes harder to read.

Integrated Development Environment

After the struggle I had with choosing the right Haskell IDE, I decided to keep the Elm environment as simple as possible. I chose the Visual Studio Code (or *VS Code*, for short) for development as it is lightweight, and based on my own experience, it is faster and more responsible.

I used the VS Code together with *elm*[23] plugin which enables features like syntax highlighting, error reporting, type definitions and "jump to definition". This made my onboarding on the project much faster and development more convenient than doing the same things on the server side.

Node environment and Webpack

Because Elm runs in an internet browser, it needs some kind of a web server to handle requests and serve the application to the user. In production, open-source servers like NGINX[12] or Apache[13] are often used[24].

For development purpose, Elm offers a simple HTTP server called *reactor*. With reactor, the developer is able to run an arbitrary Elm source code in a browser.

The reactor is however shipped with its own HyperText Markup Language (HTML) template which means that all code used in application HTML will not be loaded. This includes stylesheets, custom JavaScript scripts or *ports* (Elm-to-Javascript interoperability API).

Together with Elm code, application sources also contains SASS which is compiled into standard CSS. Therefore, building the application is a multistep process – such process, however, can not be handled by the reactor.

There are many solutions to this problem, the maintaining team of DSW chose the `Node.js`[14] environment together with `Webpack` tool.

Webpack is used to compile all application sources with appropriate compilers and creates an application bundle from output. Webpack is also shipped with development server (as a separate tool) which is able to watch application sources and run compiler whenever the source change. In addition to that, It is also capable of refreshing the browser window, so the changes are visible immediately.

The production build is also done using Webpack which will also set production configuration to all bundled sources.

Text Difference

The functional requirements from section 2.1.1 say that the application should show the exact text difference for all migrated questionnaire nodes. Because there is no information about the migration differences stored on the server, this feature is fully implemented on the client side.

The problem of finding the difference between two given texts is called *string metric* (also known as *similarity metric* or *string distance*)[25]. There are multiple algorithms solving string distance problem. To name a few, there is the Myers' algorithm[26] and Wu's algorithm[27] which are based on the same idea. Wu's algorithm, however, achieves up to four times better performance for strings, which shares most of the strings' characters.

In migrations, I assume the string differences will be in most cases rewordings or typo corrections. Therefore the Wu's algorithm is an excellent fit for this problem.

[12] NGINX homepage: `https://www.nginx.com`.

[13] Apache HTTP server homepage: `https://httpd.apache.org`.

[14] Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

This algorithm is implemented in `elm-diff`[28] open-source library. For any given collection of elements, it will return a new collection with the difference information for each element. Such elements may be in the following states:

- NoChange,
- Added,
- Removed.

With a slight modification, it can be used to receive strings (which are not collections in Elm) directly. The output can be then used to render string character-by-character and highlighting it with the appropriate color. The usage of string difference (built on top of the `elm-diff`) is shown in 4.1.

```
1  -- String diff function declaration
2  diffStrings : String -> String -> Change Character
3  diffStrings left right = ...
4
5  -- Diff old and new versions of the string
6  diffStrings "modle" "model"
7  {- Outputs:
8          List(6)
9              NoChange 'm'
10             NoChange 'o'
11             NoChange 'd'
12             Removed  'l'
13             NoChange 'e'
14             Added    'l'
15 -}
```

Listing 4.1: String difference using `elm-diff`

4.2 Application structure

After a summary of the used tools, I would like to acquaint the reader with the structure of both server and client applications. The structure for each of the applications corresponds to sections listed in analysis in chapter 2. For a more detailed description of modules, please refer to the appropriate section in the chapter mentioned above.

4.2.1 Server application structure

Firstly, I would like to introduce the structure of the server application. The structure reflects structure beginning in the project root. Some directories in the project contain a significant amount of subdirectories, which are not crucial in the context of this thesis. These directories are omitted and replaced by an ellipsis (...) sign. The structure is shown in figure 4.1.

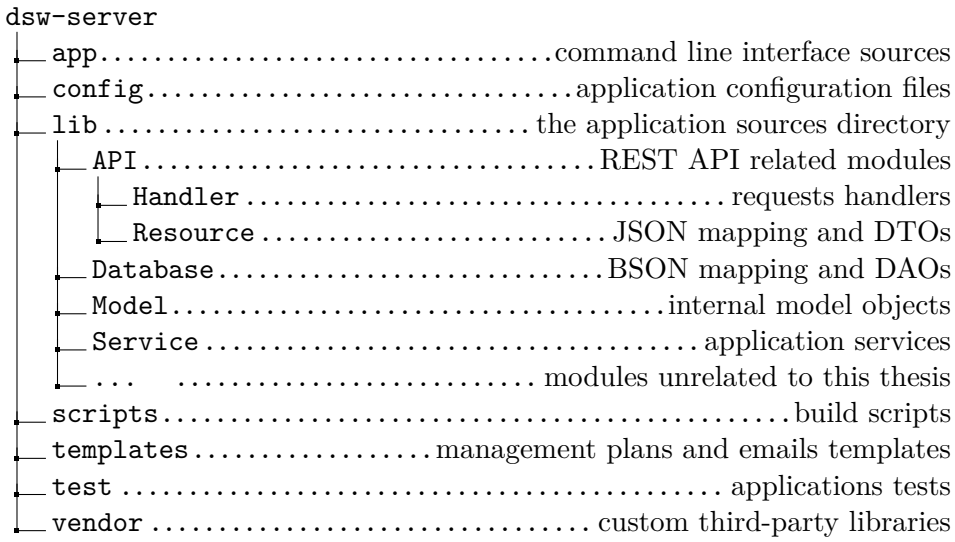

Figure 4.1: Structure of the server application

4.2.2 Client application structure

In this section, I will introduce the client application structure. Similarly to the server application structure, individual modules are described in detail in chapter 2. The structure in figure 4.2 lists main application modules. Modules, which are not related to this thesis are left out and grouped using an ellipsis (...) sign.

The client application project folder contains other sources then Elm source files. These are not relevant to this thesis and therefore the structure only shows sources located in the `src/elm` directory.

```
/src/elm
├── KMEditor ..... knowledge models editor module
├── KnowledgeModels ..... knowledge models preview module
├── Questionnaires ..... questionnaire management module
│   ├── Create ..... module for creating a new questionnaire
│   ├── Edit ..... module for editing an existing questionnaire
│   ├── Migration ..... questionnaire migration module
│   │   ├── DiffOverview ..... questionnaire diff view
│   │   └── DiffTree ..... questionnaire structure view
│   └── ... ..... modules unrelated to this thesis
└── ... ..... modules unrelated to this thesis
```

Figure 4.2: Structure of the client application

4.3 Implementation

In this section, I describe in detail how features designed in chapter 3 are implemented. Sections are appropriately divided into parts according to the application where the feature was implemented.

4.3.1 Initialize questionnaire migration

The initialization part of the migration process is handled on both client and server side. In the following sections, I will describe each part individually.

Server-side implementation

On the server side, the initialization is a one-step process. The server exposes a REST API route accessible on:

`POST /questionnaires/:qtnUuid/migrations` .

The route is registered in shared application router using its URL, HTTP method and handling function. The registration is shown in code example 4.2.

The initialization itself is simple, as it only requires to validate input data and store the migration state. The state is an uncomplicated data structure which is composed of the migrated questionnaire (still based on the older version of the knowledge model) and an identifier of the knowledge model.

Once the request pass validations, its data are passed to the `QuestionnaireMigrationService` module. This module is responsible for migration consistency – it always transitions from one consistent state to another.

After the questionnaire is created, the stored data are enriched by compiled knowledge models and the difference events and returned to the client.

By creating a separate structure for keeping the migration state, it is secured that the original version will never be modified during migration. This

```
createEndpoints :: BaseContext -> ScottyT Text BaseContextM ()
createEndpoints context = do
  -- Middleware registration
  middleware {- logging, authorization, CORS -}
  -- Routes registration
  get "/questionnaires/:qtnUuid/migrations"
        getQuestionnaireMigrationsCurrentA
  post "/questionnaires/:qtnUuid/migrations"
        postQuestionnaireMigrationsCurrentA
  delete "/questionnaires/:qtnUuid/migrations"
        deleteQuestionnaireMigrationsCurrentA
  put "/questionnaires/:qtnUuid/migrations"
        putQuestionnaireMigrationsCurrentA
  put "/questionnaires/:qtnUuid/migrations/resolveQuestionEvent"
        putQuestionnaireMigrationsQuestionFlagA
```

Listing 4.2: Registration of the REST route

approach is more sophisticated than doing the state modification in-place (by modifying the original questionnaire directly) but significantly helps to keep system consistency.

Client-side implementation

On the client side, this process requires two steps. In the first step, the application needs to fetch information about available knowledge models which user can migrate to. This action is similar to upgrading knowledge models itself, and the whole code for such functionality was reused from the existing code base (programmed by Ing. Jan Slifka[4]).

Once the data are fetched, the application will filter out invalid data (older versions of the knowledge model) and display a modal window with available options (shown in wireframe 3.1). Once the user selects a target version for the migration, the application sends data to the server. After the server responds, the application will load the migration detail and display it to the user.

The code used for the transition between application states is shown in example 4.3.

4.3.2 Cancel questionnaire migration

Now I would like to acquaint the reader with the implementation of the migration canceling feature.

```
-- Creates action for loading available knowledge models
tableActionUpgrade wrapMsg =
    wrapMsg << ShowHideQuestionnaireUpgradeForm << Just

-- Updates state based on received message
update msg wrapMsg appState model =
    case msg of
        ShowHideQuestionnaireUpgradeForm questionnaire ->
            loadAvailableKnowledgeModels questionnaire

        PostQuestionnaireMigrationCompleted (Ok _) ->
            let
                uuid =
                    model.migratedQuestionnaireUuid
            in
            redirectToQuestionnareMigrationDetail uuid

        PostQuestionnaireMigrationCompleted (Err error) ->
                processMigrationError
```

Listing 4.3: Transitioning between states during migration initialization

Server-side implementation

On the server side, the cancellation is made by deleting the existing questionnaire state. By deleting the initialized state, the original questionnaire will appear in the same state as it was before the migration was initialized.

The endpoint is available at route:

`DELETE /questionnaires/:qtnUuid/migrations` .

Once the user sends the request to the server, it is validated by the handler layer. After validation, the data are processed using service which takes care of converting request data into the internal representation. The deletion itself is done using `DAO` as shown in example 4.4.

The result of the database action is converted into an HTTP status code and returned to the client application with an empty body.

Client-side implementation

The client application offers the user to cancel migration only in case, where the server sent questionnaire data with appropriate flag (as shown in wireframe 3.5). When the user chooses to cancel the migration, the application sends

```
deleteMigratorStateByQuestionnaireId :: String
                                     -> AppContextM ()
deleteMigratorStateByQuestionnaireId qtnUuid = do
  let action = delete $
    select ["questionnaire.uuid" =: qtnUuid] qtnmCollection
  runDB action
```

Listing 4.4: DAO module handling migration state deletion

a request to the server. Once the server responds to the request, a list of questionnaires is reloaded to correspond with the latest state available at the server.

4.3.3 Create migration context

The migration context is a feature, where the user can see changes occurred in between current and target versions of the knowledge model. The context is made from knowledge model events, which are visualized in the context of the filled questionnaire. Therefore, the migration context is only done on the client side of the application – the only purpose of the server for this action is to provide the current migration state.

Server-side implementation

For the migration state, the server needs to create a *diff knowledge model*. Diff knowledge model contains nodes from the newer version together with nodes which were only available in the previous version and were deleted during customization.

Such a knowledge model is created using a specific module called `KnowledgeModelDiffService`. This service module exposes functionality to create diff knowledge model by giving original and target knowledge models identifiers.

The service will compile the original knowledge model and finds all events occurred up until the target version. After that, a specialized version of the knowledge model compiler is called to create the diff knowledge model. The custom compiler main task is to remove all destructing events and to run the standard compiler as shown in example 4.5.

Once the knowledge is compiled, it enriches the response to the client application.

```
1 -- Creates diff knowledge model from base km by applying events
2 runDiffApplicator km events =
3     runApplicator km editedEvents
4     where editedEvents = filter isNotDeleteEvent events
```

Listing 4.5: Diff knowledge model compiler

Client-side implementation

On the client side, making the context is more complex. There are three structures which need to be created and maintained; that is:

- `DiffTree`,
- `DiffStates`,
- `DiffEventsUuids`.

As mentioned in section 3.5, the first two structures are incompatible, and therefore the third structure is used to map between them.

The `DiffTree` is used to create an overview of the whole questionnaire structure (with all possible questions and answers) by also taking the user's answers into account.

The structure is built on top of the custom created knowledge model, called diff knowledge model.

Once the client application downloads the migration state, it starts building the mentioned structures. The first one created is `DiffTree`. The tree is created by recursively browsing knowledge model nodes as shown in example 4.6.

Once the tree is created, the diff events are transformed into the internal representation to build the `DiffStates` structure. Because the `DiffState` is used to show both original and new state, the transformation needs not only the event as an input, but also both versions of the knowledge model. While transforming events, there is no information about the hierarchical classification of the node. Therefore this structure cannot be directly mapped to the tree structure and can not take into account the user's replies. The example of transforming events into `DiffState`s is shown in example 4.7.

While creating the state's structure, there might be multiple events modifying the same node (for example `Added` and `Modified` or `Modified` and `Removed`). Therefore, each event has assigned a significance:

```
{- Create the tree root node -}
createDiffTree knowledgeModel replies =
    let
        chaptersSubtree = {- shortened -}
        kmNode = {- shortened -}
    in
    {- Create root node with chapters subtree -}
    kmNode chaptersSubtree

{- Create subtree for chapters -}
createChapterDiffSubTree parentPath replies chapter diffTree =
    let
        children =
            List.map getQuestionUuid chapter.questions

        path =
            List.append parentPath
                [ ChapterPathNode chapter.uuid ]

        subtreeDict =
            List.foldl (createQuestionDiffSubTree path replies)
                diffTree chapter.questions

        chapterNode = {- shortened -}
    in
    Dict.insert chapter.uuid chapterNode subtreeDict
```

Listing 4.6: Recursive initialization of the questionnaire tree (simplified)

1. `Removed`,
2. `Added`,
3. `Modified`.

Whenever an event for a node occurs, it is only processed when there was no event for the same node processed previously or when the event has a higher priority. This makes sure the user sees events classified as they happened from his perspective.

The last structure used to create a questionnaire overview is `DiffEventsUuids`. This structure is built after the previous one because its content is used to quickly search diff state nodes for the overview tree and vice versa. It

```haskell
{- Transforms km modification event into diff state -}
convertDiffEventToDiffState oldKm diffKm event dict =
    case event of
        AddKnowledgeModelEvent data _ ->
            kmCreatedState diffKm dict

        EditKnowledgeModelEvent data _ ->
            kmModifiedState data.kmUuid oldKm diffKm dict

        AddChapterEvent data _ ->
            cptrCreatedState data.chapterUuid diffKm dict

        EditChapterEvent data _ ->
            cptrModifiedState data.chapterUuid oldKm diffKm dict

        {- ... Continues for each km node-}
```

Listing 4.7: Transforming knowledge model events into diff state

is created by visiting every node in the `DiffTree` and looking up its unique identifier in the `DiffState`, when the node is found in of both these structures, it is inserted into the collection. This way, there will be multiple records for item templates question type if the user replied multiple time. Such behavior is intentional and correct by design. Illustration 4.3 shows how the mentioned data structures are connected.

4.3.4 Update migrated questionnaire state

Updating questionnaire state allows the user to add flags to questions which are listed in the list of diff events.

Server-side implementation

On the server side, the only responsibility is to persist given flags. As there might be more flags in the future with different entitlements for consistency, the server does not run any validations on flags provided from the client application. Currently, the whole consistency is managed on the client application only.

Once the provided data from the client are successfully deserialized, it replaces the current flags for a given question and is stored in the questionnaire migration state.

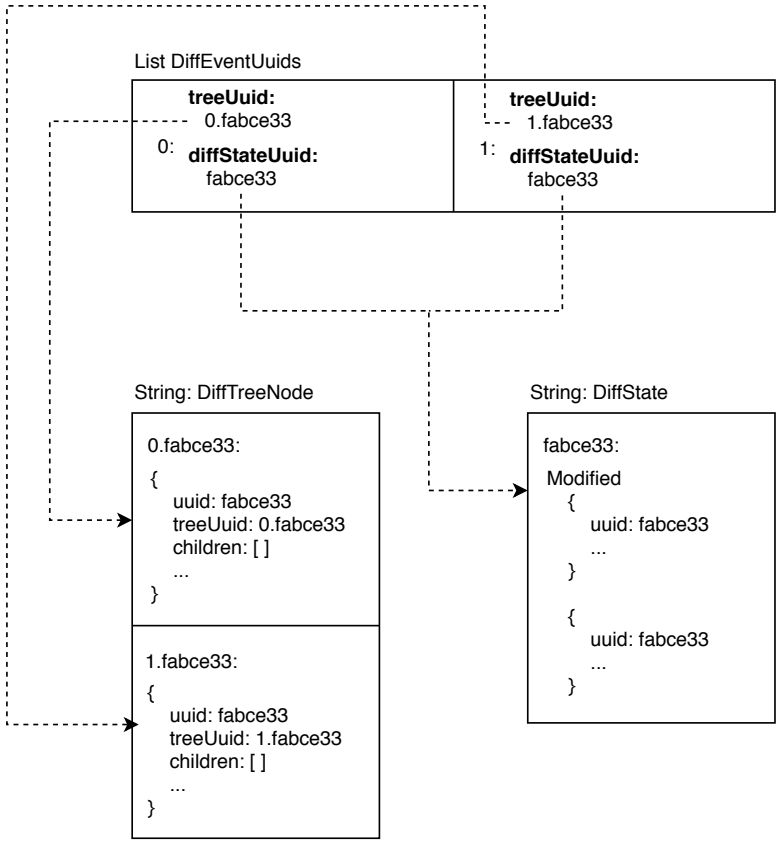

Figure 4.3: A mapping between tree structure and diff events

Client-side implementation

The client application allows modifying question state only when the question is listed in the list of events and has a default state (no flags). To do so, the application only shows the *resolved* and *needs review* buttons when the question state was not modified previously. In another case, the application renders an *undo* button instead to allow the user to return the question to the default state.

The state is synchronized immediately when the user presses the appropriate button. Once the state change is synchronized with the server, it is also updated locally (without fetching data from the server). The local update forces application to re-render the whole screen (in the sense of virtual DOM) which leads to switching action buttons for the undo action and vice versa.

The application also allows the user to add flags to system answers (for item templates or single choice answers). This, however, internally adds the flag to the question instead of the answer because the questionnaire UI does not support previews of answers only.

4.3.5 Finalize migration

Finalizing the migration is the last step to upgrade the questionnaire knowledge model version.

Server-side implementation

Once the client requests to finalize the migration, the server needs to do the following tasks:

1. create a new questionnaire with migrated answers,
2. delete the migration state.

The first step ensures that the new questionnaire is created alongside the original one. Together with the questionnaire, replies and question flags are copied too.

The second step reverts the original questionnaire into its original state (so it can be migrated again). Once the migration is finished, the user's list of questionnaires will contain the new questionnaire alongside the original.

The API endpoint is available at:

```
PUT /questionnaires/:qtnUuid/migrations
```

and returns empty body with appropriate HTTP status code.

Client-side implementation

The client application uses a similar approach as with initializing or finalizing the migration. The user initiates system action (`Message`, in Elm terminology) which is followed by creating an API request to the server.

The action is initiated from the migration detail. Once the server responds, the application redirects the user to the list of questionnaires. After the redirect, the application will fetch the latest version of the questionnaires and user will the newer version display alongside the original one.

4.4 Design

In this section, I will acquaint the reader with the final application design. The application design was chosen in such a way, so it is consistent with the rest of the application.

The existing UI components such as modal windows, buttons and color shades were reused, so the user will feel familiar while using the new migration tool. The final design is shown in figures 4.4 to 4.8.

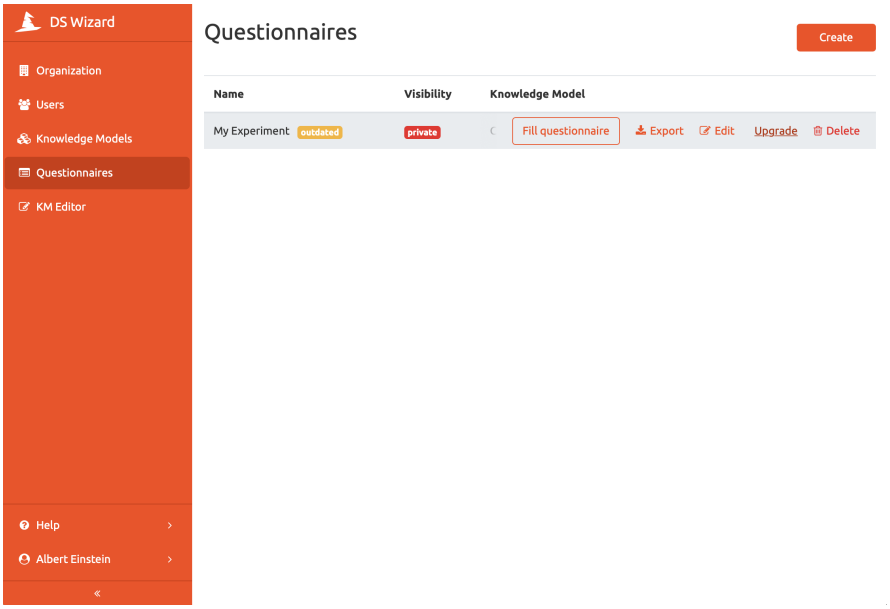

Figure 4.4: The upgrade action is only visible when there is available migration

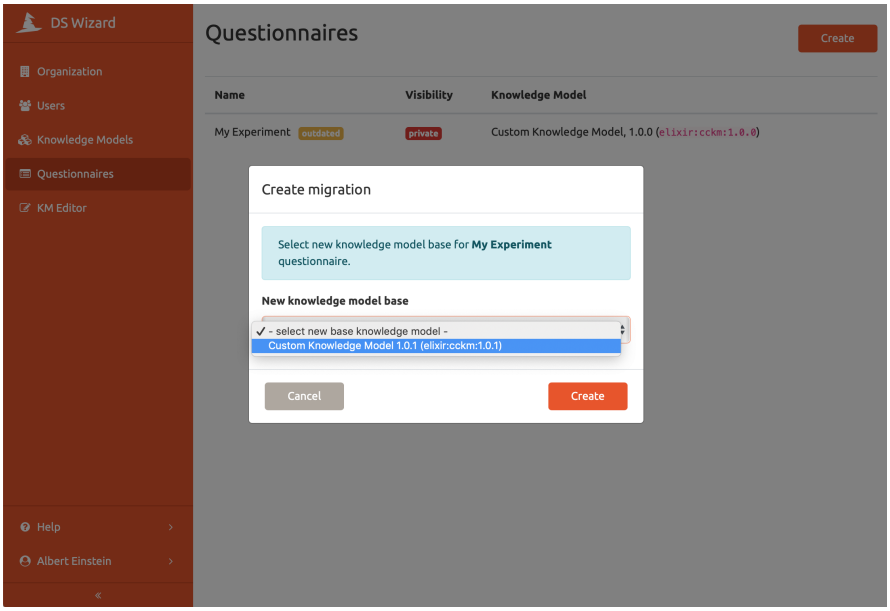

Figure 4.5: Modal with target version selection

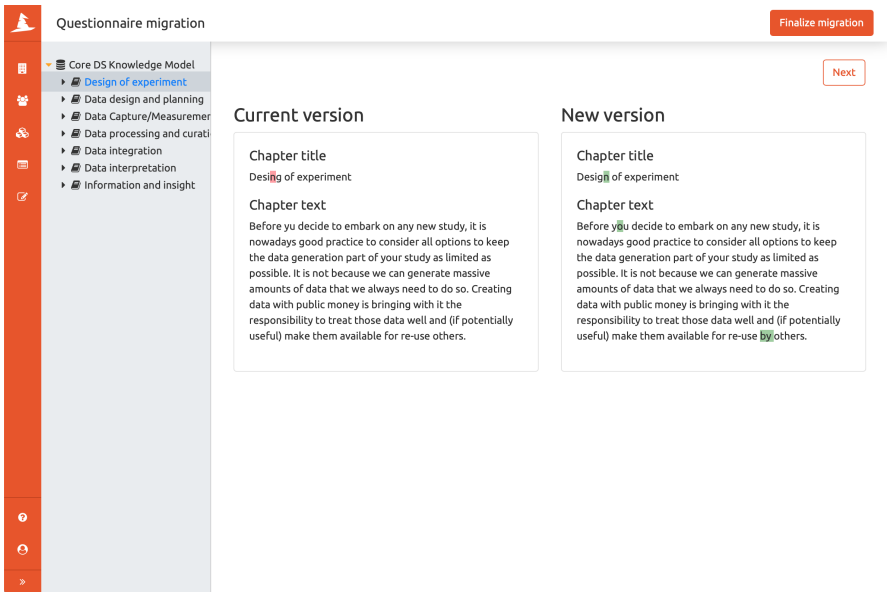

Figure 4.6: Migrated chapter text difference

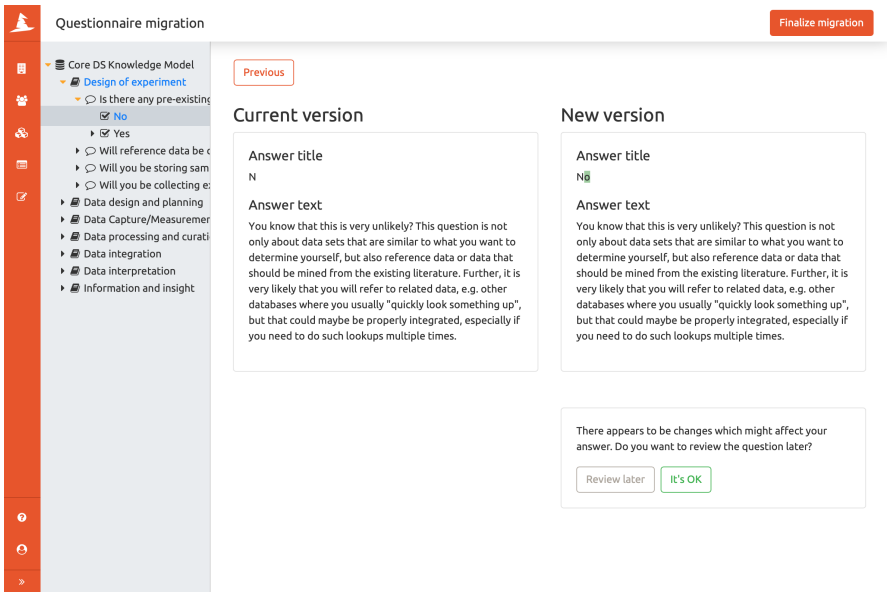

Figure 4.7: Migrated question overview with the ability to add a flag

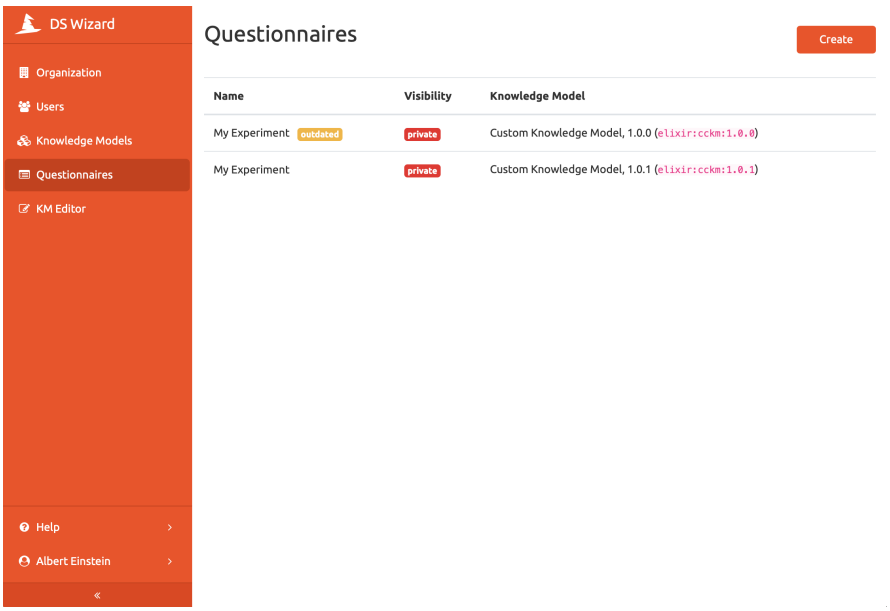

Figure 4.8: Migrated questionnaire alongside its original version

Chapter 5

Testing

In this chapter, I would like to mention the process of validation of the application correctness. For users, the key feature of the migrator tool is the migration context overview.

I will briefly mention main cases which may occur. Then, I will demonstrate a migration use-case, where mentioned cases will be used.

5.1 Testing cases

In this section, I will describe all possible cases which may occur during the migration process one by one.

5.1.1 Added questionnaire node

In each migration process, there might be a new node added to the questionnaire. Such node needs to be always contained in the list of the diff event identifiers, so the user can use application navigation to reach the node. Moreover, the node needs to be listed more than once if it is part of the reply to the item template question type.

The associated event will only reference the new value as there is no original value to be referenced.

5.1.2 Modified questionnaire node

By definition, the modified node also needs to be listed in the list of the diff event identifiers with the same condition applying to the node which is part of an item template reply.

Because the modification event is used to show both the original and a new version, it needs to reference two questionnaire nodes. One node is referenced from the original questionnaire knowledge model, the other one from diff knowledge model.

5.1.3 Removed questionnaire node

When an arbitrary node was removed from the knowledge model, it also needs to be listed in the list of diff event identifiers. Similarly to the addition event, this event only reference one node too. However, this node is referenced from the original knowledge model to reflect the version known to the user.

5.1.4 Unchanged questionnaire node

In oppose to previous events, unchanged nodes must not be contained in the list of diff event identifiers. It will only be available in the `DiffStates` structure listed as `Unchanged`.

This approach will allow the node to be displayed in the overview without any additional difference information (such as text highlighting). Because the old and new nodes must be the same (by definition), the event will only reference the original one, but it will be displayed on both the old questionnaire and new questionnaire panels.

5.1.5 Preserved migrated question type

In the case of the question node, there might occur the type change customization event in addition to textual changes. This event always needs to be marked as `Modified` because the question type can only be modified on the new node if there was aa existing node before (described in 4.3.3).

If the question type was not changed, the question could be treated as it was not changed at all (an `Unchanged` event) and its reply or replies must be preserved.

5.1.6 Changed migrated question type

In the case the question type was changed from either of types `Item templates`, `Value` or `Options` to another type, there is no possibility to (automatically) preserve the reply, and therefore it will not be available after the migration is done.

5.2 Testing use case

In this section, I will demonstrate a non-trivial use case showing how the system should behave. Firstly, I will specify an input state of the testing case. The, I will acquaint the reader with the expected result for such input.

5.2.1 Test input state

For the input, I will assume a list of events, which are needed to be presented to the user during the migration process. The given list will be visualized

in a tree structure representing the original knowledge model on which the events were applied (in oppose to questionnaire structure, it ignores the user's replies). In this tree, I will use following notation for each node:

```
[STATE] (ID) Title <Type>
```

, where:

- `[STATE]` stands for the difference state (either `D` for deleted, `U` for unchanged, `A` for added or `M` for modified),
- `(ID)` stands for node's unique identifier,
- `Title` stands for node's display name,
- `<Type>` only applies to a question node and represent its type.

To successfully demonstrate the correctness of the migrator solution, the events must cover non-trivial customizations. Therefore, all the possible difference states are taken into account. The upgrade contains the following events:

1. `EditKnowledgeModelEvent` with identifier `KM`,
2. `EditQuestionEvent` with identifier `Q2`,
3. `AddAnswerEvent` with identifier `A2`,
4. `DeleteChapterEvent` with identifier `C3`.

There are two important events. One is adding an answer to the template item question type, and the other is deleting a chapter.

By adding an answer to an item templates question type, the application needs to take into account user's replies and create the change overview structures accordingly. To thoroughly test this behavior, the input state assumes that there are precisely two user's reply (filled item template) to this question.

The chapter deletion event demonstrates how the application should behave while deleting node composed from other nodes – in this case, the chapter is composed of nested questions.

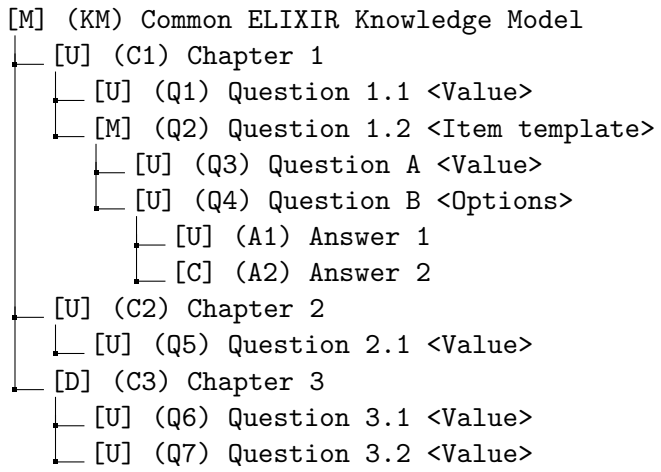

Figure 5.1: Visualization of knowledge model customization events

Figure 5.1 shows how events in the context of existing knowledge model for better understanding.

5.2.2 Expected output

The expected output contains five changes for the list of four customization events. The summary of mapping customization events to changes visible to the user follows.

EditKnowledgeModelEvent

This event should be presented as a single change. Because it is a modification event, the application presents a knowledge model title and description in the original and new versions.

The change is previewed in both panels, each displaying an appropriate version of the knowledge model.

EditQuestionEvent

In this case, editing question should be shown as a single change. The change should inform the user that there are two replies.

The change is previewed in both left and right panels, each displaying the appropriate version of the question.

AddAnswerEvent

Adding an answer to a question included in the item template should be shown as change as many times as the user replied to the question. In this case, the

user replied by filling two item templates and therefore the added question should be shown twice – each time in the context of one of the answers.

The change is displayed only in the right panel.

DeleteChapterEvent

To ensure simplicity of the migration tool, the user should not be bothered by information about the deletion of its nested nodes. Because of that, the chapter deletion should be shown as a single change, and nested questions should not be shown.

The change is displayed only in the left panel.

Conclusion

The goal of the thesis was to design and implement the migration tool for the data management plans created in the Data Stewardship Wizard.

The thesis starts with the research of the state-of-the-art of the DSW application. After the research, the reader is acquainted with the analysis of the solution along with its requirements. Based on defined requirements, individual use cases and their scenarios are discussed.

In the design chapter, the application design together with its architecture and primary classes are discussed.

Then, the development process, used technologies, and final application are introduced.

In the last chapter, the testing case is demonstrated describing how the application functionality was validated.

The outputs are two application modules, which may be integrated into the existing system. The first module is a server application written in the Haskell programming language. The second module is a web application frontend for the first module, written in the Elm language.

Working with such technologies was a huge personal challenge because not only I did not write a production-ready application in those technologies before, I did not even write a frontend or backend applications up until now. Another unknown field for me was Data Stewardship itself. I had to acquaint myself with two books [29, 30] to fully understand the domain of the problem.

Project future

Shortly, both modules should become part of the production application available to the general public. The system itself is in active development; this means all major changes should be reflected in those modules too. Therefore, the output will probably stay in active development. In my opinion, it is the actual usability of the implemented application what makes this thesis exceptional.

Bibliography

[1] Rada pro výzkum, vývoj a inovace. RVVI schválila návrh rozpočtu na příští tři roky [online]. April 2018, [Cited 2019-01-17]. Available from: https://www.vyzkum.cz/FrontAktualita.aspx?aktualita=837471

[2] European commission. EU budget: Commission proposes most ambitious Research and Innovation programme yet [online]. June 2018, [Cited 2019-01-17]. Available from: http://europa.eu/rapid/press-release_IP-18-4041_en.htm

[3] ELIXIR Data Stewardship Knowledge Model. [Cited 2019-05-03]. Available from: https://github.com/ds-wizard/ds-km

[4] Slifka, J. *Data Stewardship Portal: Client-side Web Frontend*. Master's thesis, Czech Technical University, Faculty of Information Technologies, 5 2018.

[5] Evan Czaplicki. Blazing Fast HTML: Round Two [online]. August 2016, [Cited 2019-02-11]. Available from: https://elm-lang.org/blog/blazing-fast-html-round-two

[6] Plain old Java object. December 2018, [Cited 2019-04-10]. Available from: https://en.wikipedia.org/wiki/Plain_old_Java_object

[7] Narayan Prusty. How Does HTTP Authentication Work. May 2014, [Cited 2019-04-10]. Available from: http://qnimate.com/understanding-http-authentication-in-depth

[8] Introduction to JSON Web Tokens. [Cited 2019-05-03]. Available from: https://jwt.io/introduction/

[9] Sulaiman, A. File System vs. Database. April 2017, [Cited 2019-05-03]. Available from: https://dzone.com/articles/which-is-better-saving-files-in-database-or-in-fil

[10] What Is a Document Database? [Cited 2019-05-03]. Available from: https://aws.amazon.com/nosql/document/

[11] What is MongoDB? [Cited 2019-05-03]. Available from: https://intellipaat.com/blog/what-is-mongodb/

[12] Minnick, C. The Real Benefits of the Virtual DOM in React.js. April 2016, [Cited 2019-04-15]. Available from: https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/

[13] Knaisl, V. *Migration Tool for Data Stewardship Knowledge Model*. Master's thesis, Czech Technical University, Faculty of Information Technologies, 5 2018.

[14] Rodriguez, T. S. Understanding Event-Driven Architectures. September 2018, [Cited 2019-04-13]. Available from: https://medium.com/drill/understanding-event-driven-architectures-eda-the-paradigm-of-the-future-7ae632f056bb

[15] la Torre, C. D. What is Docker? August 2018, [Cited 2019-04-16]. Available from: https://docs.microsoft.com/cs-cz/dotnet/standard/microservices-architecture/container-docker-introduction/docker-defined

[16] Wikipedia. Domain model. April 2019, [Cited 2019-04-19]. Available from: https://en.wikipedia.org/wiki/Domain_model

[17] Winston, A. Importance Of UI/UX Design In The Development Of Mobile Apps. July 2019, [Cited 2019-04-20]. Available from: https://www.dotcominfoway.com/blog/importance-of-ui-ux-design-in-mobile-app-development

[18] DSW – Release 1.5. [Cited 2019-04-22]. Available from: https://github.com/ds-wizard/dsw-server/pull/59

[19] Haskell Programming Language. [Cited 2019-04-24]. Available from: https://www.haskell.org

[20] Richardson, C. What are microservices? [Cited 2019-04-24]. Available from: https://microservices.io

[21] Rouse, M. Integrated Development Environment. [Cited 2019-04-25]. Available from: https://searchsoftwarequality.techtarget.com/definition/integrated-development-environment

[22] Elm: Functional Frontend. [Cited 2019-04-25]. Available from: https://mmhaskell.com/blog/2018/11/12/elm-more-functional-frontend

[23] elm - Visual Studio Marketplace. [Cited 2019-04-25]. Available from: https://marketplace.visualstudio.com/items?itemName=sbrink.elm

[24] January 2018 Web Server Survey. [Cited 2019-04-25]. Available from: https://news.netcraft.com/archives/2018/01/19/january-2018-web-server-survey.html

[25] String metric. October 2018, [Cited 2019-04-27]. Available from: https://en.wikipedia.org/wiki/String_metric

[26] MYERS, E. W. An O(ND) Difference Algorithm and Its Variations. November 1986, [Cited 2019-04-27]. Available from: http://www.xmailserver.org/diff2.pdf

[27] Sun Wu, G. M., Udi Manber. An O(NP) Sequence Comparison Algorithm. August 1989, [Cited 2019-04-27]. Available from: http://myerslab.mpi-cbg.de/wp-content/uploads/2014/06/np_diff.pdf

[28] elm-diff. [Cited 2019-04-27]. Available from: https://package.elm-lang.org/packages/jinjor/elm-diff/latest/

[29] Plotkin, D. *Data Stewardship*. Morgan Kaufmann, 2013.

[30] Mons, B. *Data Stewardship for Open Science*. CRC Press, 2018.

Appendix A

Acronyms

API Application Programming Interface. 1, 3, 5, 11, 44, 52, 54, 57, 65

BSON Binary JSON. 1, 11

CRUD Create, Read, Update, Delete. 1, 7, 10, 11, 17

CSS Cascading Style Sheets. 1

DAO Data Access Object. 1, 5, 7, 10

DMP Data Management Plans. 1, 2

DOM Document Object Model. 1, 12, 64

DSL Domain Specific Language. 1, 11

DSW Data Stewardship Wizard. 1–4, 7–10, 12, 13, 15–19, 21, 28, 40, 51, 52, 54, 75

DTO Data Transfer Object. 1, 5–7, 18

FAIR Findable, Accessible, Interoperable, Reusable. 1, 2, 13

HTML HyperText Markup Language. 1, 54

HTTP HyperText Transfer Protocol. 1, 6, 7, 19, 52, 54, 57, 59, 65

IDE Integrated Development Environment. 1, 52, 53

JSON JavaScript Object Notation. 1, 4, 8, 11, 15, 16, 52

JWT Javascript Web Token. 1, 6–8, 16

KM Knowledge Model. 1, 2

POJO Plain Old Java Object. 1, 7

REST Representational State Transfer. 1, 5, 44, 52, 57, 58

SASS Syntactically Awesome Style Sheets. 1

SQL Structured Query Language. 1, 11

UI User Interface. 1, 3, 12, 14, 33, 64, 65

URL Uniform Resource Locator. 1, 6, 12, 44, 57

Appendix B

Development Guide

This guide contains instructions for running both the server and client side applications in the development environment.

B.1 Client-side Application

The application's source code is available at GitHub in the following repository:

`git@github.com:ds-wizard/dsw-client.git`

For building the application, the `Node.js` environment together with the `yarn` package manager is needed. The installation guide for the package manager is available at:

`https://yarnpkg.com/en/docs/install`

In the terminal, clone the repository change your working directory to the project root:

```
$ git clone git@github.com:ds-wizard/dsw-client.git
$ cd dsw-client
```

Install the project dependencies:

```
$ yarn install
```

Run the application:

```
$ yarn start
```

The application is now running at `http://localhost:8080`. It expects the backend to be available at `http://localhost:3000`.

B.2 Server-side application

For the server-side application, the Stack tool is required. Installation guide for the Stack is available at:

`https://docs.haskellstack.org/en/stable/install_and_upgrade/`

You will also need a MongoDB database and a RabbitMQ messaging server. For simplicity, there is a `Docker compose` configuration publicly available at GitHub:

`https://github.com/josefdolezal/dockerfiles/tree/master/dsw-local`

Once your development environment is ready, clone the application repository and change your working directory to the project root:

```
$ git clone git@github.com:ds-wizard/dsw-server.git
$ cd dsw-server
```

Create your application configuration file and fill in your settings:

```
$ cp config/app-config.example.cfg config/app-config.cfg
$ vim config/app-config.cfg
```

Build the application from sources:

```
$ stack build
```

Run the server in a development configuration:

```
$ stack run dsw-server
```

The application is now running at `http://localhost:3000`.

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `exe` ..... the directory with executables
  - `dsw-client` ..... directory with the compiled Elm sources
  - `dsw-server` ..... directory with the the Haskell executable
- `src` ..... the directory of source codes
  - `dsw-client` ..... client-side implementation sources
  - `dsw-server` ..... server-side implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format