CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER’S THESIS

Title:	Migration Tool for Data Stewardship Knowledge Model
Student:	Bc. Vojtěch Knaisl
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 2017/18

Instructions

- Acquaint yourself with the Data Stewardship Planning Portal project and its Knowledge Model (DS-KM).
- Acquaint yourself with the Haskell programming language.
- Analyse the requirements for the Migration Tool for DS-KM (MT).
- Design a solution of the MT.
- Implement the solution in Haskell.
- Test and document your solution.

References

https://github.com/DataStewardshipPortal

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague December 23, 2016

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Migration Tool for Data Stewardship Knowledge Model

Bc. Vojtěch Knaisl

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

January 8, 2018

Acknowledgements

I would like to thank my supervisor Ing. Robert Pergl, Ph.D. for great feedback during my work on the thesis and for leading my thesis. Further I would like to thank my family for a priceless support during my studies.

Declaration

In Prague on January 8, 2018 ....

Citation of this thesis

Knaisl, Vojtěch. *Migration Tool for Data Stewardship Knowledge Model.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce navazuje na již existují projekt portálu pro správu dat, který je vyvíjen ve spolupráci s organizací ELIXIR-CZ. Práce stručně shrnuje, co bylo na projektu již vytvořeno. Dále popisuje, proč byla nutnost vytvořit migrační nástroj pro aktuální znalostní model (rozšiřitelná datová struktura, která je nositelem znalostí a vyvíjí se spolu s časem) a proč bylo nakonec rozhodnuto, že spolu s tímto nástrojem bude realizován i základ portálové aplikace. Migrační nástroj a portál je následně zanalyzován. Součástí práce je i návrh migračního nástroje s možnostmi variant jeho realizace. Tyto možnosti jsou diskutovány a jedna z variant je posléze implementována. Výsledkem práce je tedy požadovaný plně funkční migrační nástroj a základ portálové aplikace.

Klíčová slova Migrační nástroj, ELIXIR, Správa dat, Haskell, Funkcionální programování, Git

Abstract

This thesis continues on an already running project of Data Stewardship Planning Portal which is developed together with ELIXIR-CZ organization.

The thesis shortly summarizes a current state of the project. Further it describes why there was a need to create a migration tool for the current knowledge model (an extensible data structure carrying the knowledge that naturally evolves over the time), and why it was decided to implement also bases of the portal application. The migration tool and portal application are then analyzed. A part of the thesis is also a design of the migration tool together with possible approaches of its implementation. These approaches are discussed and finally one is choosen and implemented. The result of this thesis is the requested fully-working migration tool together with bases of the portal application.

Keywords Migration Tool, ELIXIR, Data Stewardship, Haskell, Functional Programming, Git

Contents

List of Figures

List of Tables

Introduction

ELIXIR[1] is an organization which coordinates activities around a life science across Europe. It consists of 21 nodes which represent countries. A purpose of ELIXIR is to help researchers to share and exchange their data and expertises.

ELIXIR CZ[2] is one of 21 nodes which cover the activities in the Czech Republic. Its structure is divided into 4 areas called `platforms` – Data, Tools, Interoperability, Compute and Training. This thesis belongs to an Interoperability platform, whose aim is to help in discovery, integration and analysis of biological data.

I was temporarily a part of the team of Interoperability platform in ELIXIR CZ which engages mainly in a data stewardship and a data management, in foundational ontologies and supports F.A.I.R.[3] data principle in ELIXIR community.

Goals of the Thesis

The main goal of this thesis is to create a migration tool for a knowledge model from Data Stewardship Planning Portal. A purpose of the migration tool should be to offer an option how to update existed compiled knowledge models with new information coming from core knowledge model or its localizations.

First it is needed to acquaint myself with the Data Stewardship Planning Portal project and its Knowledge Model. Project will be written in the Haskell[4] programming language so it is necessary to acquaint also with Haskell. In initial part, it should be analysed requirements for the migration tool. Based on that, the analysis should be conducted.The main part of the thesis is then dedicated to a design of the migration tool and its implementation. And finally, the solution should be properly tested and documented.

Methodology and Thesis Structure

Thesis is structured according to a standard software development process – it consists of an analysis, a design, an implementation, a testing and a deployment.

In the first chapter (1) which belongs to analysis I recapitulate a state of the project (see section 1.2). Further I sum up solution requirements (see section 1.3). For diagrams I use a notation of UML in version 2.x[5]. Then I talk about tools (a programming language, a framework, etc.) which I selected based on the requirements (see section 1.4). And the last section in this chapter is dedicated to a few concepts of a functional programming which I used frequently in an implementation (see section 1.5).

In the second chapter (2) I discuss a design of the migration tool. I mention an idea how a solution was intended to look like (see section 2.2) and describe an example of migration from the real world which inspires me – Git (see section 2.3). Further I write about my proposal (see section 2.4). I explain my suggested approaches and discuss what led me to choose a final solution.

The third chapter (3) is focused on an implementation. It consists of a description of a project structure (see section 3.2) and a documentation how the application was implemented (see section 3.4). I mention here how I designed API (see section 3.5), how I worked with a database (see section 3.6) and how I handled errors (see section 3.7).

The last chapter (4) is dedicated to a configuration, a testing and a production deployment. I mention here how the application can be configured (see section 4.2), how I tested the application (see section 4.3), how the application should be rightly deployed (see section 4.4). I also describe here a deployment to a server provided by my faculty.

Chapter 1

Analysis

1.1 Introduction

In this chapter I would like to recapitulate a state of the project (1.2). I describe here what has been already done and what are the plans for the future. Further I specify solution requirements and do an analytic work with a goal of to sum up what will be a final product of this thesis (1.3).

Next I discuss here a list of tools which I used and describe them a little bit (1.4). So the reader can more understand the reasons why I chose them.

And finally as I used concepts and patterns which are not very common, I created a quick overview over them (1.5).

1.2 Current Project State

Project is currently at its beginning but we can already see some results.

1.2.1 Completed parts

1.2.1.1 Project website

The project has its own website where we can find some basic information about the project. And it is also possible to find there a quick overview what a data stewardship means, how a process of managing data works and what are roles which participate in it. Next, the website contains a managerial questionnaire whose goal is to collect an information about current processes of managing data in research institutions. The website was done by Robert Pergl from FIT CTU and it is written in Haskell[4].

1.2.1.2 Knowledge Model

The core of the project is a knowledge model which was previously stored in mind maps. For a better machine usage it had to be rewritten to a JSON

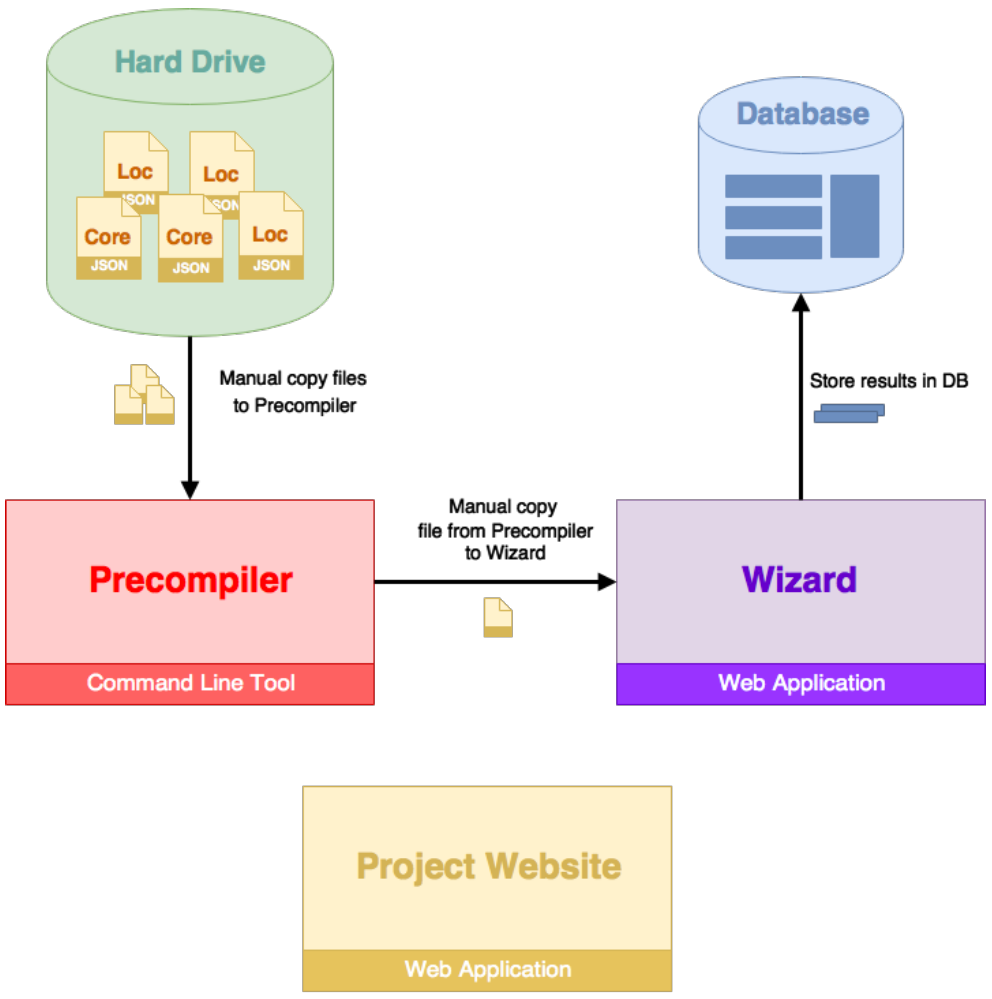

Figure 1.1: Current project state

format. This work was done by Rob Hooft from DTL Netherlands.

1.2.1.3 Precompiler

A base knowledge model is currently named as a core. It can be extended by localizations. A final knowledge model is then created by composing core and their localizations. The result is one JSON file with one knowledge model. The precompiler which composes these parts together was done by Marek Suchánek from FIT CTU and currently it is written in Python[6].

1.2.1.4 Wizard

The compiled knowledge model is used in Wizard application where researchers answer the questions from a knowledge model. These produced data is planned to be used for a generating of data management plans in the future. The application was done by Robert Pergl and currently it is written in Haskell.

1.2.2 Prepared applications

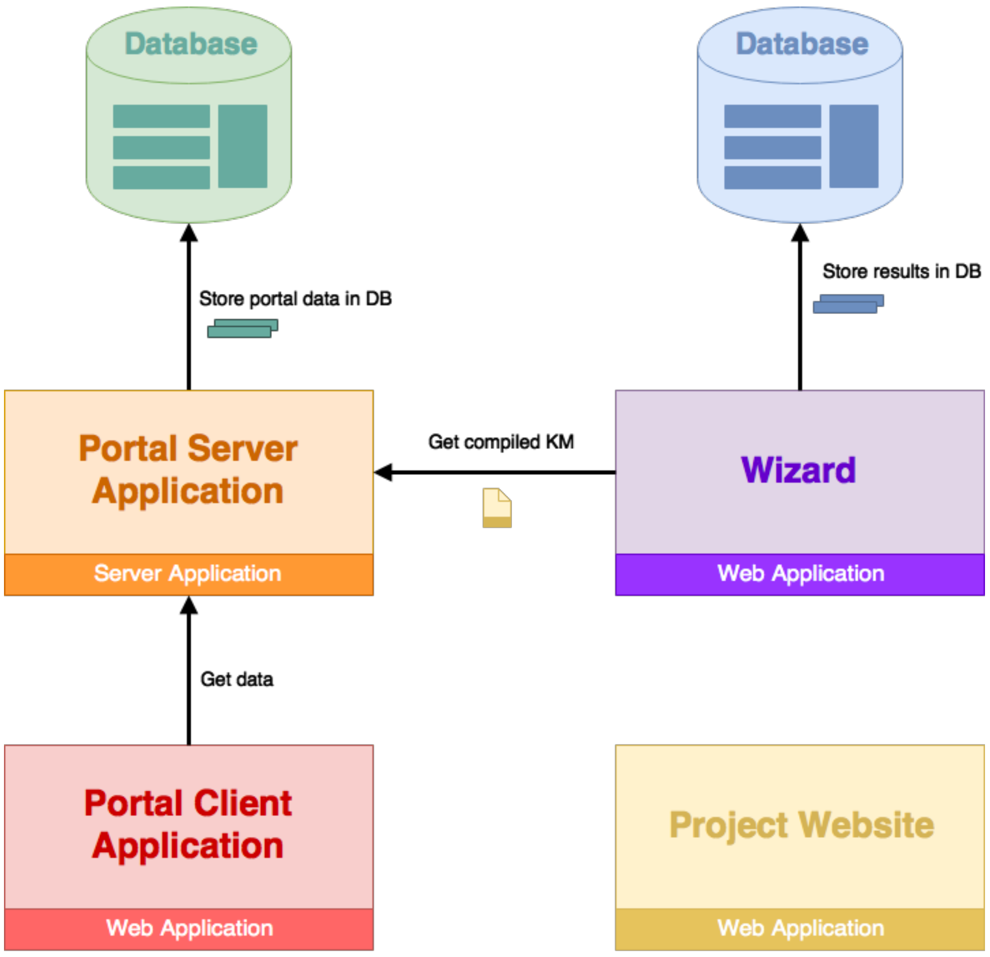

Figure 1.2: Desired project state

1.2.2.1 New portal application

As a next step it was decided to centralize UI for users. In the future there should be one central point (an application) for users. It should server to administrators, data stewards and researchers for managing all things related to a data management.

An implementation of this new portal application was divided into 6 steps:

1. Create a basic application setup (user management module, authentication and authorization module, etc.)
2. Create an administration for managing packages (a package is a reimbursement for the core and localization files)
3. Create an editor for knowledge models
4. Create a process for upgrading knowledge models
5. Convert currently already implemented wizard application to the portal
6. Create a generator for data management plans

First 4 steps are covered on a server side in this diploma thesis and on a client side by Jan Slifka from FIT CTU in his diploma thesis.

1.3 Solution Requirements

1.3.1 Introduction

In the previous section (1.2) I defined 4 parts which I would like to cover in this diploma thesis. The main and the most crucial was to implement a migration tool. But in the beginning this diploma thesis should be just about this migration tool. But there appeared problems how to handle a user interaction during migrations, how to handle conflicts which can appear there, etc. Still it could be done through command line but because the tool was intended to be used by data stewards I rejected this idea. Finally it was decided to create a portal and integrate the migration tool inside it. My work was to provided an API which was consumed by a new web client made by Jan Slifka. To have some presentable demo we also had to cover other modules like a management of users, a management of packages or an editor of knowledge models as I mentioned in the end of the previous section (1.2).

1.3.2 Functional and Nonfunctional Requirements

First I would like to analyze functional and nonfunctional requirements.

Functional requirements:

- F1: Support of an editing of base information about an organization
- F1: Support of a management of users in the system
- F2: Support of an editing of knowledge models
- F3: Support of an upgrading of knowledge models
- F4: Support of a management of packages

Nonfuctional requirements:

- NF1: Application is written in Haskell[4]
- NF2: Application provides a REST API[7]
- NF3: Application is deployed using Docker[8]
- NF4: Application is deployed on Linux server
- NF5: API is publicly accessible
- NF6: Multiple organizations in one deployed instance of an application are not allowed (one instance of an application can covers just needs of one organization)

1.3.3 Use Cases

For clarity I divided use cases to 4 groups – Organization, User Management, Editor together with Migration Tool and Package Management.

1.3.3.1 Roles

I identified 3 roles – `Administrator`, `Data Steward` and `Researcher`. In general `Administrator` is allowed to perform all actions. Its primary role is to set up the application and managed users. In case of some problem he can intervene and try to fix the problem. `Data Steward` should primary take care of knowledge models and packages. `Researcher` is here just for the future usage. `Researcher` will be the person who will fill the knowledge models and who will manage data management plans.

1.3.3.2 Group: Organization

This group is very simple. It belongs here just one use case for `Administrator` – `Edit organization` which means to edit information about organization.

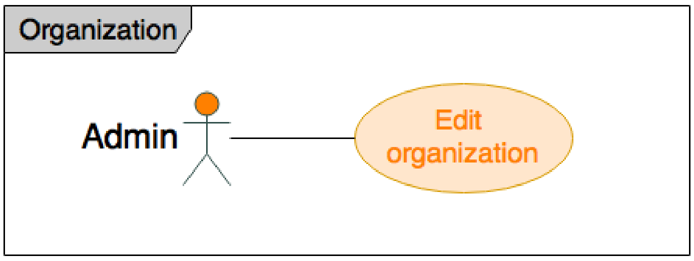

Figure 1.3: Use Cases: Organization

1.3.3.3 Group: User Management

User management is primary just for `Administrator`, too. One exception here is just `Edit profile` which can perform anyone and means to edit your own profile. Rest of use cases covers CRUD operations over a user entity.

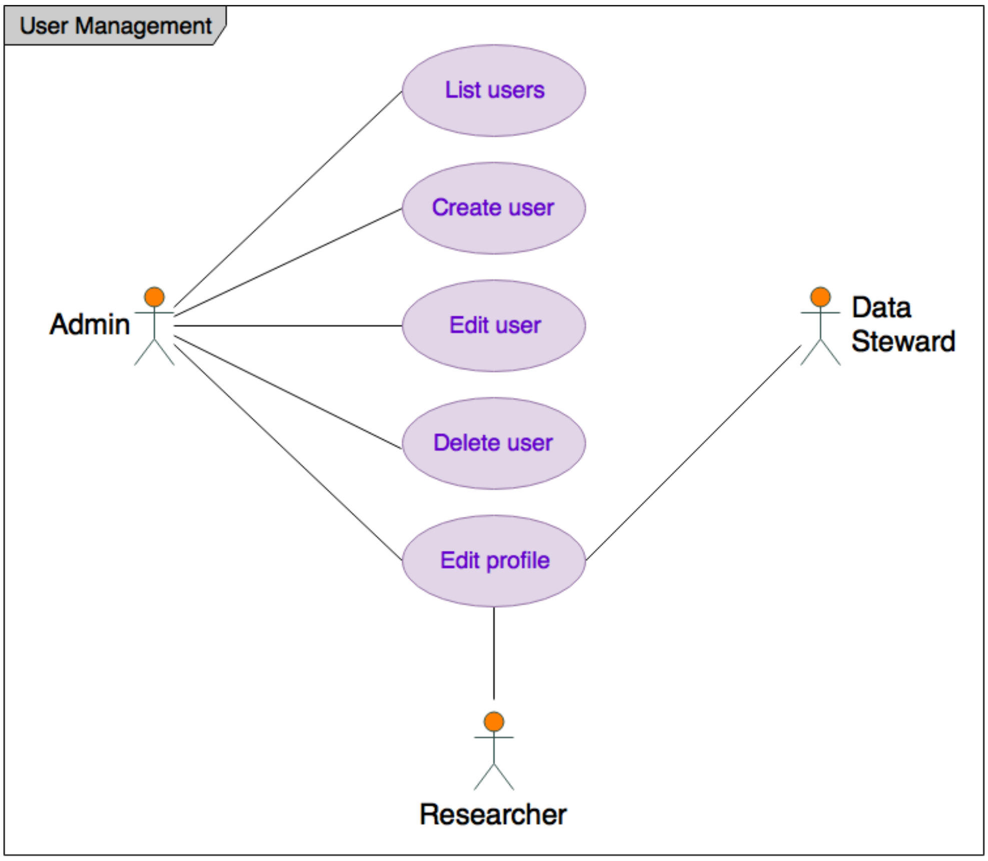

Figure 1.4: Use Cases: User Management

1.3.3.4 Group: Editor and migration tool

Use cases in this package contain mainly CRUD operations over a knowledge model. Then it contains `Publish knowledge model` which means to take current knowledge model with all changes which were made and create a package from it. This package can be used as a template for other knowledge models and it can be also distributed to outside world. And finally it contains `Upgrade knowledge model` which takes a knowledge model and apply new changes to it. All actions can be perform either by `Data Steward` or by `Administrator`.

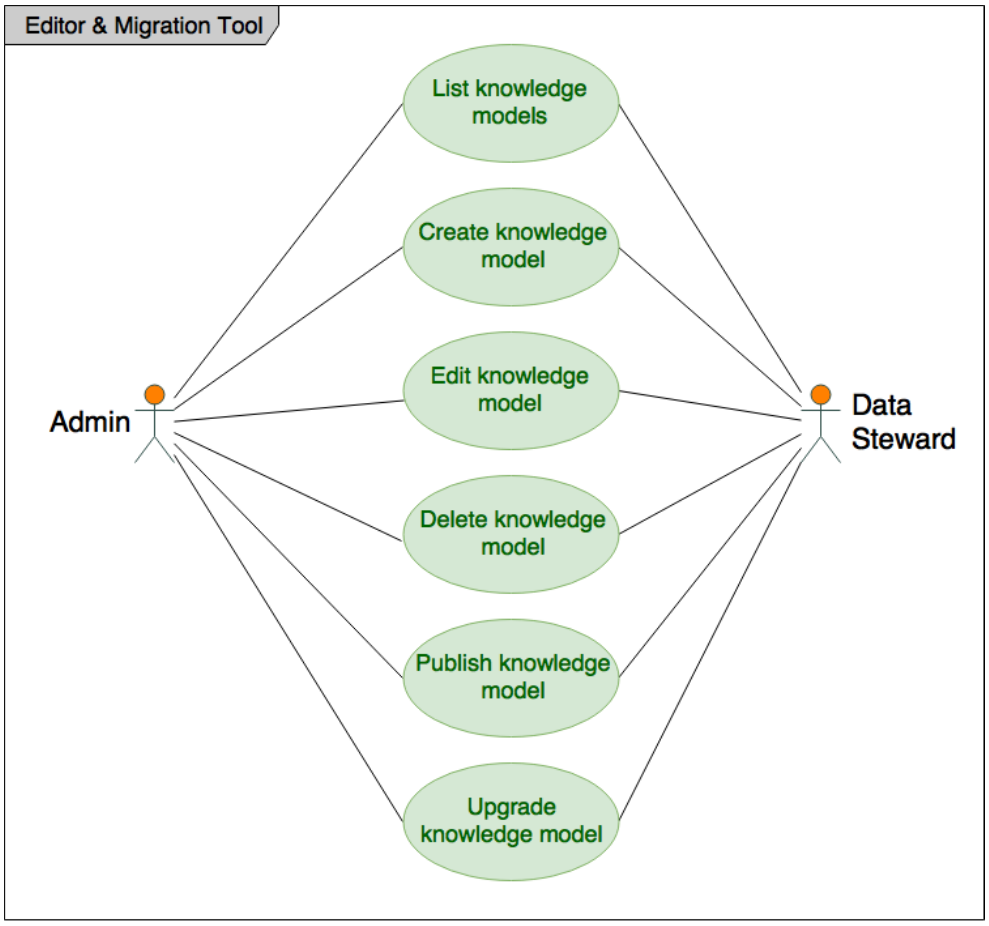

Figure 1.5: Use Cases : Editor And Migration Tool

1.3.3.5 Group: Package Management

Last analyzed domain is a package management. Package represents a unit which can be distributed to others. It contains information about itself and data from which a knowledge model can be built or upgraded. Briefly this group contains cases like `List packages`, `Import/Export packages` or `Delete packages`. All actions can be perform either by `Data Steward` or by `Administrator`.

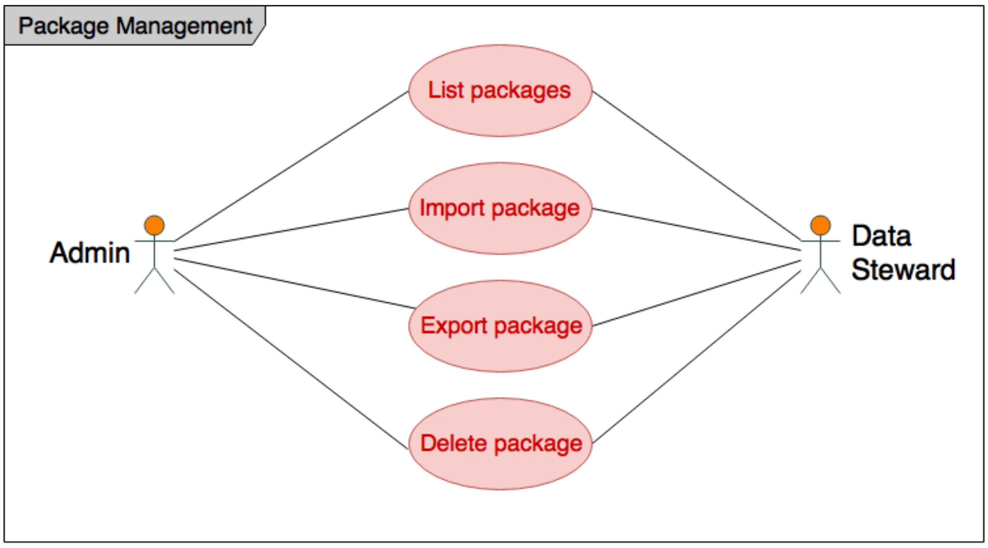

Figure 1.6: Use Cases : Package Management

1.3.3.6 Summary

For the summarization there is a table which map the use cases to the functional requirements. We can check that all of the functional requirements are covered by use cases.

Table 1.1: Check accomplishment of all functional requirements

	Functional requirement				
Use case	F1	F2	F3	F4	F5
Edit organization	X				
List users		X			
Create user		X			
Edit user		X			
Delete user		X			
Edit profile		X			
List knowledge models			X		
Create knowledge model			X		
Edit knowledge model			X		
Delete knowledge model			X		
Publish knowledge model			X		
Upgrade knowledge model				X	
List packages					X
Import package					X
Export package					X
Delete package					X

1.3.4 Domain Model

I divided the diagram into 4 parts according to a domain which they relate to – red, green, purple and orange.

For the entity identification I mostly used `UUID` (Universally unique identifier). It was necessary to use this approach for entities which can be shared by different institutions. This approach is able to not have a conflict when for example I export a package in my institution and in some other institution they will import this package to their system. In case we would use some sequence we would have to guarantee that this sequence is unique over all instances of the application. When we use `UUID` this situation is very unlikely.

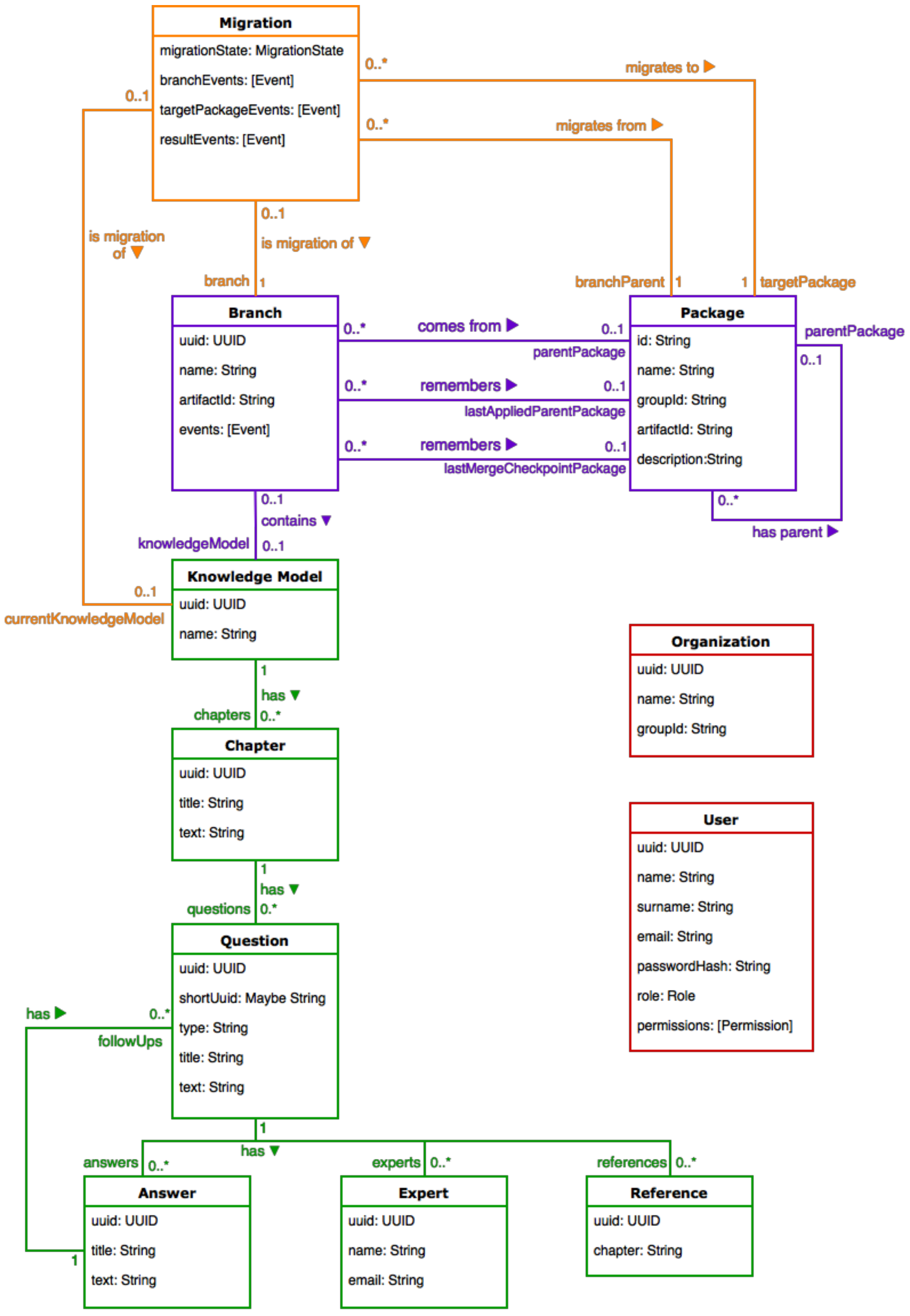

Figure 1.7: Domain Model

1.3.4.1 Green part

A green part describes entity related to `Knowledge Model`. All of the entities have an already mentioned unique identification through UUID.

1.3.4.1.1 Knowledge Model

`Knowledge Model` is a root of the knowledge model tree and it should represent a whole knowledge model to outside world. Except from unique identification (`uuid`) and it has a human-readable `name` and list of chapters.

1.3.4.1.2 Chapter

`Chapter` should pack related questions together. Therefore it includes a list of questions. Except from that it has a unique identification (`uuid`), a `title` and a little description about the chapter which is placed in `text` property.

1.3.4.1.3 Question

`Question` is a way through which it is planned to get a knowledge from researchers. Except from unique identification (`uuid`) it has also short UUID – `shortUuid` which connects the question with a book from Barend Mons – Data Stewardship for Open Science: Implementing Fair Principles [9] (the book is planned to be released in February 2018). In this book user can find more information about things related to this question.

Further `Question` has a `title` and a `text` for more detailed description. `Question` also has a type. Users can choose one from these following types:

- **Option** – User selects one from the predefined answers. In wizard it will be displayed as radio buttons
- **List** – User fills one or more inputs with his answers. It will be displayed as a list of inputs with an ability to add or remove inputs based on the count of user's answers.
- **String** – User answers the question in his own words. It will be displayed as a classic input.
- **Text** – User's answers will be probably a longer text. It will be displayed as a text area.

As you can see, a question can have answers (or not) based on the question type. Question can additionally has a list of experts (see 1.3.4.1.5) and a list of references (see 1.3.4.1.6)

1.3.4.1.4 Answer

`Answer` is presented just for the question type `Option`. It includes of course unique identification `uuid`, a `title` and a `text` for a description. It can also have a list of following questions which can more clarify the answer. We can see that this ability enables us to have an infinite depth of knowledge model tree.

1.3.4.1.5 Expert

`Expert` is here to be helpful for users which are not sure which answer they should choose or what they should fill in. Expert is identified by a unique identification (`uuid`) and it contains a `name` of the expert and his `email`.

1.3.4.1.6 Reference

Same as `shortUuid`, `Reference` connects a question with a book from Barend Mons – Data Stewardship for Open Science: Implementing Fair Principles [9] (the book is planned to be released in February 2018). Reference contains a unique identification (`uuid`) and a `chapter` property which refers to a chapter in this book.

1.3.4.2 Purple part

A purple part includes `Branch` and `Package`.

1.3.4.2.1 Package

`Package` is a wrapper which includes basic information about itself and about data which is needed for a building or an upgrading a knowledge model.

Package identification
`Package` is identified in the same way as maven dependencies in Java world[10]. `Package` has an `packageId` which consists of `groupId`, `artifactId` and `version`. These 3 coordinates should together create a unique identification of package in the whole world.

The `groupId` here represents an organization, e.g. `cz.ctu.fit` (FIT CTU). The `groupId` should include just letters, numbers (not in the begging of the `groupId`). Words/abbreviations are separated by dots. The idea is to start `groupId` with a more general identification, e.g. country, and ends with the most concrete one, e.g. faculty in my example.

The `artifactId` should be a name of the package which should be unique across organization (across `groupId`). The format is same as for `groupId` except of the separator. Here it is used dash instead of dot to separate words/abbreviations.

The last part of `packageId` is version. We should not create lower versions if higher version is already present. The format is simple here. Version is composed from 3 numbers which are separated by dots.

Together these 3 coordinates are joined by colon (`:`).

Package data
As it was already said main purpose of `package` is to provide data for a building or an upgrading a knowledge model. The data is represented in a form of events.

`Package` can extend some other already defined package. For this purposes it exists here an optional property `parentPackage`.

1.3.4.2.2 Branch

`Branch` is something like working directory, where we can perform changes on a knowledge model. `Branch` is identified by a unique identification (`uuid`). It is assumed that a user will have more open branches. Therefore it exists a `name` property which should be a human-readable naming of the branch. `Branch` includes of course a knowledge model (`knowledgeModel` property) which can be published as a package. Therefore `Branch` also has an `artifactId` property which is used for creating a right package identification (`packageId` – see 1.3.4.2.1).

Further it contains events which represent changes in a knowledge model which happened in the branch.

Finally package contains relationships to three packages. All three are optional. `ParentPackage` denotes the same thing as `parentPackage` in `Package`. `Branch` can be based on some package which extends. If `Branch` was created from some package it can be upgraded to a newer version of this package. For the migration process there exist 2 properties – `lastAppliedParentPackageId` and `lastMergeCheckpointPackageId` which are clearly technical and they are filled automatically. Property lastAppliedParentPackageId denotes what was the last version of parent which was applied to a knowledge model. And property `lastMergeCheckpointPackageId` references to a point in a branch history where the last changes from parent package was applied.

1.3.4.3 Orange part

An orange part belongs just to `Migration`.

1.3.4.3.1 Migration

`Migration` represents an entity which holds information which is used during a migration process.

Migration can be in 4 states:

- **Running State** – indicates a running migration without conflict and errors,
- **Conflict State** – indicates a migration with a conflict which needs to be solved by a user (information about the conflict is stored in the state),

- **Error State** – indicates a migration which ended with an error (the error is stored in the state),
- **Completed State** – indicates a finished migration.

`Migration` has a reference to a branch on which is the migration running. Further `Migration` has 2 properties which points to a target package to which we migrate (`targetPackage` property) and which points to a parent package of the branch (`branchParent` property).

`Migration` entity holds also 3 lists of events. The first list contains events which will be applied to a knowledge model (`targetEvents` property), the second list contains events which has been applied to a knowledge model since last migration (`branchEvents`). If no migration has been applied yet it takes all events from the time when the branch was created. And the last list contains events which were successfully applied to a knowledge model during the migration (`resultEvents`).

1.3.4.4 Red part

Last part is a little bit away from the others. It consists of 2 entities – `Organization` and `User`. They are not connected with others.

1.3.4.4.1 Organization

`Organization` is a very simple entity. It contains just a unique identification (`uuid`), a human-readable name of the organization and a `groupId` property. `GroupId` is used in package identification when we want to create a package from a knowledge model (see 1.3.4.2.1). It has to be in a format which I already described when I talked about `Package`.

Currently it is presumed that an organization will be just one over the instance of an application.

1.3.4.4.2 User

`User` entity contains what we would expect to contain. It has a unique identification (`uuid`), user's `name`, `surname` and `email`. `Password` is stored as a hash.

User permission

I introduced 3 roles in use cases – Administrator, Data Steward and Researcher. These roles represent most common types of users. But for a better flexibility in the future I created a set of permissions. So the actions are not validated against the roles but against the permissions. Even though the roles stay. When a user is created it is created with a role. Role has a set of permissions inside itself which are copied to a user entity during creation of the user. So the role serves here as a template for permissions. In the future there is a plan that the Administrator could add/remove a permission to specific user. This means that the user could do more/less actions that his colleagues. Then the role type would changed to type `Custom`.

Here is the list of all current permission in system:

- **UM_PERM** (User Management permission) – User can manipulate with other users – create/edit/delete them.
- **ORG_PERM** (Organization permission) – User can change information of the organization
- **KM_PERM** (Knowledge Model permission) – User can create/edit/delete a branch and edit a knowledge model
- **KM_UPGRADE_PERM** (Knowledge Model Upgrade permission) – User can upgrade a knowledge model to a new version
- **KM_PUBLISH_PERM** (Knowledge Model Publish permission) – User can publish a knowledge model as a package
- **PM_PERM** (Package Management permission) – User can manipulate with packages – create/import/export/delete them.
- **WIZ_PERM** (Wizard permission) – This permission is here for a future usage. It will allow to fill the Wizard.
- **DMP_PERM** (Data Management Plan permission) – This permission is here for a future usage. It will allow to manage generated Data Management Plans.

1.3.5 Activity Diagrams

There are no processes which are difficult to understand except one – a process of a migration. Therefore I prepared an activity diagram to clarify this process. A view on the migration process is very high-level here. More technical details will be provided in the following chapter (2).

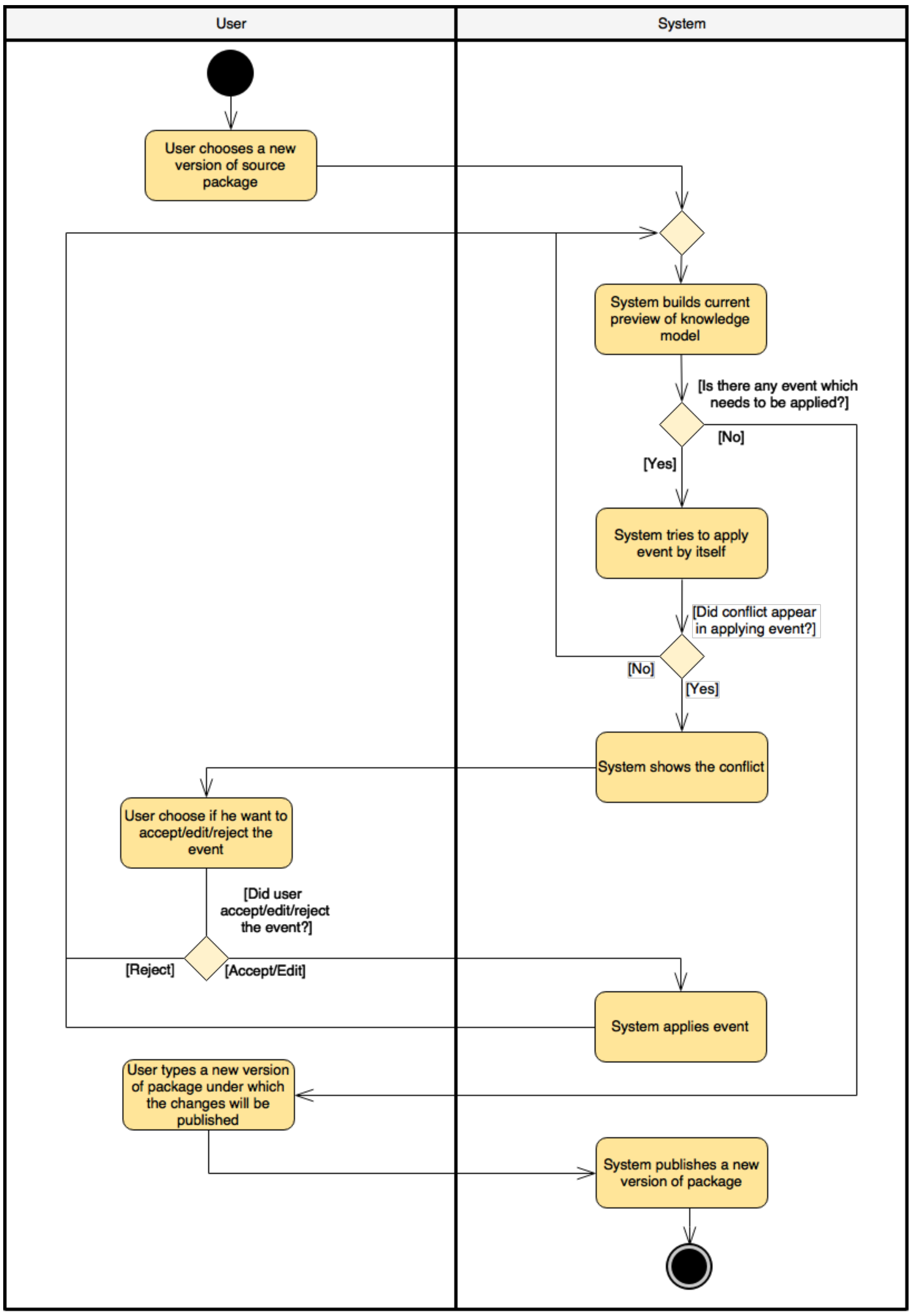
User
System
User chooses a new version of source package
System builds current preview of knowledge model
[Is there any event which needs to be applied?]
[No]
[Yes]
System tries to apply event by itself
[Did conflict appear in applying event?]
[No]
[Yes]
System shows the conflict
User choose if he want to accept/edit/reject the event
[Did user accept/edit/reject the event?]
[Reject]
[Accept/Edit]
System applies event
User types a new version of package under which the changes will be published
System publishes a new version of package

Figure 1.8: Activity Diagram: Migration

1.3.6 High-Level Deployment Diagram

I have already mentioned a few words about the related applications in project. The current project has 2 main parts – a portal part and a wizard part. These parts are connected through REST API.

1.3.6.1 Portal part

From the picture we can see (1.9) that the portal has a classic client-server architecture. The server is written in Haskel and the client is written in Elm[11]. The client communicates with the server through REST API. Further the server is connected to a MongoDB[12] database.

1.3.6.2 Wizard part

From the picture we can see (1.9) that the wizard has same architecture like the portal – a client-server architecture. Both the client and the server are written in Haskel. They use a technology Haste which has its own protocol that wraps the communication between the client and the server. Further the server is connected to a PostgreSQL[13] database.

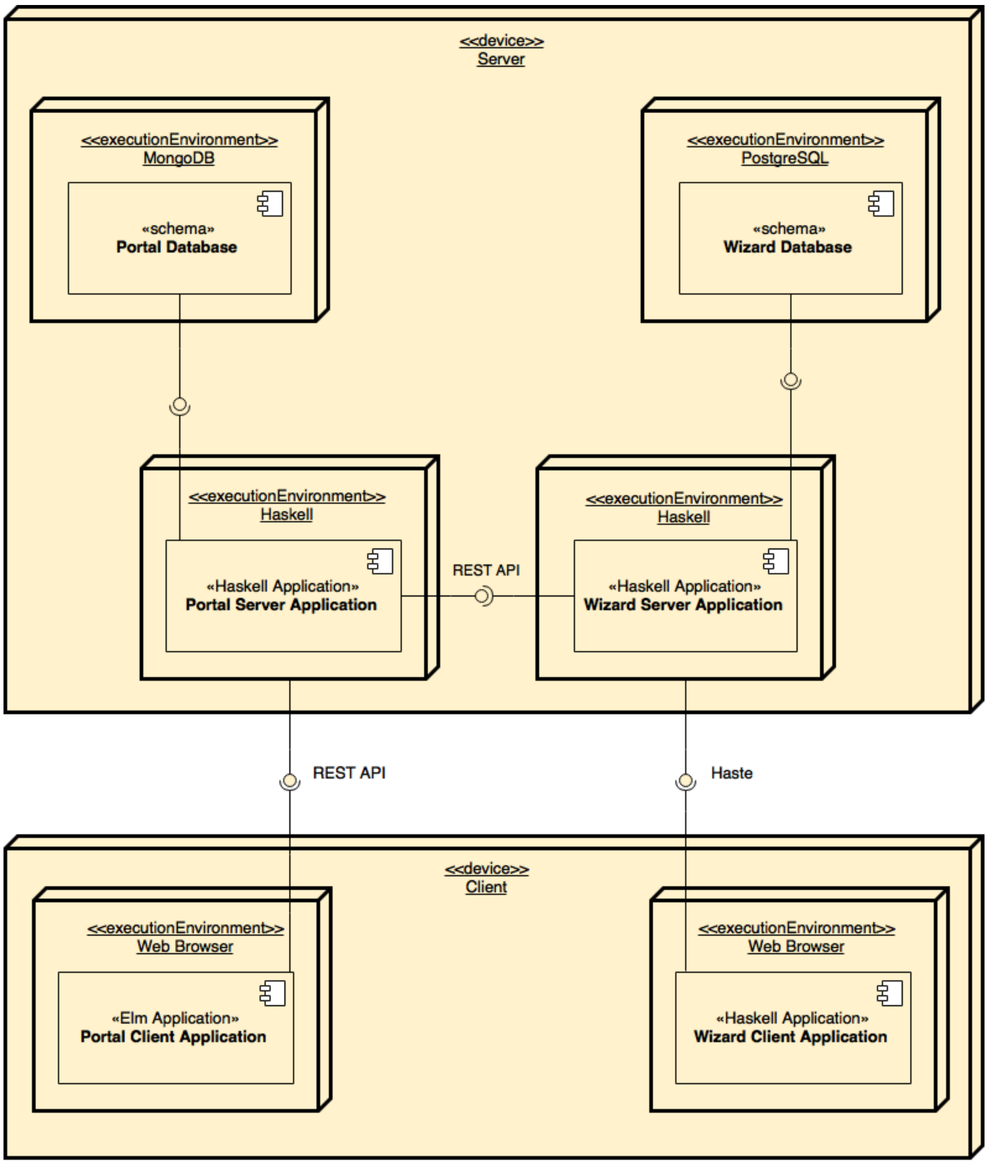

Figure 1.9: High-Level Deployment Diagram

1.4 Selected Tools

In this section I want to present my decision on which tools I chose and why.

1.4.1 General Requirements

The project has already been running so I had to keep the line with already selected tools. But still there were some blank spaces where I had to decide which technology or approach I will use.

A quick summary of the general requirements:

- select tools based on a current tools stack,
- try to keep world-proven standards and best practices,
- use appropriate tools with a respect to the problem domain.

1.4.2 Programming Language

I decided to keep the same programming language (Haskel[4]) as it is in other applications of the project. It should help current developers to easily maintain code and future developers to easily adopt to the project. Other advantage is also in sharing some common libraries and utilities across applications.

And the explanation why Haskel was chosen as a main programming language is its great type system which is very handy when you work with complex data structures (which is the case of this project). I really appreciated this advantage during my work at the project.

1.4.3 API

I have chosen REST[7] as the main application interface because currently it's an unwritten standard in today's web applications. Many tools and frameworks are ready to work with this kind of architecture. The application endpoints provide mostly just data with no additional logic so REST data-oriented approach perfectly fits to our case.

1.4.4 Web Framework

Haskel provides a few web frameworks which are worth to use. Here is quick comparison of them

Scotty [14]

+ it's currently used in 2 previous projects,
+ simple usage and project setup,
+ simple and satisfactory documentation,
+ favorite in Haskell community,
– not a full stack web framework,
– fewer plugins and extensions,
– needs additional libraries for managing a connection to a database.

Yesod [15]

+ most advanced Haskell Web Framework,
+ full stack web framework,
+ a lot of libraries,
+ the best documentation,
+ favorite in Haskell community,

– complicated usage and setup,

– not very straightforward,

– problems with connection to a NoSQL database.

Snap [16]

+ simple usage and project setup,
+ nice framework architecture which uses advance concepts of a functional programming,

– comparing to Scotty and Yesod – less favorite in Haskell community,

– not a full stack web framework,

– fewer plugins and extensions,

– needs additional libraries for managing a connection to a database.

During the first attempt I gave a chance to Yesod. After very nice start I had a problem with creating connection to MongoDB database. During the second attemp I tried Scotty and everything, including a database connection, worked well. So I kept the line with project and chose Scotty.

1.4.5 Build Tool

There are 2 options in Haskell with which we can build Haskell projects. First is **Cabal**[17] (Common Architecture for Building Applications and Libraries). This build tool is older and is used mainly in older projects. It has some disadvantages which complicates development (e.g. problem with non-deterministic installation of dependencies).

On top of Cabal it was built **Stack**[18] which fixes main problems of Cabal. Stack provides full backwards compatibility with Cabal. It uses the same configuration file for a package description and one extra file for its configuration (`stack.yaml`). On top of that `Stack` adds a multi-package project structure or it adds an automatic installation of a correct `GHC` version.

1.4.6 Mongo

As I mentioned in previous section I used MongoDB[12] as a primary database. Most of the data is tree structured so document-oriented style of storing data should be more suitable then a model for storing data in a relational database. And because nobody knows what data exactly the researchers will want to store it is better to be less strict and be more open to potential changes of a data model.

1.4.7 JWT

JSON Web Tokens[19] (JWT) are becoming more and more popular as a standard how the token should look like. The JWT is a classic token which is used in an authorization process but its unique structure offers to store user-defined data inside the token. Normally it is used for storing information about user, his permissions, etc. This payload is encoded by base64 and signed by a unique secret key on server. So if a signature is valid, the client can be sure that the data was not modified. And of course server can use this information as well when user sends a request with this token. It can save some calls to a database for retrieving user information.

1.4.8 Continuous Integration

I did not want to spend much time with a deployment but I wanted to continuously present my results of work. That brought me to set up a continuous integration process (CI). The goal was to build a target which can be easily and independently deployed to a server every time I do a change in code.

In talking about CI I have to start where the code is stored. The project code is kept in a Git repository. Further it is published under Apache License 2.0 so it is an open source. These reasons led us to choose GitHub[20] where the most open sources projects live. Regarding to the usage of GitHub we chose Travis CI which has a great integration with GitHub and for open source projects, it is free.

Because Haskell needs to be compiled for each platform separately (Windows, Linux, macOS), it could cause platform specific problems. And even the build would be much more complicated because for compiling a Windows version we have to compile it on Windows, for a macOS version we have to compile on macOS, etc. The problem in future could also bring some specific libraries which program needs from system. I tried to avoid this problem by using Docker. So the output of CI process is not runnable binary but Docker image.

Because there were no free servers where I could run my application I had to set up a new server. The server was chosen with an idea that all projects will be deployed here in the future. I prepared the server together with Jan Slifka. He worked on a client portal application and he needed a server, too.

Together we set up a Docker Registry[21] on the server so we could easily store our build images. We also set up a general server proxy (we used Nginx[22]) which holds certificates for our applications and proxy requests to them. All components are wrapped in Docker.

The usage of docker greatly simplifies a deployment process to a new server. Now it is possible to deployed to any platform we want (Linux, Windows, macOS). There is just one requirement – server has to have installed Docker tools. Then we can just download Docker images from the Docker Registry and run them. I hope this approach will help with an expansion of the project.

Workflow summary:

1. Do a change in an application and commit/merge the change to a master branch.
2. After a successful push, Travis will detect a change and run a build.
3. The result of the successful build is a Docker image which is pushed to our Docker registry.
4. For a deployment of the image to our server it is needed to connect to a server, pull a new image and restart the application.

As we can see the process is not fully automatic. But even so it very helped me during development and it surely saved a lot of my time.

1.5 Applied Concepts of functional programming

Each of us probably knows concepts and common patterns in an imperative programming. But if we switch to a pure functional paradigm we have to change our mind and start thinking about the problem and the way, how to solve the problem, differently. Because nowadays the functional programming is not very spread (for example I did not have any subject about it during my studies) I will describe a few concepts which I used and found very useful. We can see that some features from functional programming are already implemented in originally object programming languages. I can name, e.g. a pattern matching, a Maybe type, an idea of transformation functions for a manipulation with a list.

1.5.1 Few words about Haskell

Haskell[4] is a pure functional language with a great strong typed system which is built on an extended version of the Damas-Hindley-Milner type system [23].

Here are the most important language features:

- lazy evaluation (a result is not computed unless we want to show the result),
- pattern matching (deconstruction of data according to a pattern),
- list comprehension (a way how to filter, transform and combine lists),
- type classes (interfaces which define some behavior).

I will not describe these features in details. Instead of that I would like to focus on concepts which I used most and found very useful for everyday work.

1.5.2 Working with list

Functional languages use as a main enumeration type a list (compares to an array or a collection). Working with enumeration types in a way, that the code is effective and readable, has been a problem since the programming began. No approach is the best for all cases. The way how Haskell works is to compose data transform functions, use lazy evaluation and compute the final list as late as possible.

Here we can see quick comparison of method written in Java[24] and code written in Haskell. To be fair to Java I chose one of the examples which fits more to Haskell. Of course we can find cases where Java would win in clarity and effectiveness of code.

The purpose of the function (method) below is to take just even numbers from incoming list and return a list of the second powers of the numbers.

Listing 1.1 : List transformation in Java 7

```java
public List<Integer> squareEven(List<Integer> inArray) {
  List<Integer> outArray = new ArrayList<>();
  for (int i = 0; i < inArray.size(); i++) {
    Integer elem = inArray.get(i);
    if (elem % 2 == 0) {
      outArray.add(elem * elem);
    }
  }
  return outArray;
}
```

Listing 1.2 : List transformation in Haskell

```haskell
isEven :: Int -> Bool
isEven x = x `mod` 2 == 0

square :: Int -> Int
square x = x * x

squareEven :: [Int] -> [Int]
squareEven = fmap square . filter isEven
```

As we can see the code written in Java can easily start to be very unclear in this case. On the other hand Haskell composing functions is very clear, intuitive and together with list transformation functions like `fmap`, `filter` or `fold` code can be very expressive. And if we write a test for each simple function, we can be pretty sure that the whole transformation will work.

1.5.3 Error Handling (Maybe, Either)

Many programs are getting to a state where each place in a complicated program can throw an exception and it is unable to think in a context of all these error states which can happen.

Another common problem is calling a method on an object which is actually `null`. This situation can simply happen when a function does not return results for all inputs, e.g. finds an object in a database, parses an integer from a string, etc.

In functional world there exists a concept of `Maybe` and `Either` which tries to face these problems [25].

1.5.3.1 Maybe

First is `Maybe`. `Maybe` is a structure which wraps a value with a context. Most common usage of a `Maybe` is in a function which can either produce a value or not. Normally you would return a value or `null`. But this can easily lead to a bad state. Therefore it is better to use a `Maybe` and returns a value with a context. So instead of a value we return a value wrapped in `Just` and instead of `null` we return `Nothing`.

Listing 1.3 : Definition of Maybe from standard library

```haskell
data Maybe a = Nothing | Just a
```

But this improvement causes a more difficult usage when we work with a value (because it is wrapped in `Maybe`). Of course, we can ask if a value is inside and then unwrap the value. But we can imagine that it would lead to a code which would be totally unreadable.

Fortunately Haskell provides a support which simplifies the usage. As we have already seen in this chapter Haskell has a function call `fmap`. This function is a typeclass method of a class `Functor` and `Maybe` of course implements an instance of this typeclass.

Listing 1.4 : Definition of Functor from standard library

```haskell
class Functor f where
  fmap :: Functor f => (a -> b) -> f a -> f b
```

A usage with `Maybe` is simple. The function `fmap` (or the symbol `<$>`) can be used instead of a classic function application. A benefit, which this special application adds, is that if a value exists inside the `Maybe`, it unwraps the value and applies the function to the value. If not, it immediately returns `Nothing` so nothing bad happens.

Here is a quick example of usage:

Listing 1.5 : Example of usage of Functor

```haskell
-- 1. Define function without usage of Maybe
multiplyByTwo :: Int -> Int
multiplyByTwo a = a * 2

-- 2. Apply function to the maybe through fmap
multipliedValue :: Maybe Int
multipliedValue = multiplyByTwo <$> readMaybe "2"

nothing :: Maybe Int
nothing = multiplyByTwo <$> readMaybe "2a"
```

1.5.3.2 Either

Using `Maybe` is very useful but sometimes you want to inform user what happened wrong. For this purpose there is `Either` which can hold a value or an error.

Listing 1.6 : Definition of Either from standard library

```haskell
data Either a b = Left a | Right b
```

Otherwise working with `Either` is very similar to working with `Maybe`. You can use `fmap` for simpler function application, use `Left` and `Right` constructor in pattern matching, etc.

I mentioned these 2 patterns because when I learned them I started to use them very often. It helped me to avoid an exception like `NullPointerException` and to think about the code differently – count with errors if there is a possibility that they can occur. So most of my function returns a value which is wrapped in `Either`.

Listing 1.7 : Example of a usage of Either

```
runApplicator :: Maybe KnowledgeModel -> [Event] ->
   Either AppError KnowledgeModel
runApplicator mKm events = ...
```

1.5.4 IO Handling (Monad)

When an application wants to communicate with outside world it has to use `IO` actions in Haskell. Typical `IO` actions are a reading files, a writing to files, a communication with a database or a modifying a global state.

So `IO` enables us to perform impure computation in purely functional world. If we are talking about `IO` we have to mention `Monad`. `Monad` is very strong tool but it is not easy to learn it. With `Monad` we can do a stateful computation, compute a calculation with side effects, etc. [26] So it is not a surprise that `IO` is Monad. Therefore it can use all of its functions and properties which `Monad` has.

Here is an example from a usage of `IO` (`Monad`) in my application where I want to demonstrate a usage of this concept.

Listing 1.8 : Example of usage of IO

```
findUserById :: DBPool -> String -> IO (Either AppError
   User)
findUserById dbPool uUuid = do
  let query = select ["uuid" =: uUuid] collection
  let action = findOne query
  maybeUserS <- runMongoDBPoolDef action dbPool
  return . deserializeMaybeEntity $ maybeUserS
```

We can see that the function uses a special construct `do` which composes actions inner the block into one. Then we see an action which communicates with outside world – `runMongoDBPoolDef`. If something bad happens (a connection to database failed), our function (`findUserById`) returns a failure. If not, function returns a result which is binded (unwrapped) to maybeUserS . Then we can processed an unwrapped value maybeUserS (in the example I deserialized the result). And finally we have to wrap back the value into the IO which is done by calling a return function.

1.5.5 Working with complex structures (Lens)

The last concept which I want to mention and which I used a lot is Lens[27]. Lens helps you with a changing or a traversing over parts of complex tree structures. But on the other hand, learning Lens is not very easy. For understanding Lens you need to learn Foldable and Traversable at first. The Lens is built on top of them and provides an abstraction which simplifies the work with these complex tree structures.

But for a basic usage of Lens you do not have to understand a whole concept. It is a same as with Monad. Even if you know basics, it still helps you a lot.

But finally I got used to usage of Lens during a development as much that I used Lens rules even for normal simple structures like Organization, User, etc.

The Usage of Lens

If we want to use Lens, it is good to use an language extension Template Haskell. Otherwise we will have to write a lot of code by ourself. Than we have to name properties of a structure with prefix _.

Here is an example of Lens-ready structures from my code:

Listing 1.9 : Define Lens-ready structure

```haskell
data KnowledgeModel = KnowledgeModel
  { _kmUuid :: UUID
  , _kmName :: String
  , _kmChapters :: [Chapter]
  }

data Chapter = Chapter
  { _chUuid :: UUID
  , _chTitle :: String
  , _chText :: String
  , _chQuestions :: [Question]
  }

makeLenses ''KnowledgeModel

makeLenses ''Chapter
```

Because one name of a function in Haskell can have just one definition (it is forbidden to overload functions) it is common to add to properties of a structure some unique prefix (like I add `km` and `ch` prefix to properties in my example).

Using `Lens` we do not miss the classic property read and set approach. Here is a comparison of classic way with the `Lens` way on example of reading and setting properties:

Listing 1.10 : Comparision of a standard approach with a Lens approach on a basic usage of getter and setter

```
let myKm = KnowledgeModel
           { _kmUuid = myKmUuid
           , _kmName = "My Knowledge Model"
           , _kmChapters = [chapter1, chapter2]
           }

-- 1. Getter
-- Classic approach
let name = _kmName myKm

-- Lens approach
let name = myKm ^. kmName

-- 2. Setter
-- Classic approach
let editedKm = myKm { _kmName = "EDITED: My Knowledge
   Model" }

-- Lens approach
let name = myKm & kmName .~ "EDITED: My Knowledge Model"
```

These notations are almost the same – `Lens` library in this example does not add anything special but what happens if we want to get all chapter UUIDs? Here is the comparison:

Listing 1.11 : Comparision of a standard approach with a Lens approach on a more difficult example

```
-- Classic approach
fmap _chUuid (_kmChapters myKm)

-- Lens approach
myKm ^.. kmChapters . traverse . chUuid
```

Here we see that the `Lens` approach starts to be more readable. And as we go deeper in a structure than `Lens` shows its benefit more and more.

Of course we can define our own getters and setters for `Lens`. It can be useful when we do not want to miss `Lens` composition and keep clear `Lens` interface. Here is one of my own setter for `Lens`:

Listing 1.12 : My own defined Lens setter

```
kmChangeChapterIdsOrder :: ([Chapter] -> Identity
   [UUID]) -> KnowledgeModel -> Identity KnowledgeModel
kmChangeChapterIdsOrder convert km
  = Identity $ km & kmChapters .~ orderedChapters
  where
    ids :: Identity [UUID]
    ids = convert (km ^. kmChapters)
    orderedChapters :: [Chapter]
    orderedChapters = concatMap getChapterByUuid
       (runIdentity ids)
    getChapterByUuid :: UUID -> [Chapter]
    getChapterByUuid uuid = filter (\x -> x ^. chUuid
       == uuid) (km ^. kmChapters)
```

As we can see – it is not that trivial and obvious as we would expect after the nice examples in this section. The purpose of the function `kmChangeChapterIdsOrder` is to change order of chapters which are stored in `_kmChapters` property. But the usage is simple. We can use this setter as a normal Lens setter.

Listing 1.13 : Example of usage my Lens setter

```
let uuids =
    [ "291ab0e7-8100-403c-9d01-9d6528d347dc"
    , "6c16165e-6db7-4bf3-8da4-4d527df33eaa"
    ]
let editedKM = myKm & kmChangeChapterIdsOrder .~ uuids
```

It is really hard to fully understand `Lens` library but when you did it helps you a lot. It brings the dot notation from object world which is very natural. But compare to the object world `Lens` functions can be composed together which we appreciate in a situation when we work with really complex structures.

CHAPTER 2

Design of The Migration Tool

2.1 Introduction

In the beginning of this chapter I talk about an idea how the migration tool was intended to be implemented (2.2 and I discuss here advantages and disadvantages of this approach and why the idea had to be rethought.

Further I quickly introduce Git which is a tool that also had to solve a problem of merging things together (2.3).

Next section My Proposal (2.4) is more about my ideas. I describe here my 2 suggested approaches, compare them to Git and discuss their benefits. Finally I chose one approach and recapitulate the whole proposed solution.

2.2 Current Approach

2.2.1 Core and Localizations

First idea was to keep a knowledge model in a core and its changes and modifications in localizations. Core looked just like the knowledge model now. Localization was a special entity which has the same base structure as knowledge model. Its main purpose was to add specifics for a field of research, or to add specifics for an institute. For creating a localization there existed rules which describes what should have been filled. Here they are:

- **Rule 1** – When I wanted to edit something from a core I had to mention a namespace `core`. When I wanted to add something new I had to add a namespace property with some other value than `core`.

- **Rule 2** – When I had a chapter in localization I checked if the chapter with given number already exists. If not, I added it to its right place (according to a number). If it existed I tried to apply all fields from the chapter to the existed chapter.

- **Rule 3** – A localization could hide a question in a chapter if there was a property `hidden` on the question and the namespace `core`.
- **Rule 4** – A localization could replace a whole question if properties `questionId` from the core and from the localization matched and the namespace was `core`.
- **Rule 5** – A localization could extend a core question if the namespace was not set to `core` and properties `quesitonId` from the core and from the localization matched. For a purpose of adding an answer, an expert or a reference, there existed properties with prefix `add` in the localization – `addAnswers`, `addExperts` and `addReferences`

2.2.2 Precompiler

A precompiler was an application written in Python which had a command-line interface and should do a transformation from a core and localizations into one final knowledge model. This model should have been deployed and used in wizard. If we did not have any localizations, we could take a core and use it as final knowledge model. If not, the precompiler did its job and merged localizations to core.

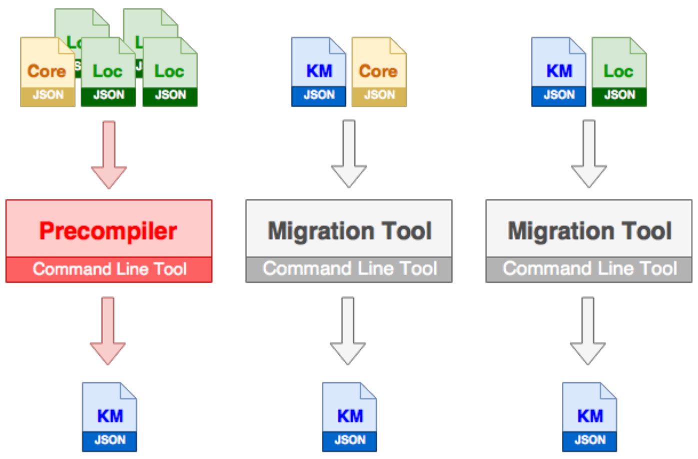

Figure 2.1: Precompiler

But the precompiler was not able to merge for example an upgraded core into a current compiled knowledge model (see picture 2.1). For this case it was decided to create a migration tool which should handle the cases when we upgrade core or localization and we want to apply these changes to a current compiled knowledge model.

The tool should have command-line interface same as the precompiler. But it should have been written in Haskell. This was the state when I joined to project. And the migration tool should have been the content of this diploma thesis.

2.2.3 Problems in current solutions

The big problem in this solution was that changes in a knowledge model could have been done just by person with technical knowledge because knowledge models (represents by core and localizations) were stored in JSON. So the person had to have a knowledge of this format. The person should also have to have a knowledge about the rules how to create a localization. This was planned to be solved by some editor in future.

Further problem was in a management of core and localization files. It was done manually. For the core and some example localizations there existed a Git repository. But for other localizations the files had to be stored on a disk and someone had to be an authority who has a right versions of these files and distribute these files to others.

And because the core and localization were not homogeneous we have 2 cases in merging (see picture 2.1). And as we can imagine a merging of 2 different files without any additional logic was not very straightforward process.

Because of all of these pitfalls I decided to change the concept from core and localization to more simple one.

2.3 Git

Here I would like to describe a Git which is currently so far the world most popular version control tool[28]. The reason, why I mention here, is that it includes a very advanced merging tool which inspired me in designing my migration tool.

I inspired myself in Git with 2 things. First I checked how Git handles a process of branching and a process of merging/rebasing. Second I took over the naming from Git. The reason was a quicker adoption to the project and a faster understanding how it works for new developers.

2.3.1 Common properties

The migration tool and Git has some common properties. Here they are:

- Git is a distributed system, same as instances of the project will be decentralized spread around the world.

- Git has a commit which bundles changes into one package. In my application I defined an event which is equivalent to a change in Git and I bundled these events to a package.
- Git uses a lightweight branches so the Git branch is just a pointer to some commit. I use branches in my application in the same way. They are also very lightweight and have just pointers to their parent package.
- Git can have multiple branches which can be merged one into another. In my application we can create more branches, too. And the merge process is called upgrade.
- In Git we can perform changes and then commit them or revert them. I provide these abilities, too.

We can see many common properties between the Git and the migration tool. But there are some differences which I will discuss later. Further my approach of merging (upgrading) is more sophisticated and can be used just for this case which is a big restriction compares to Git which has more wider usage.

2.3.2 Branching, Merging and Rebasing

One of the big benefit, which Git brings, is that it allows users to work in parallel. A user, which wants to start working on a new feature, creates a new branch from the master branch. After finishing his work, he merges its branch into a master branch (and possibly fix conflicts). This simple worklow is called feature branching [29] and it is currently very spread across developer teams. For an integrating changes from one branch to another, Git provides 2 approaches – `merging` and `rebasing`.

2.3.2.1 Merging

If we want to merge some branch (e.g. our feature branch – iss53) to another branch (e.g. a master branch), first we need to checkout (switch) to the master branch. Then we can start merging. Git does a simple three way merge which uses the common ancestor of the two branches and their last commits (pointed to by the branch tips).

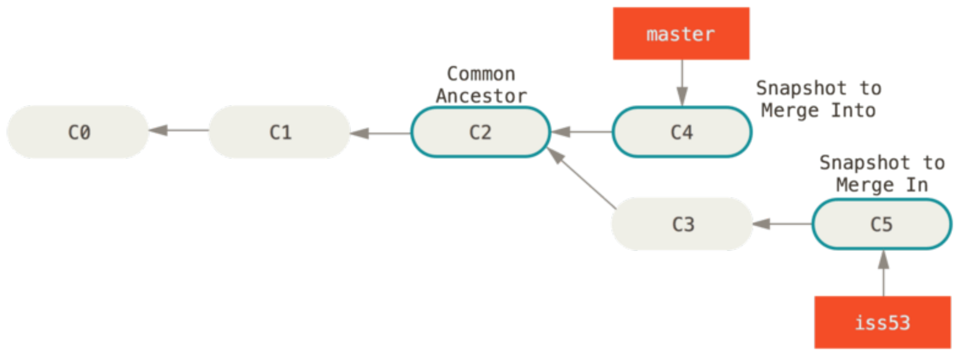

Figure 2.2: Git: State before merge (according to [30])

To join 2 branches together Git uses many of its strategies. But sometimes it may arise a situation where Git does not know what to do. Mostly it happens when more people edit same thing. This ends in conflict which needs to be solved by us.

But if not, Git creates a new commit (a merge commit) that includes changes results from this three way merge. Otherwise we have to resolve the conflict first and then perform the merge commit by ourselves.

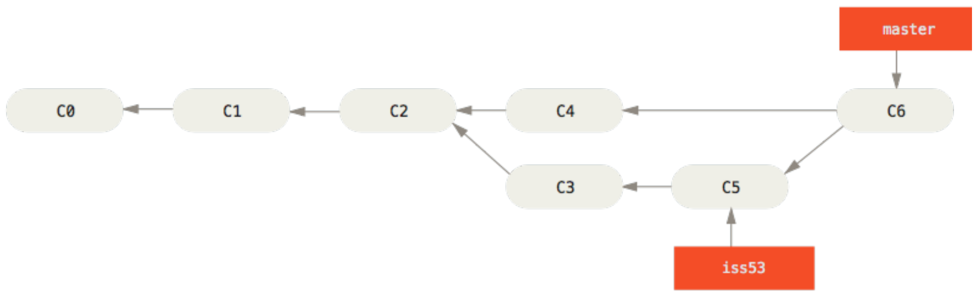

Figure 2.3: Git: State after merge (according to [30])

2.3.2.2 Rebasing

The second main approach how to integrate changes from one branch to another is rebase. We have more kind of rebases but for our needs the simple version is enough. Imagine we have the same situation as previous. We have 2 branches – one our branch (I will call it `experimental` branch in this case) and one master branch. And we want to integrate changes from our experimental branch to the master branch.

So first we need to checkout (switch) to the experimental branch and perform rebase against the master branch.

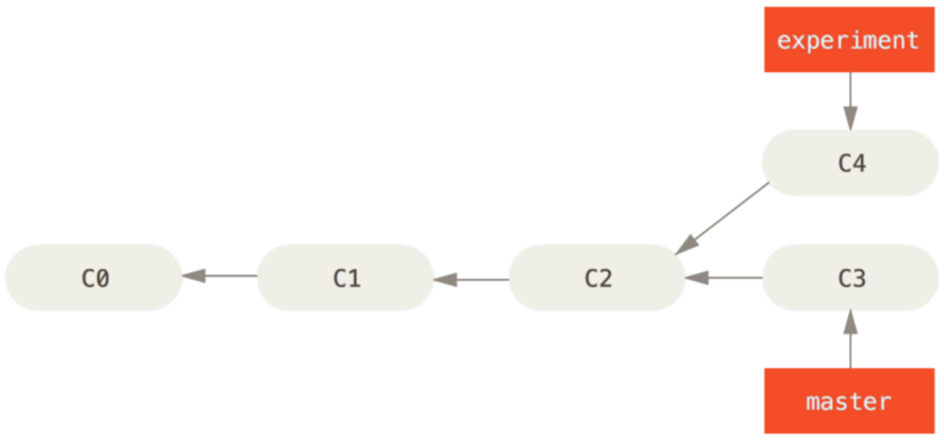

Figure 2.4: Git: State before rebase (according to [30])

Simply put, Git rearranges commits in our branch in order to put first commits from the branch against which we are rebasing (master branch in our case – commit `C3`), and then it tries to apply our commits (commit `C4`) on top of it. If some conflict appears, we solve this conflict within the problematic commit. After a successful run of rebase command we should end in this state (see image 2.5).

We can see that the new commit has a different designation (`C4'` instead of `C4`). It is because Git marks a commit by a hash. This hash is computed from the actual state. And we can see from the picture (2.5) that the state changed (our parent commit is `C3` instead of C2 and we may differ in some files if we had to solve a conflict).

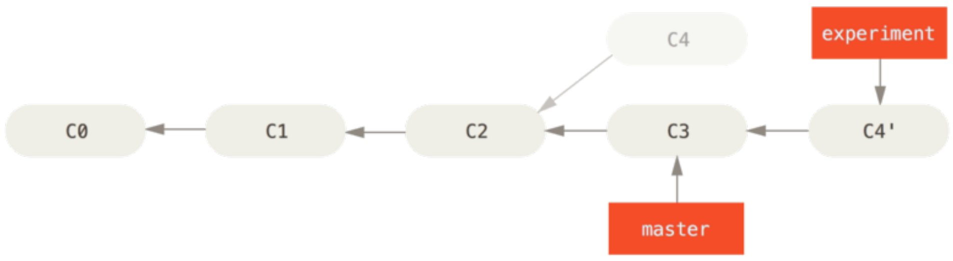

Figure 2.5: Git: Rebase in progress (according to [30])

And for finalization we need to checkout to the branch against which we performed rebase (in our case to the master branch) and we perform a merge. But in this case this merge will not be a three way merge but the fast-forward merge. This merge just change the pointer of the master branch to the new commit (`C4’`).

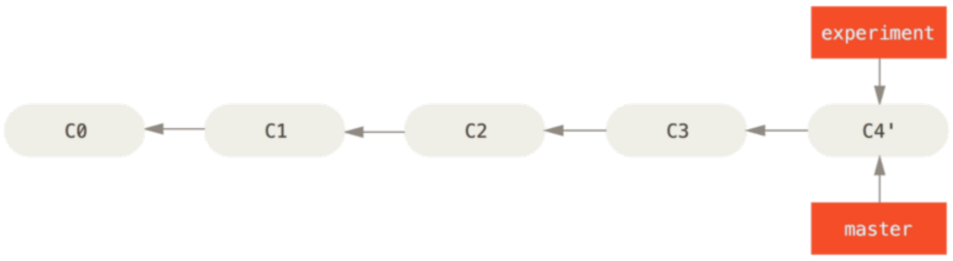

Figure 2.6: Git: State after rebase (according to [30])

2.3.2.3 Comparison of merging and rebasing strategy

An advantage of this approach is a cleaner history which seems to look like more linear (comparing to a history where we combine branches by three way merge). But otherwise there is no difference between these 2 approaches.

2.4 My Proposal

Now I would like to talk more about the solution which I created.

2.4.1 Event-based Approach

2.4.1.1 Introduce Event sourcing

I decided to leave the current proposal because of the pitfalls which it has and because the migration process would be very hard and difficult. Instead of that I proposed another approach.

My idea was inspired by an event sourcing pattern [31]. I decided to transfer the source of truth from a knowledge model to a serious of events from which we can build a knowledge model. So by an application of these events one after another we will get a knowledge model. And because the knowledge model is used widely in application I decided to compile the knowledge model after each creation of a new event and I cached the knowledge model in database. This pattern is known as a read model[32].

This approach brings a huge simplification to a process of a merging (an upgrading). Until now we had to compare 2 knowledge models or knowledge model to localization, find changes and make some relevant application of these changes to a final knowledge model. On top of that we had to detect conflicts which could lead to unwanted changes of the knowledge model and let the user to choose a solution.

Currently I transform this complicated process to just merging 2 series of events. The final knowledge model will be then compiled from merged series of events.

2.4.1.2 Events

I designed events as something which can not be modified. Event is identified by a unique identifier (`uuid`), the same as most of other entities. If we want to change event, we have to produce a new event with new `uuid`. This mechanism leads to the fact that one event should be unique over all instances of the project. Every event should be also fully independent to other events. So the event can not be embedded in other event. So it has to have all information for a successful application to a knowledge model. Of course it can happened that the event could not be applied to a current knowledge model because it expects some state which is not currently present. Then an application of this event ends in an error state.

Here you can find a list of all available events:

- **Knowledge Model**
  - `AddKnowledgeModelEvent`
  - `EditKnowledgeModelEvent`
- **Chapter**
  - `AddChapterEvent`
  - `EditChapterEvent`
  - `DeleteChapterEvent`
- **Question**
  - `AddQuestionEvent`
  - `EditQuestionEvent`
  - `DeleteQuestionEvent`
- **Answer**
  - `AddAnswerEvent`
  - `EditAnswerEvent`
  - `DeleteAnswerEvent`
- **Expert**

  - `AddExpertEvent`
  - `EditExpertEvent`
  - `DeleteExpertEvent`
- **Reference**
  - `AddReferenceEvent`
  - `EditReferenceEvent`
  - `DeleteReferenceEvent`
- **Follow Up Question**
  - `AddFollowUpQuestionEvent`
  - `EditFollowUpQuestionEvent`
  - `DeleteFollowUpQuestionEvent`

We can see that except from a knowledge model, events cover operations `add`, `edit` and `delete` over the each entity.

As I have already said the event should be fully independent. For that reason I had to distinguish 2 cases for manipulation with questions. First – I manipulate with a question which is directly under some chapter. Second – I manipulate with a question which is directly under some answer (follow up question). Therefore I created two separated groups of events – `Question` and `FollowUpQuestion`. Their only difference is that events in a `FollowUpQuestion` group contain also `answerUuid` to identify under which answer they belong to.

2.4.1.3 Disadvantages of event-based approach

Every solution has also some disadvantages. I found a few here compares to a previous solution.

Readability of model
When we had a core and localizations we could easily check without any tool what we actually have. With increasing number of localization it was getting harder and harder but still the format was far away more readable then a list with hundreds of events.

Slower response time
When we do a change in knowledge model we have to recompile a whole list of events. With increasing amount of events this process will last more and more time. In the future this problem will have to be solved. I propose 2 improvements. First – compile events in different thread so this job will not extend the response time. Second – in some cases it is possible to cache knowledge model and use it as a snapshot. Adding a new event would mean to just take a snapshot and apply one more event to it.

2.4.2 Applicator

First component which needed to be done is Applicator. Purpose of Applicator is simple – it should take events and optionally a knowledge model and produce a new knowledge model with applied event. Or if the application failed, it should produce an error.

Now I want to look more deeper on what we can change in the knowledge model and how these changes are represented in events.

In following paragraphs I use a term – node. By node I mean one of the type from a knowledge model tree – either knowledge mode, chapter, question, answer, expert or reference.

2.4.2.1 Add node

The add type of event specifies where the new node should be placed in knowledge model and what should be placed there. Of course, there are some limitations as we could not add a chapter under an answer. Next this event contains properties specific to the type of the node, e.g. `text`, `title`, `name`, etc. It does not contain any information about children. So if we want to add a new chapter with a question, we have to add the chapter at first and then add the question.

2.4.2.2 Edit property on node

When we want to edit a property on some node we use one of the edit events. Edit event contains where the node is placed in the knowledge model and what we want to change in the node.

2.4.2.3 Edit order of children in node

Edit event does not include children because the event could quickly become very large. Instead of that it contains a list of `uuids` of its children. If we want to make a change of an order of the children, we change the order of `uuids`.

2.4.2.4 Remove node

For deleting of some node we just have to include information where the node is placed in knowledge model tree. That is all.

2.4.3 Migrator

During the design and the implementation of Migrator I changed the idea about how the migration should work. Here I would like to describe these 2 approaches, compare them and explain why the final solution is better than the other one.

2.4.3.1 First Approach

First I had an idea which was similar to a rebase in Git terminology. I wanted to rearrange events in versions. The plan was first apply all events which came from a parent and then apply events which were done by myself. In Git we would checkout to our branch and run rebase against parent branch.

Description of how to merge events

The big benefit was that conflicts which appeared in the parent was already solved in the parent. This was very useful in the case when my parent is not core but it is a package which has its own parent package and this parent package has its own parent package, etc. (see picture 2.7). All conflicts which appeared during migration processes in parents were solved there and we just have to solve conflicts which appeared during the application of our events.

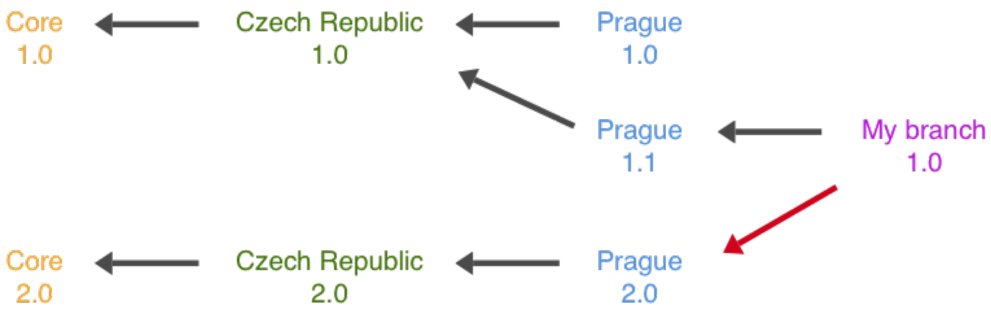

Figure 2.7: A package dependency structure. Red arrows means the package on which we want to upgrade.

The opposite approach would bring conflicts from all parents whose events had not been applied yet. In example from picture (2.7) it would mean to solve conflicts which would come from `Core 2.0`, `Czech Republic 2.0` and from `Prague 2.0`.

But in our case conflicts could appear just in applying our changes. These changes were applying in the end so we could manipulate with them as much as we wanted because we could not caused a conflict to anyone. But the price for unlimited interventions was that a new version of package had to include all previous events, not just a diff of events to previous version (see picture 2.8). We can see that the events in new `core 2.0.0` are not a concatenation of package `core 1.0.0` and `core 2.0.0`. They differ in event number 5 which is missing and instead of it there is a new event number 7. I wanted to show there a state when due to the need of resolving a conflict we had to modify the event. And because there is forbidden to edit events it meant to create a new event.

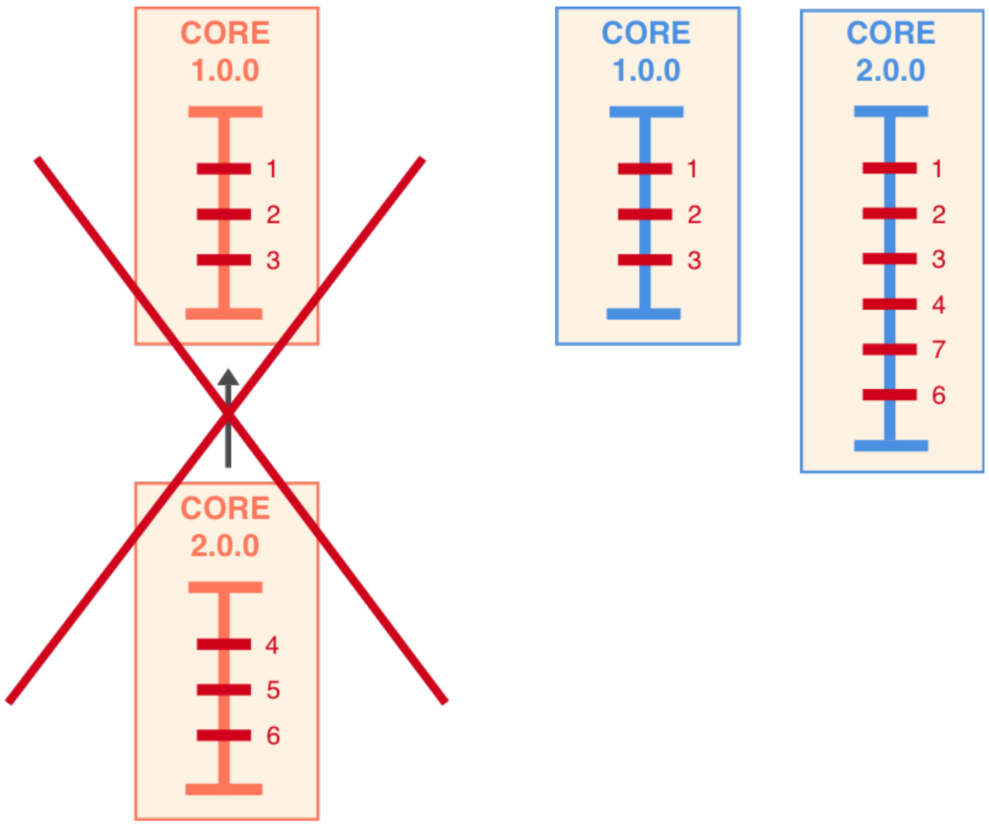

Figure 2.8: Structure of package

Now I prepare an image (2.9) where I would like to summary how the events were put into a branch during upgrading of the knowledge model. On the left we can see a `core` branch which offers itself in 2 versions. The middle branch is our branch which is based on a `core 1.0.0`. We can see yellow line which marks the events from the `core` and the light green line which marks our events. And the right branch is based on a `core 2.0.0` and shows how the events were rearranged. First they are events marked with blue line. These events are copies of events from `core`. After them there are our events with the light green label. We can see that one event is missing and one is new. This is the situation which I described above – a conflict appeared and it had to be solved by replacing a problematic event with a new one.

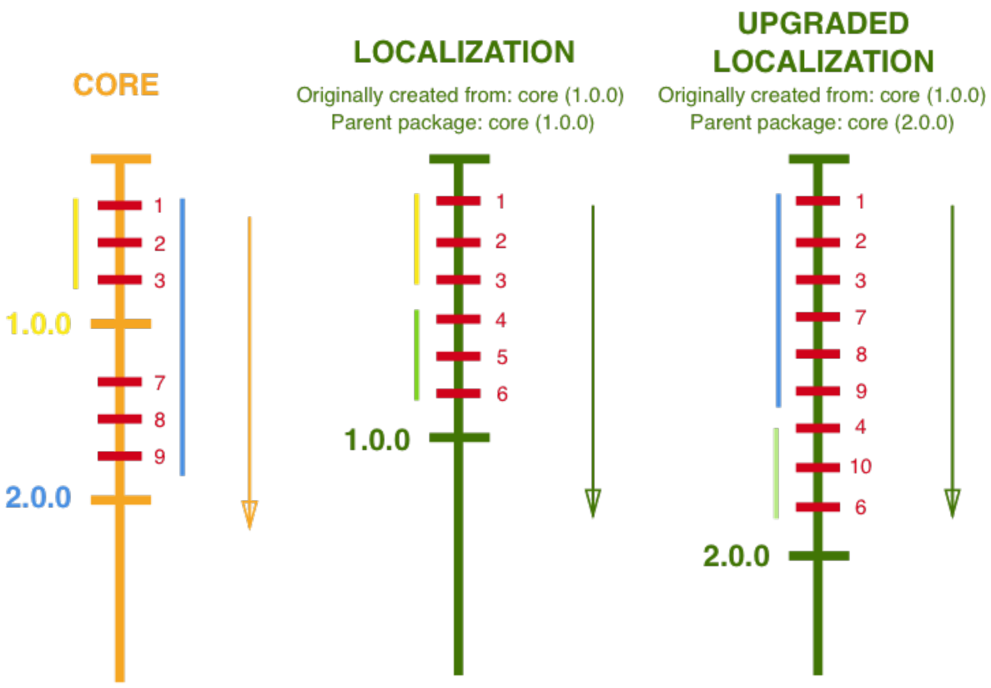
CORE
1
2
3
1.0.0
7
8
9
2.0.0
LOCALIZATION
Originally created from: core (1.0.0)
Parent package: core (1.0.0)
1
2
3
4
5
6
1.0.0
UPGRADED LOCALIZATION
Originally created from: core (1.0.0)
Parent package: core (2.0.0)
1
2
3
7
8
9
4
10
6
2.0.0

Figure 2.9: Upgrade localization branch from `core 1.0.0` to `core 2.0.0`

Conflict solving strategies

I prepared 5 strategies how to behave when a conflict appears. The migration process is started by using Applicator which applies all events from a parent and build a knowledge model. Then the migration tool takes each event which was created by us in our branch and tries to apply these events to the current knowledge model. To reveal a conflict I made a table where are described all situations which can happen. So we know if we are in a situation where a conflict can appear or not.

Here is the table (2.1) where I assign migration strategies to the situations. The situations are distinguished by comparing a current solving event with new events which came from a new version of parent package. Originally the table was much more longer because there was a row for each combination of events. But finally I found that we can reduce the combinations just to combinations of types of events (by types I mean – `add`, `edit` or `delete` types of event).

For comparing types of events I defined a binary relation between the types of events. Here is its definition:

- `"="` – means that both events are at the same level (depth) of the knowledge model.

- "<" – means that an event on the left side of the operator is placed deeper than an event from the right side.
- ">" – means that an event on the right side of the operator is placed deeper than an event from the left side.

To be sure it is understandable I will describe the first row. In parent events there exists some event which belongs to a subtree of some node which we edited in our localization. We can see that this situation does not cause any conflict. So we use `No Conflict` strategy. The symbol `"-"` in column `Strategy` means that this situation can not happen.

From table (2.1 we can see that there existed 4 strategies. For `Diff Tree` and `Pool` I had to create auxiliary precomputed structures for speed up the migration process.

- **Diff Table** – It is a hash map where key is a node `uuid` and values are events. By node `uuid` I mean `chapterUuid` for `AddChapterEvent`, `questionUuid` for `EditQuestionEvent`, etc.
- **Diff Tree** – It is a tree which has the same structure as the knowledge model. But the node is a flag which marks if the node was edited or not
- **Pool** – It is a list where the unused events are stored. It is used when we want to remove some node and we do not want to loose all the events which build these node and its children. So we save these events into a pool for later usage.

And here are descriptions of the concrete methods.

1. **No Conflict**
    - **Precondition:**
        - Matched if any other precondition did not match
    - **View:**
        - Nothing to show.
    - **User Actions:**
        - No user actions.
    - **Solution:**
        - Server will apply event without interrupting the user.
2. **Choice**
    - **Precondition:**

Table 2.1: Migration strategy: First approach

Parent Branch	**Relation**	**Current Branch**	**Strategy**
add	$<$	edit	No Conflict
edit	$>$	add	No Conflict
edit	$>$	edit	No Conflict
edit	$>$	delete	No Conflict
edit	$<$	edit	No Conflict
delete	$>$	delete	No Conflict
delete	$=$	delete	No Conflict
delete	$<$	edit	No Conflict
delete	$<$	delete	No Conflict
add	$=$	add	No Conflict
edit	$=$	edit	Choice
edit	$=$	delete	Choice
delete	$=$	edit	Choice
add	$<$	delete	Diff Tree
edit	$<$	delete	Diff Tree
delete	$>$	add	Pool
delete	$>$	edit	Pool
add	$>$	add	-
add	$>$	edit	-
add	$>$	delete	-
add	$=$	edit	-
add	$=$	delete	-
add	$<$	add	-
edit	$=$	add	-
edit	$<$	add	-
delete	$=$	add	-
delete	$<$	add	-

  - We have an edit action in our branch and there is a record with the same node `UUID` which refers to edit or delete action in Diff Table.
  - We have a delete action in our branch and there is a record with the same node `UUID` which refers to edit action in Diff Table.
- **View:**
  - User can see a preview of change from parent on the left and a preview of change from our branch on the right.
  - User chooses which variant he wants to use by clicking on buttons below the previews.
  - If user selects a variant which contains Edit action, Editor will show form with prefilled values from this event. User can edit values in the form or he can let the prefilled values in form as they are.
  - If the selected variant contains delete event, no editor will be displayed.
- **User Actions:**
  - User chooses one of the variants and if variant contains edit action he can edit the values
- **Solution:**
  - If user chooses a variant from parent branch and does not do any change in the form, the event from his branch is removed.
  - When user chooses a variant with the event from his branch and does not do any change in the form, the event from his branch is kept in and applied.
  - When user uses the editor, the new event (with new `UUID`) is created and this event replaces the event from the branch.

3. **Diff Tree**
   - **Precondition:**
     - We have a delete event of some node in our branch which is marked as edited in the Diff Tree.
   - **View:**
     - User can see a diff comparing the old and the new node.
   - **User Actions:**
     - User can choose to keep the new version of entity or delete the entity.
   - **Solution:**

    - If the user decides to keep the new version of entity, the original delete event is removed.
    - Otherwise the delete event is kept in the localization.

4. **Pool**
    - **Precondition:**
        - Parent node could not be found in the knowledge model.
        - Event is either of type `Add` or `Edit`.
    - **View:**
        - User is informed that the node could not be added or edited because there is no longer the parent.
    - **User Actions:**
        - User can save the entity to a pool of entities and then move it to a new parent (entity will not be lost).
        - Or user can throw the entity away.
    - **Solution:**
        - If the user decided to delete the entity, the server will delete all the following events connected to this entity.
        - Otherwise:
            * Server checks if the entity is in the pool, if not it picks the entity from the old knowledge model and puts it into the pool.
            * All related events will be also moved to the pool.

Summary

As I said in the beginning this process can be compared to a rebase in Git. The migration process tries to apply events from our branch on top of the events from parent (even though our events can be older then some from parent). Advantages are that conflicts from parent are solved in parent and we can manipulate with our history.

But there is one strong disadvantage which I have not mentioned yet. If we have packages as they are here (package with version 2.0.0 contains also events from version 1.0.0 – see picture 2.8), there is a problem how to distinguish new events in the newer version of the package. This can cause problems in the migration process where we want to build a list of events which are new in the version. This could be solved with some mark or by separating events in the package to groups by versions. But because we can change a history, some events do not have to be here. So the built list of events does not have to be relevant.

2.4.3.2 Second Approach

In second approach I tried to avoid a manipulation with a history at all, because as we saw it caused big problems. This approach can be compared to a classic merge process in Git.

Imagine situation: we have a branch and we want to apply new changes from some other branch (let's say from a parent branch). So we will perform a classic Git merge which takes new changes from parent branch and apply them to the head of our current branch. Finally we do a merge commit (in our terminology we publish a new version) to save these changes.

Change package structure

First thing which I changed was the package structure. One package now contains just events from its version (no events from previous versions). So the problem from the first approach will not happen again. The thing, why it is possible to do it like that here, is that the history will not be edited so we can let old versions of packages as they are.

Description of how to merge events

Because I decided to add events from new versions on top of the branch, the conflicts can appear only on top of the branch where they can be solved.

Now I would like to describe the workflow of this approach and demonstrate it on an example. For better understanding I prepared an image where we can see the situation graphically inscribed (2.10).

1. **Create branch** – If we want to start editing knowledge model, we have to create a branch at first. The branch can refer to some parent package or not. In example we can see 3 branches. One without parent (orange `Core`) and two with parents (green `Localization` and purple `My Branch`). We can of course add events and then create versions on our branches.
2. **Deduction of branch properties** For better understanding I put three light green marks in circle into a picture (2.10) – `A`, `B` and `C` which refers to three states of branches. Here is the quick description of each state.
   - **Mark `A`**:

     Unversioned events:
     - No unversioned events

     Parent package:
     - `localization:2.0.0`

     Merge properties:

  - `lastAppliedParentPackageId`: `core:1.0.0`
  - `lastMergeCheckpointPackageId`: `core:1.0.0`

  List of parent packages:

  - `localization:2.0.0 -> localization:1.0.0 -> core:1.0.0`

  List of events for application:

  - `9, 8, 7, 3, 2, 1`

- **Mark `B`**:

  Unversioned events:

  - No unversioned events

  Parent package:

  - `localization:4.0.0`

  Merge properties:

  - `lastAppliedParentPackageId`: `core:2.0.0`
  - `lastMergeCheckpointPackageId`: `localization:3.0.0`

  List of parent packages:

  - `localization:4.0.0 -> localization:3.0.0 -> localization:2.0.0 -> localization:1.0.0 -> core:1.0.0`

  List of events for application:

  - `6, 12, 4, 12, 11, 10, 9, 8, 7, 3, 2, 1`

- **Mark `C`**:

  Unversioned events:

  - `16, 15, 14`

  Parent package:

  - `localization:4.0.0`

  Merge properties:

  - `lastAppliedParentPackageId`: `core:2.0.0`
  - `lastMergeCheckpointPackageId`: `localization:3.0.0`

  List of parent packages:

  - `localization:2.0.0 -> localization:1.0.0 -> core:1.0.0`

  List of events for application:

  - `16, 15, 14, 6, 12, 4, 12, 11, 10, 9, 8, 7, 3, 2, 1`

3. **Merge branch** – We can upgrade our branch just if we have no unversioned events in our branch. Upgrade technically means to merge a new version of the parent branch into the current branch. Here I would like to describe an upgrade process on branch `localization`.

   What we will migrate:

- Branch `localization` in version `3.0.0` (In picture (2.10) I marked the state as blue `M` in square)

Migration Info:

- **From version:** `core:1.0.0`
- **To version:** `core:2.0.0`

Merge properties before merge:

- `lastAppliedParentPackageId`: `core:1.0.0`
- `lastMergeCheckpointPackageId`: `core:1.0.0`

List of events from `core` *which needs to be applied (in picture (2.10) they are displayed with blue label):*

- `4,5,6`

Result:

- We can see that 2 events were applied as they came from `core` (`4` and `6`). And one was applied but changed (`5`). We can see a new event unique identificator (`uuid`) for this event (`13`) in `localization` branch.

Merge properties after merge:

- `lastAppliedParentPackageId`: `core:2.0.0`
- `lastMergeCheckpointPackageId`: `localization:4.0.0`

Conflict solving strategies

Solving conflicts in this approach is less difficult than in the first approach. I identified 2 conflict solving strategies for situations when a conflict can appear.

I also prepared the same table as in the first approach (2.2). Comparing to the first approach where a solving event was from the current branch, here is the solving event from the parent branch. The rest remains same. For a definition of a relation and for more information about the table (how to read it), please look at the first approach section (2.4.3.1).

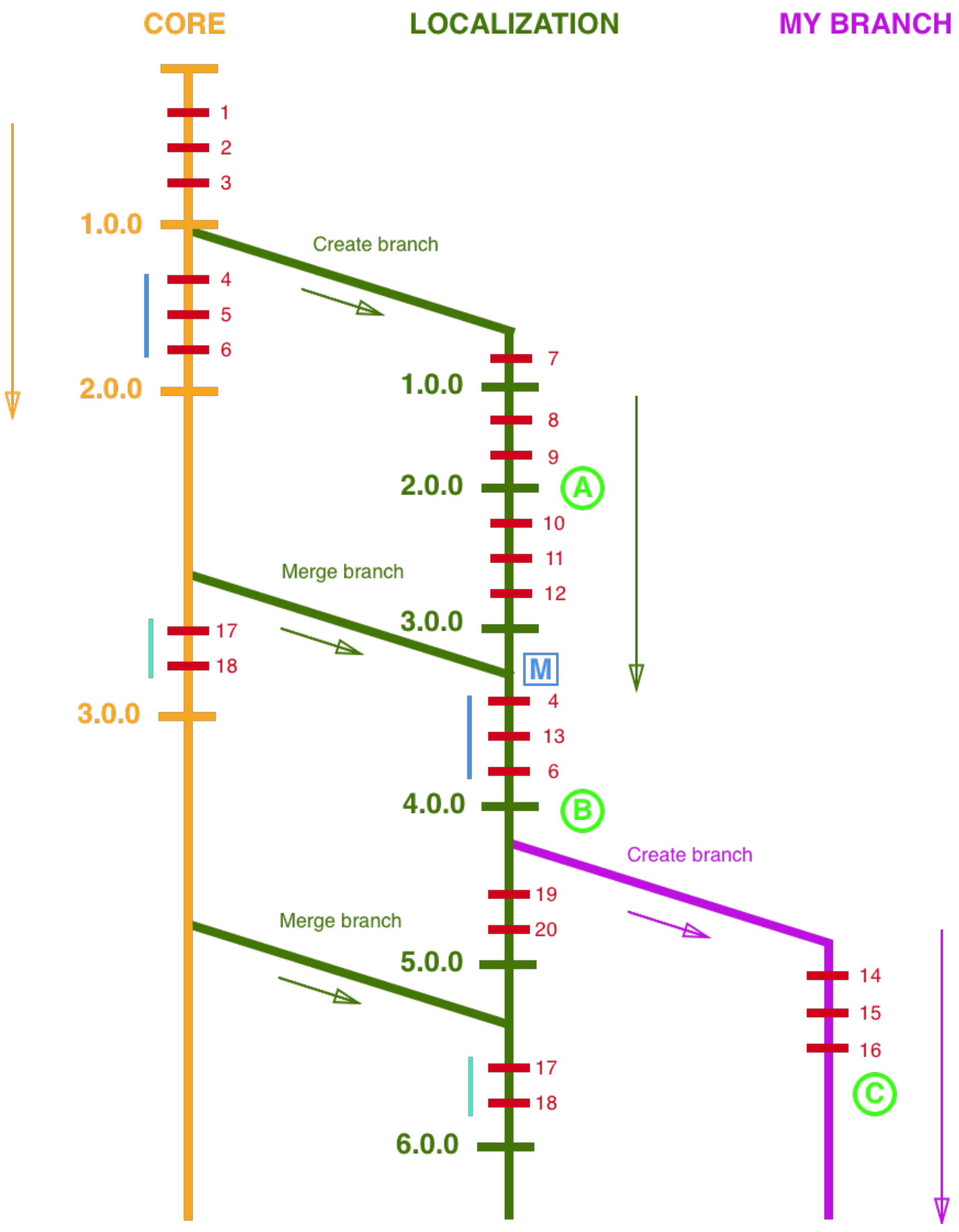

Figure 2.10: Merge process

Table 2.2: Migration strategy: Second approach

Parent Branch	Relation	Current Branch	Strategy
add	$<$	edit	Corrector
edit	$<$	edit	Corrector
delete	$<$	edit	Corrector
edit	$=$	edit	Corrector
delete	$=$	edit	Corrector
add	$=$	add	Corrector
edit	$>$	add	Corrector
edit	$>$	edit	Corrector
edit	$>$	delete	Corrector
delete	$>$	delete	Corrector
delete	$>$	add	Corrector
delete	$>$	edit	Corrector
delete	$<$	delete	Cleaner
add	$<$	delete	Cleaner
edit	$<$	delete	Cleaner
delete	$=$	delete	Cleaner
edit	$=$	delete	Cleaner
add	$>$	add	-
add	$>$	edit	-
add	$>$	delete	-
add	$=$	edit	-
add	$=$	delete	-
add	$<$	add	-
edit	$=$	add	-
edit	$<$	add	-
delete	$=$	add	-
delete	$<$	add	-

Unlike previous case strategies `Corrector` and `Cleaner` do not require any additional special precomputed structures like `DiffTree`, `DiffTable` or `Pool`.

Here are descriptions of the concrete methods.

1. **Cleaner**
    - **Precondition:**
        - Matched if the situation corresponds to one of the five from table (2.2). Basically it checks for each event if the current node or one of the parent nodes were not deleted.
    - **View:**
        - Nothing to show.
    - **User Actions:**
        - No user actions.
    - **Solution:**
        - Server will remove the event.
2. **Corrector**
    - **Precondition:**
        - Matched if the cleaner method did not match.
    - **View:**
        - User can accept or refuse the event. If he decides to accept the event, the client will show a form where user can edit properties of the event.
    - **User Actions:**
        - User can refuse the event.
        - User can accept the event as it is.
        - User can accept and edit the event.
    - **Solution:**
        - If user refused the event, the event is deleted and nothing will be applied.
        - If user accepted the event without any change, the event is normally applied.
        - If user edited the event, new `uuid` is generated for the event and then the event is applied to a knowledge model.

Summary

As I said in the beginning, this process can be compared to a merge in Git. Migration process tries to apply events from the parent branch on top of the events from our branch. An advantage is that the conflicts are solved on top of the branch which means that when we have to edit the event, because a conflict appears, we do not have to manipulate with the history.

The disadvantage is that the history of the branch is not as clear as in the first approach. If we want to upgrade our branch we need to physically add the events from the parent branch. And in the end of the migration process we need to publish these new added events in a new version. So when we look into a history, it is harder to recognize if the event comes from the parent branch or if it was originally created in our branch.

2.4.3.3 Comparison of the approaches

Both approaches are relevant and can be realized. But if we compare a complexity of solutions, a difficulty of an implementation and their advantages and disadvantages, at least for me, the second approach seems to me better than the first one. I find the second approach more elegant and because the strategies are simpler there will be less space where bugs can appear.

2.4.4 Sanitizator

Because I decided to choose the second apporach in previous section (2.4.3.3) all further ideas and suggestions will refer to this approach.

Next to `Applicator` and `Migrator` component I also created `Sanitizator` component. This component should cope with minor discrepancies which could appear in events application in migration process.

It is likely to grow in the future as improvements and smart things will be added. Currently there is just one. It corrects discrepancies in the list of children in edit events.

2.4.4.1 Correction of discrepancies in the list of children nodes

Imagine that some knowledge model looks like this:

```
Knowledge Model (km1)
\- Chapter (chapter1)
   \- Question (question1)
      |- Answer (answer1)
      |- Answer (answer2)
      \- Answer (answer3)
```

And we have this list of events to apply:

```
-- 1. Delete some child of a node
DeleteAnswerEvent (..., qUuid=question1.uuid, ...,
   qAnswerUuid=answer2.uuid)

-- 2. Change order of children in the node where we
   previously deleted  a child
EditQuestionEvent (..., qUuid=question1.uuid, ...,
   qAnswers= [answer3.uuid, answer2.uuid, answer1.uuid])
```

If we would just let these events as they are, it would display to user that we would like to change an order of answers to this order – `answer3`, `answer2` and `answer1`. But `answer2` does not exist anymore. So it does not make sense to display to user deleted answer on the second place.

We may think how this answer could even appear in the event. Simply, because during the creation of event, the knowledge model could look like different so the answer could exist there. Therefore a work of `Sanitizator` is to go though all events, inspect them and if there is an event like this, `Sanitizator` will correct it.

The correction process is applied to all edit events and it consists of these steps:

1. Append all children `uuids` from the current knowledge model to the end of a list of children `uuids` in an event
2. Go though the list of children and if some `uuid` appears for second time, remove it (remove duplicities in the list)
3. Remove all children's `uuids` which are not currently presented in the knowledge model
4. Change `uuid` of the event

We can see that the process can be applied to all edit events even if there is nothing wrong. Only side-effect of this process is that the `uuid` of an edit event will be changed in every migration (even if it does not have to). But the big advantage is that the process is simple and deterministic.

2.4.5 Summary

The final migration tool consists of three parts – `Applicator`, `Migrator` and `Sanitizator`.

- `Applicator` is used for an event application on a knowledge model and is used even outside the migration tool.

- `Migrator` applies new events from a parent branch to a knowledge model in the current branch. I chose finally the second approach instead of the first one (for more information see: 2.4.3.3)

- `Sanitizator` is used to correct events which will be applied to a knowledge model in the migration process

Chapter 3

Implementation

3.1 Introduction

In previous chapters I analyzed requirements, designed a solution and now I would like to devote my attention to an implementation. I start with an introduction of how the project is structured (3.2). Then I talk about build tools which I used and their configurations (3.3).

Further I describe an architecture of the application and its individual layers (section 3.4, 3.5 and 3.6). Separate sections I devoted to an error handling (3.7) same as for a more detailed description of the migration tool (3.8).

3.2 Project Structure

Mostly I kept a recommendation of `Stack` build tool and I divided the project into these base folders:

```
<root>
|- app
|- config
|- lib
|- scripts
\- test
```

- In the root directory (**`<root>`**) we can find a license and configurations for build tools.
- The subdirectory **`app`** contains just one file which is a main entry-point to the application.

- The subdirectory **`config`** contains a configuration and a build information for the application.
- The subdirectory **`lib`** contains a whole application logic
- The subdirectory **`test`** includes expected tests.

3.3 Build Tools Setup

3.3.1 Stack

As I described in the Selected Tools section (1.4.5) I used `Stack`[18] as a main build tool. When we use `Stack` we have to decide which LTS (Long Term Support) we want to use. LTS is a set of packages (the packages are fixed to concrete versions) which guarantees a build consistency, passing tests and a good package inter-cooperation.

During the development I used more LTSs and I finally ended with version `9.11`[33] (published in October 30, 2017).

3.3.2 Hpack

Information about our package is written in Cabal file (`<project-name>.cabal`). It includes base information about the package like a name, an author, which `GHC` the package was tested with, a license, etc. It also includes a list of project dependencies, a description of modules, an information about used Haskell language extensions, etc.

We can see there is a plenty of information. But actually the Cabal file has its own format and many things are duplicated there. This problem is solved by `Hpack`[34] which is a tool for generating Cabal file from a template. `Hpack` has much more shorter notation (in my project Cabal file has about 300 lines compares to 130 lines of `Hpack` file) which is caused by a reduction of duplicities.

3.4 Application structure

I divided the application into 3 layers[35]:

1. **Presentation layer** – REST API (`Api` package)
2. **Domain layer** – Services, Business logic (`Service` package)
3. **Data access layer** – Persistence to MongoDB database (`Database` package)

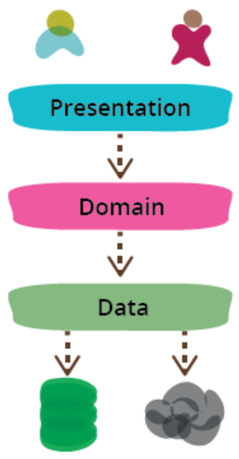

Figure 3.1: Application layers (according to Martin Fowler[35])

3.4.1 Presentation Layer

Presentation layer is represented here by REST API which provides application functionalities and presents data to the outside world. Code is stored inside `Api` package. This package contains 3 additional subpackages – `Handler`, `Middleware` and `Resource`. Further there is a file `Routes.hs` which defined routes mapping, a usage of a middleware and a base security setup (a definition of secured and unsecured routes).

In `Handler` package we can find functions which process incoming requests and pass the requests into the service layer. These functions are packed into files according to their domains, e.g. `Organization`, `User` or `KnowledgeModel` modules.

In `Middlerware` package we can find 2 Haskell modules – `Auth` and `CORS` modules. `CORS` (Cross-origin resource sharing) module has a definition of middleware which takes care about adding `CORS` headers to every outcoming response. And a purpose of `Auth` module is to take care about the security. It checks a presence and a validity of JWT token for the secured routes.

In `Resource` package there are placed structures on which is mapped incoming requests and outcoming reponses. In modules we can find next to a definition of structures also mappings of structures from and to JSON format. Structures and their mappings are packed to files according to their domains (same as with `Handler` functions).

3.4.2 Domain Layer

Domain layer includes most of the business logic of the application. Code is stored inside the `Service` package and then it is again divided into packages according to a domain (`Organization`, `User` or `KnowledgeModel`). Most of the domain package has one module for business functions and one module for mapping functions between business structures and structures which are used in `API`.

Business structures are used widely across a whole application so they are stored in a root package `Model`. They encapsulate objects which have been identified in domain model. In some modules there are additional functions to make a work with these structures simpler. For example in `KnowledgeModel` module we can find a utility function which takes a chapter and returns `uuids` of children's questions. Or we can find here a function which takes a knowledge model with question's `uuid` and returns a question with given `uuid` from the knowledge model (if exists).

Service functions are mostly functions which call functions from data access layer and put the result together. Between service functions there are also functions for validation structures. And because domain layer should contains business logic, I put there also the migration tool.

3.4.3 Data Access Layer

Last layer is a data access layer. This layer is divided into 3 main packages – `BSON`, `DAO` and `Migration`. Together with these 3 packages there is a module `Connection`. This module is responsible for creating a connection pool to MongoDB database.

`BSON` package contains mappings of business structures to and from `BSON` database format. Structures and their mappings are packed to files according to their domains (same as with `JSON` mappings in `Resource` package).

`DAO` package contains functions which translate our requests to database queries or manipulate with data in the database.

And the last package `Migration` contains some base initial migrations to load some dummy data into the database (currently it is used for testing and demo purposes).

3.4.4 Module naming convention

3.4.4.1 Choosing convention

To make an application more readable for new people on the project I decided to keep some naming conventions from widely well-known object programming. I did it because current naming conventions in functional programming are not well develop in this particular area so I mostly create my conventions which are inspired in object-oriented world. So it is a nonsense to think about the fact what the suffix of acronym or just the suffix means because they do not make sense in functional programming.

3.4.4.2 Haskell Modules

I decided to distinguish modules by adding suffix to the end of the name. So to all modules which include service functions I added a suffix `Service` or to all modules which include handler functions I added a suffix `Handler`.

Here is a complete list of these suffixes:

- **Handler** – a module containing handler functions
- **DTO** – a module containing structures which represents request/response in API
- **Middleware** – a module containing middleware functions
- **Service** – a module containing service functions
- **Mapper** – a module containing mapper functions
- **DAO** – a module containing functions for a manipulation with data in database
- **Migration** – a module containing functions for running initial database migrations

3.4.4.3 Structure properties

Because Haskell does not allow function overloading by default, it is necessary to prefix properties of structures. Otherwise we may get a conflict if two different structures would have a property with the same name.

Listing 3.1 : Example of prefixed properties

```haskell
data Organization = Organization
  { _orgUuid :: UUID
  , _orgName :: String
  , _orgGroupId :: String
  }

data OrganizationDTO = OrganizationDTO
  { _orgdtoUuid :: UUID
  , _orgdtoName :: String
  , _orgdtoGroupId :: String
  }
```

3.5 API

In this section I would like to describe how I designed the API and how a security and an error handling issues are solved.

3.5.1 Authentication and Authorization

Most of the endpoints are secured (see API specification – attachment A). For accessing these secured routes the application requires a usage of a valid token. This token can be obtained in `/tokens` endpoint against valid credentials. Public endpoints can be used without token. As I mentioned in Selected Tools section (1.4), I used JSON Web Tokens (JWT) format for tokens. Because a payload in tokens can change, the tokens are not stored in a database or cached somewhere. They are generated after every successful login.

Token payload currently contains an identification of a user (`userUuid`) and a list of permissions which belongs to the user.

Application requires to include the token in `Authorization` header. Here is an example of the token (with used secret: `secret-key`).

Listing 3.2 : Example of authorization header

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVC
J9.eyJ1c2VyVXVpZCI6ImVjNmY4ZTkwLTJhOTEtNDllYy1hYTNmLTllY
WIyMjY3ZmM2NiIsInBlcm1pc3Npb25zIjpbIlVNX1BFUk0iLCJPUkdfU
EVSTSIsIktNX1BFUk0iLCJLTV9VUEdSQURFX1BFUk0iLCJLTV9QVUJMS
VNIX1BFUk0iLCJQTV9QRVJNIiwiV0laX1BFUk0iLCJETVBfUEVSTSJdf
Q.BFBXG8gjJeqt3i-hKzsp10_ePM5st34vuJqiYeNwyu4
```

3.5.2 Error Handling

When an error occurs I serialized it to a JSON format and set a right status code to a response which at most corresponds to REST architecture.

Here is a list of error responses which can be returned:

- **400 Bad Request** – It is returned when:
  - Request body was not correctly parsed to a Haskell structure (a response body includes just status, error and message properties),
  - Some validation failed (a response body includes status, error, message, formErrors and fieldErrors properties),
  - Problem in an event application in Applicator (a response body includes just status, error and message properties).

Listing 3.3 : Example response body

```
{
  "status": 400,
  "error": "Bad Request",
  "message": "",
  "formErrors": [],
  "fieldErrors": {
    "parentPackageId": "Parent package doesn't
       exist"
  }
}
```

- **401 Unauthorized** – it is returned when a login failed or we try to access secured routes without a proper authorization

Listing 3.4 : Example response body

```
{
  "status": 401,
  "error": "Unauthorized"
}
```

- **403 Forbidden** – It is returned when we try to access resources which we do not have rights (permissions) to

Listing 3.5 : Example response body

```
{
  "status": 403,
  "error": "Forbidden"
}
```

- **404 Not Found** – It is returned when a resource does not exist

Listing 3.6 : Example response body

```
{
  "status": 404,
  "error": "Not Found"
}
```

- **500 Internal Server Error** – It is returned when some server side error occurs, e.g. a problem with a connection to a database, a problem with a deserialization of an entity from a database, etc.

3.5.3 API Specification

A complete API specification is described in an attachment A).

3.6 Database

For persisting data I decided to use a document database MongoDB which I chose as the most suitable for the application. A connection to the database is created during a start of the application. Configurations are stored in the application configuration file (`app-config.cfg`).

I do no create a new connection for each request but I hold a pool of connections. The requests then take connections from this pool.

3.6.1 Advanced usage

I do not think that it is necessary to describe all queries which I did but I would like to mention a few concepts and advanced features which I used.

Embedded documents

I frequently use a principle of embedded documents. That was also one of the reason why I chose MongoDB. We can see this principle for example in storing of a knowledge model. The knowledge model contains chapters which contains questions, etc. And all these data is stored in one document. So we do not have to join more tables when we want to retrieve a knowledge model.

Partial update

Next thing which I used is a partial update. That allows us to update just one property of a document without loading the whole document into a memory. This is done by using `$set` construct. An example of usage from the application is when I update a knowledge model in a branch document.

Listing 3.7 : Example of usage from the application

```
let sQuery = (select ["uuid" =: bUuid] branchCollection)
let aQuery = ["$set" =: ["knowledgeModel" =: (km)]]
let action = modify sQuery aQuery
runMongoDBPoolDef action dbPool
```

Append to a list in a document
When we want to just add a new item into some list in a document we do not have to retrieve a whole list or even a document. We can just append our items into the current list. MongoDB has also special construct for that – `$push`. An example of usage from the application is when I add new events which came from a client into a branch.

Listing 3.8 : Example of usage from the application

```
let sQuery = (select ["uuid" =: bUuid] branchCollection)
let aQuery = ["$push" =: ["events" =: ["$each" =:
   events]]]
let action = modify sQuery aQuery
runMongoDBPoolDef action dbPool
```

3.7 Error Handling

As I mentioned in a section Error Handling in Analysis chapter (see 1.5.3) I tried to avoid unhandled error states by using `Maybe` and `Either` wrappers.

For a purpose of the application I created my own structure which should cover all errors – `AppError`. This structure has more data constructors which should cover all types of errors which can occur in the application.

Listing 3.9 : My custom error structure

```
type ErrorMessage = String

type FormError = String

type FieldError = (String, String)

data AppError
  = ValidationError ErrorMessage [FormError]
     [FieldError]
  | NotExistsError ErrorMessage
  | DatabaseError ErrorMessage
  | MigratorError ErrorMessage
```

Because almost every part of the application can fail, almost every returned type is either `Maybe AppError` or `Either AppError <Returned_Value>`. Advantage of this concept is that it allows us to handle errors by default.

I used `Maybe` for cases when I did not expect any result. `Nothing` means everything is okay and `Just` contains error if it occurs.

And I used Either if I expected a value. The value (wrapped in Right) indicates a successful returned value and a value (wrapped in Left) means that an error was returned.

3.7.0.1 Types of Errors

I distinguish 4 types of errors. First is ValidationError which can occur during the validation of incoming data. Depending on a situation it can contain some base error message, errors related to a form and also errors related to a concrete form field. A message, form errors and fields errors are then used in a client application to display what went wrong in the form.

NotExistsError is used when requested data was not found in a database. I separate this error from DatabaseError so I could easily translate this exception to 404 Not Found error on API.

DatabaseError means that something bad happened during our communication with database, e.g. a document could not be well decoded from BSON.

And the last error – MigratorError encapsulates all errors from a migration process and errors from a bad event application.

3.8 Migration Tool

A principle on which is the migration tool based I have already described in a chapter Design of The Migration Tool (see chapter 2). Same as I have already mentioned 3 components from which the migration tool is consisted of – Applicator, Migrator and Sanitizator. These components hold all migration logic. An orchestration of these components is then managed by functions in MigrationService module.

3.8.1 Applicator

3.8.1.1 Interface

Applicator has an interface which contains just one function – runApplicator function. This function takes a knowledge model if it exists and a list of events to apply. If the application ends with a success it returns a new knowledge model, otherwise it returns an error. Here is a type specification of this function:

Listing 3.10 : Type specification

```
runApplicator :: Maybe KnowledgeModel -> [Event] ->
   Either AppError KnowledgeModel
runApplicator km events = ...
```

3.8.1.2 Implementation

Function `runApplicator` goes though incoming events and each of them applies to a knowledge model (call `applyEventToKM` on the knowledge model and the event). A method `applyEventToKM` is a member of typeclass `ApplyEventToKM` and all events implement this method in their instances. So the behavior, how the event should apply to knowledge model, is defined in instances of the typeclass `ApplyEventToKM`. Here is a definition of the typeclass:

Listing 3.11 : Definition of ApplyEventToKM

```
class ApplyEventToKM e where
  applyEventToKM
      :: e
      -> Either AppError (Maybe KnowledgeModel)
      -> Either AppError (Maybe KnowledgeModel)
```

Due to implementations of this method we can change just knowledge model. So if an event wants to edit for example a chapter, the implementation of this method contains just a delegation to knowledge model chapters. The delegation is done by calling `applyEventToChapter` on all chapters which is a method of a typeclass ApplyEventToChapter. So all events also implement this typeclass.

Implementation of `applyEventToChapter` are done in the same way as implementations of `applyEventToKM` were. Every event has an instance of this typeclass.

For every type of node (a knowledge model, a chapter, a question, an answer, etc.) I created a typeclass with a method how to apply an event to this specific type of node. Their instances then know how the event should be applied to this type of node.

3.8.2 Migrator

An interface of Migrator component consists of 2 functions – `migrate` and `solveConflict`.

3.8.2.1 Function `migrate`

The first function `migrate` takes a migration state and produces a new migration state which is wrapped in `IO`. It does not have to have additional parameters because the migration state contains all necessary things which are needed for a migration (see the domain model for more information – 1.3.4.3).

Function `migrate` takes an event from the migration state and tries to apply this event with one of the methods (`Cleaner` or `Corrector`) to the knowledge model. Then it calls recursively itself until it ends in some other state than in `RunningState`.

The function needs to return a result wrapped in `IO` because sometimes the logic may need to generate a new `uuid` which is an impure action because it communicates with the outside world.

3.8.2.2 Function `solveConflict`

Function `solveConflict` takes a migration state and an action how to resolve the conflict. And it produces a new migration state where the conflict is solved.

3.8.3 Sanitizator

`Sanitizator` edits event properties (if necessary) to be in an actual state against the current knowledge model. For more information, what `Sanitizator` does, see section Sanitizator in Design chapter (2.4.4).

Interface of `Sanitizator` component is very simple and it includes one typeclass `Sanitizator`. `Sanitizator` has currently instances just for editing events because there is no need for a sanitization of add or delete events.

Listing 3.12 : Definition of Sanitizator

```
class Sanitizator a where
  sanitize :: MigratorState -> a -> IO a
```

3.8.4 Migration Service

Functions in `MigrationService` module encapsulate a logic of `Applicator`, `Migrator` and `Sanitizator`. It also prepares a migration state, validates inputs and persists the current migration state to a database.

Chapter 4

Configuration, Testing and Deployment

4.1 Introduction

In the last chapter I would like to talk about 3 last things. First I would like to mention how the application can be configured (4.2). Then I would like to sum up how I tested the application (4.3). In this section I describe types of tests and also I slightly touch a code coverage theme. In the last section I talk about a production deployment (4.4), how I set up a server provided by my faculty and how I deployed the application into it.

4.2 Configuration

Especially for a future development I prepare a configuration through a configuration file. Because Haskell/Scotty does not provided any prepared mechanism for it I had to implement it by my own. Currently there are 2 files – `app-config.cfg` and `build-info.cfg` which contain all information needed by a running application. For tests there exist special versions suffixed by `-test`. A format of configuration files is an old-style `Windows .INI` format.

An advantage of an external configuration is that the configuration does not have to be present during a compilation process and it can be attached later (during a start of the application).

4.2.1 Build Configuration

A build configuration (`build-info.cfg` file) contains information about the last build of the application. This configuration can be created simply by running a prepared script `build-info.sh`. Normally this script is started by a CI Tool (Travis CI) during build process.

Listing 4.1 : Example of build-info.cfg

```
name = Data Stewardship Portal Server
version = 1.0.0
builtat = 2017/10/25 19:50:20Z
```

4.2.2 Application Configuration

Application Configuration (`app-config.cfg` file) contains 4 sections – `Web`, `Database`, `JWT` and `Role`. Currently there are not many things which can be configured but it is assumed that it will grow in the future.

In `Web` section we can configure on which port the server will be running. In `Database` section we can set up connection properties to our database. `JWT` section contains property which holds a value of secret which is needed in a process of signing JWT payload. And in `Role` section we can assign permissions to roles. These sets of permissions will be used as templates for new users in a way that when a new user is created with some role he will get assigned a list of permissions based on this template from the configuration file.

Listing 4.2 : Example of app-config.cfg

```
[Web]
port = 3000

[Database]
host = mongo
dbname = dsp-server
port = 27017

[JWT]
secret = secret-key

[Role]
admin = UM_PERM, ORG_PERM, KM_PERM, KM_UPGRADE_PERM,
   KM_PUBLISH_PERM, PM_PERM, WIZ_PERM, DMP_PERM
datasteward = KM_PERM, KM_UPGRADE_PERM,
   KM_PUBLISH_PERM, WIZ_PERM, DMP_PERM
researcher = WIZ_PERM, DMP_PERM
```

4.3 Testing

The application contains about 240 tests. My goal was to test as much as it was possible by automatic tests. One thing which had to be tested manually was an integration with a client application. But all other parts could be tested automatically.

Martin Fowler says that we should have more low-level tests (unit tests) than high-level tests [36]. Because a price of creation and maintenance of unit tests is much more lower than a price of more high-level tests. And of course a simple unit test can be written much more faster.

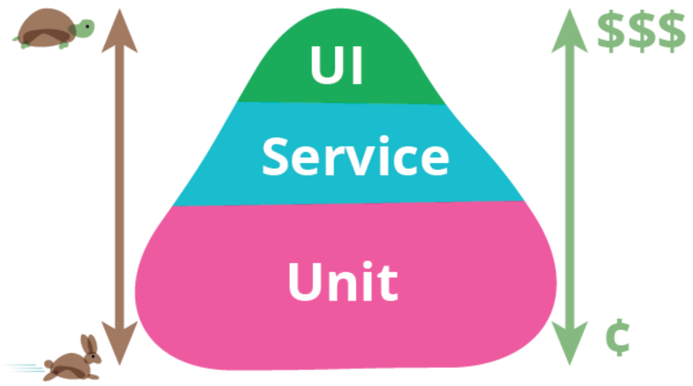

Figure 4.1: Test pyramid (according to Martin Fowler[36])

But I finally decided not to follow this recommendation so strictly. I ended with a ratio 92 unit tests (27 percent) against 148 integration tests (73 percent). Here are reasons why I ended in such a state.

- My only integration is with MongoDB database. So I do not have to set up additional systems to be able to run integration tests. So a creation of an integration test is almost as simple as a creation of an unit test.
- Service layer mostly does just a delegation to database layer which does a selection, an insertion, an update or a delete to or from a database. So it is a nonsense to mock these database operations and tests just the service logic because we would test nothing (just the delegation).

4.3.1 Unit tests

I used unit tests mainly for a testing of these things:

- A functionality of `Applicator`,
- A functionality of `Migrator`,
- A functionality of `Sanitizator`,

- Query methods for knowledge model,
- A validation logic,
- Utility functions.

Functions in these parts do not require a communication with outside world so they are perfectly suited for unit tests. They also include a complex logic, they could end in many states with different results and they can have various inputs. So it made sense to deeply test these functions and be sure that they behave as I wanted them to behave.

4.3.2 Integration tests

In integration tests I mainly tested if a HTTP request is correctly translated to a database query. Without one exception, integration tests contain just API tests.

I tested all API endpoints except `IO` endpoints (`/import` and `/export`). Each test on endpoint includes several sub-tests which covers all possible cases which could appear during a call of this endpoint.

Here is an example of cases which are tested on `POST /users` endpoint (this endpoint serves for a creation of a user).

- **Case 1: `HTTP 201 CREATED`** – I can successfully create a user.
- **Case 2: `HTTP 400 BAD REQUEST when JSON is not valid`** – When I provide invalid `JSON` I will get a response with a status code `400 Bad Request`.
- **Case 3: `HTTP 400 BAD REQUEST if email is already registered`** – When an email is already registered I will get a response with a status code `400 Bad Request` .
- **Case 4: `HTTP 401 UNAUTHORIZED`** – When I do not provide a valid token I will get a response with a status code `401 Unauthorized`.
- **Case 5: `HTTP 403 FORBIDDEN`** – When I do not have a requested permission I will get a response with a status code `403 Forbidden`.

We can see that the cases 2, 4 and 5 are more general and also other endpoints will need to test them as well. Therefore I created a template from which I can generate these test cases. So I do not have to duplicate code.

Test scenario

A normal scenario of one integration API test is:

1. Prepare a request
2. Optionally prepare a database
3. Call a server with the prepared request
4. Check a response if it has an expected body and headers
5. Optionally check a state of the database

4.3.3 Code Coverage

In the end I want to show a code coverage of my code. But because an integration of `Stack` tool with `hpc` tool (code coverage library for Haskell) is not as far as I expected, I was able to just generate code coverage results from all codebase. So the result includes for example structures from API and code which should not be included in.

Listing 4.3 : Code coverage results

```
52% expressions used (11165/21153)
80% boolean coverage (42/52)
50% guards (1/2), 1 always True
82% 'if' conditions (41/50), 5 always True, 2 always
   False, 2 unevaluated
100% qualifiers (0/0)
50% alternatives used (476/941)
88% local declarations used (321/362)
41% top-level declarations used (824/1966)
```

The most important things according to me are percentages of tested expressions and alternatives. We can see 52% of expressions are tested with 50% of alternatives.

But these numbers are not really relevant. So I manually extracted an information about each module (`hpc`[37] tool can generate this data) and counted more relevant data. Fo the result I create a table (4.1). In my option the code coverage is satisfactory.

Table 4.1: Code coverage by layers

Layer	Expressions	Alternatives
Presentation Layer	84%	76%
Domain Layer	76%	66%
Data Access Layer	92%	33%

4.4 Production Deployment

The application is distributed in a form of a Docker image. So everyone with installed Docker can run the application.

4.4.1 Building Image

I create my own Docker image which encapsulate the application (`dsp-server`). As a template I used my another image (`stack-hpack`). This image contains preinstalled `Stack` and `Hpack`. My application image (`dsp-server`) then contains source codes and compiled application. An advantage of having one application image is that we do not have to have 2 images – one for building and one for running a compiled application. A disadvantage is a size of Docker image (it has to include source codes and build tools).

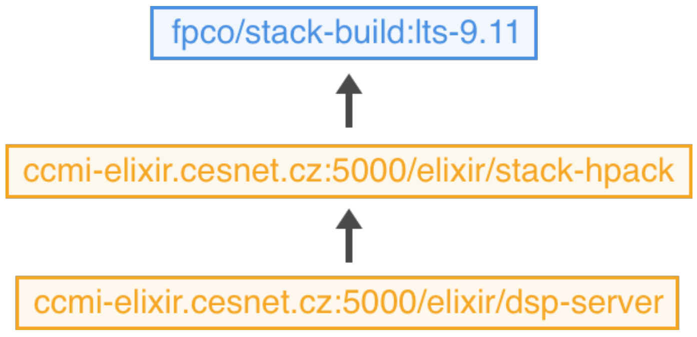
fpco/stack-build:lts-9.11
ccmi-elixir.cesnet.cz:5000/elixir/stack-hpack
ccmi-elixir.cesnet.cz:5000/elixir/dsp-server

Figure 4.2: Docker images

As I mentioned in Continuous Integration section (1.4.8) I build a Docker image automatically after a detected change in a master branch. A produced image is then uploaded to Docker Registry.

4.4.2 Running Containers

On server I set up with Jan Slifka Docker Compose[38] which is a tool to simplify a deployment process of more Docker Containers. A principle of Docker Compose is to write one file where you describe which containers you want to start, how they are connected, which files you want to mount in, etc.

Here is an example of a configuration for Docker Compose.

Listing 4.4 : Code coverage results

```
version: '3'
services:

  mongo:
    image: mongo
    ports:
    - 27017:27017

  dsp_server:
    image: ccmi-elixir.cesnet.cz:5000/elixir/dsp-server
    ports:
      - 3000:443
    links:
      - mongo

  dsp_client:
    image: ccmi-elixir.cesnet.cz:5000/elixir/dsp-client
    environment:
      - API_URL=https://api.dsp.fairdata.solutions

  dsp_nginx:
    image: nginx
    ports:
      - 80:80
      - 443:443
    volumes:
      - ....
    links:
      - dsp_client
      - dsp_server
```

We can see from the example that I have to deploy also Nginx. Its role here is to serve as a reverse proxy and holds certificate for a client and a server so they do not have to manage certificates by themselves and offer secure connection.

For better understanding how the containers are connected now, I created a more low-level deployment diagram (4.3). Compares to one from Analysis chapter (1.9) which is more high-level and where I wanted to show a whole platform, here, the main focus is on how client and server are deployed using Docker.

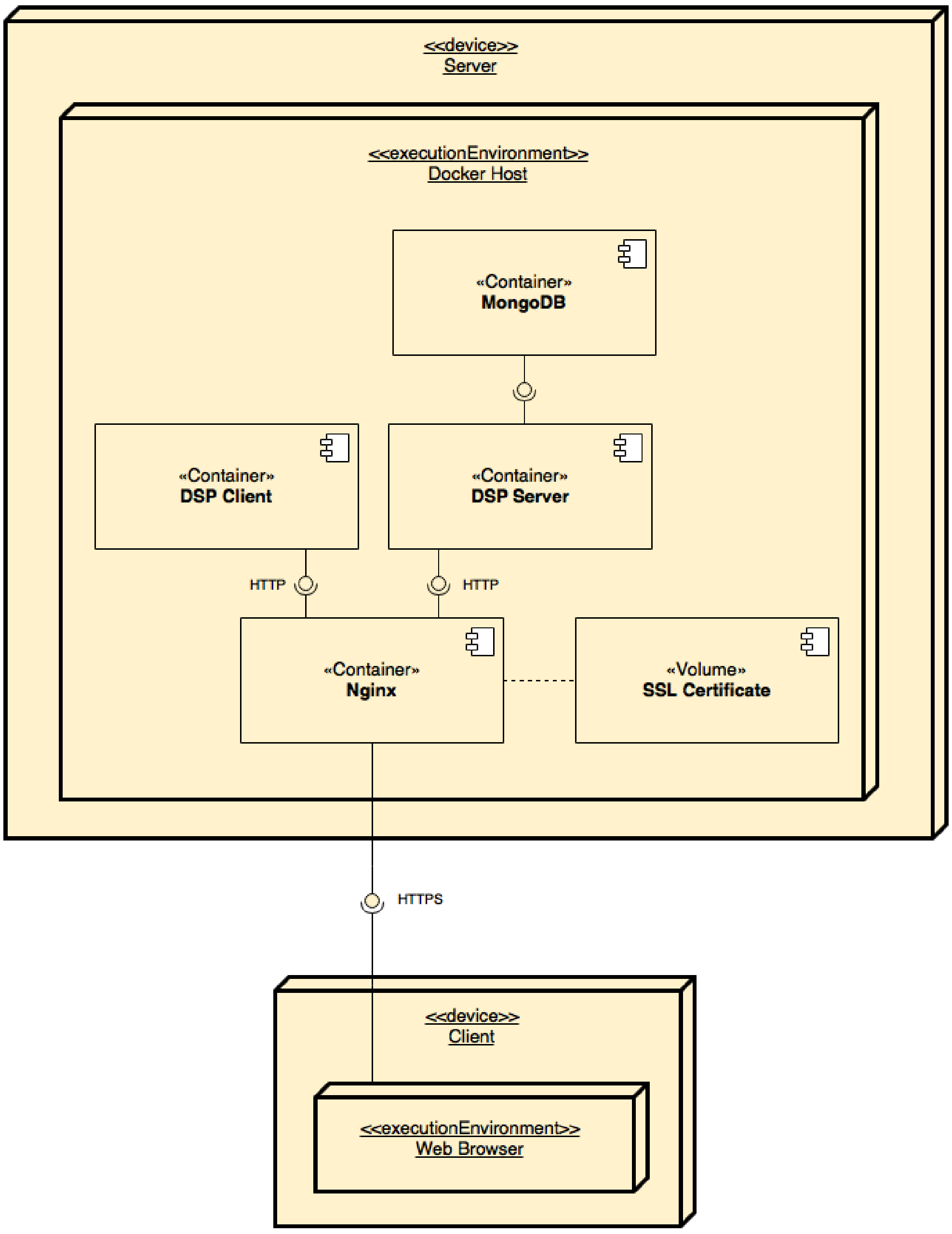

Figure 4.3: Low-Level Deplyment Diagram (Production Server)

The application is currently (during writing a diploma thesis) deployed on a server provided by my university. Here are the addresses of running applications:

- **Server:** https://api.dsp.fairdata.solutions

- **Client:** https://dsp.fairdata.solutions

Conclusion

Goals Assessment

The result of the thesis is a fully-working migration tool which fulfills all specified requirements. The tool was implemented in Haskell programming language as it was requested.

During the development it was revealed that for a proper implementation it is necessary to extend the goal of the diploma thesis (the migration tool). Together with the migration tool, there were implemented bases of the future Data Stewardship Planning Portal. Currently the portal contains a user management module, a package management module, an editor of knowledge models and the migration tool. The application was implemented purely as a server side and it offers a REST API to outside world. Simultaneously Jan Slifka from FIT CTU developed in his diploma thesis a client application which used the newly offered API. So the result is a nice web application which is ready to use and which contains base modules for the future portal application.

The application is well tested by many automatic unit and integrations tests. The documentation includes a README which consists of information about how to run, how to configure and how to deploy the application. API and functionalities of the application are documented in extensive tests where developers can find an expressive scenarios. Further API Specification is attached to this diploma thesis.

To summarize all specified goals were successfully fulfilled.

Project Future

The project is currently in a very good state and it is assumed that it will be further developed in the future.

New Features

To make a portal application complete, there are 2 things which need to be done:

1. **An integration of Wizard application into the portal** – Currently the Wizard application is not connected to the portal application so it can not use generated knowledge models. So there is a need to integrate the Wizard application through the offered REST API to the portal.

2. **An implementation of a module for generating Data Management Plans** – One of the future goals of the portal application is to offer an ability to automatically generate data management plans. This feature should help researchers to save some time because they will not have to write the plan manually.

First thing on the list is The feature will enable to researchers fill the generated knowledge model with their answers.

So the researchers will be able to work with a generated knowledge model in the wizard.

Application Improvements

1. **Fully Automatic Deployment** – Currently there is one last step which is not fully automatic in the deployment process. The application is not redeployed after a successful build. It is needed to manually sign into the server and start a deployment process of a new version of the application.

2. **Tests in CI pipeline** – There are 2 types of automatic tests in the application - unit and integration. For the integration tests there is a need to run the MongoDB database. That is the reason why tests are not running during the build process on CI server.

3. **Consider API Documentation** – Currently the API of the server application is sufficiently documented in the attachment of this thesis. For all positive and negative scenarios there also exist automatic integration tests which show what should we send to API and what we will retrieve. The API is not currently public accessible but because in the future it could be, it should be considered to create a full-fledged API documentation.

Bibliography

[1] ELIXIR. *Elixir - What we do.* [online], [cit. 2018-01-06]. Available from: `https://www.elixir-europe.org/about-us/what-we-do`

[2] ELIXIR CZ. *About ELIXIR CZ* [online], [cit. 2018-01-06]. Available from: `https://www.elixir-czech.cz/about-elixir-cz`

[3] WILKINSON, Mark, DUMONTIER, Michel et al. *The FAIR Guiding Principles for scientific data management and stewardship.* March 2016, [online], [cit. 2018-01-06]. Available from: `http://dx.doi.org/10.1038/sdata.2016.18`

[4] MARLOW, Simon et al. *Haskell 2010 - Language Report.* 2010, [online], [cit. 2017-12-08]. Available from: `https://www.haskell.org/onlinereport/haskell2010/`

[5] OBJECT MANAGEMENT GROUP. *UML Specification.* May 2015, [online], [cit. 2017-12-06]. Available from: `http://www.omg.org/spec/UML/2.5/PDF`

[6] PYTHON SOFTWARE FOUNDATION. *Python - Official Documentation.* [software], [cit. 2017-12-08]. Available from: `https://docs.python.org/3.6/contents.html`

[7] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* Dissertation thesis, University of California, Irvine, 2000.

[8] DOCKER INC. *Docker.* [software], [cit. 2017-12-11]. Available from: `https://docs.docker.com`

[9] MONS, B. *Data Stewardship for Open Science: Implementing Fair Principles.* London, United Kingdom: Chapman and Hall/CRC, 2018, ISBN 978-1498753173.

[10] THE APACHE SOFTWARE FOUNDATION. *Maven - Official Documentation.* [software], [cit. 2017-12-18]. Available from: `https://maven.apache.org/pom.html`

[11] CZAPLICKI, Evan. *Elm.* [software], [cit. 2017-12-18]. Available from: `http://elm-lang.org/docs`

[12] MONGODB, INC. *MongoDB.* [software], [cit. 2017-12-18]. Available from: `https://docs.mongodb.com`

[13] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL.* [software], [cit. 2017-12-18]. Available from: `http://elm-lang.org/docs`

[14] FARMER, Andrew. *Scotty.* [software], [cit. 2017-12-19]. Available from: `http://hackage.haskell.org/package/scotty`

[15] SNOYMAN, M. *Developing Web Applications with Haskell and Yesod.* Sebastopol, California, USA: O'Reilly Media, 2018, ISBN 978-1449316976.

[16] SNAP CONTRIBUTORS. *Snap.* [software], [cit. 2017-12-19]. Available from: `http://hackage.haskell.org/package/snap`

[17] JONES, Isaac. *The Haskell Cabal, a common architecture for building applications and libraries.* 2004, [online], [cit. 2017-12-22]. Available from: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.9361&rep=rep1&type=pdf`

[18] STACK CONTRIBUTORS. *Stack.* [software], [cit. 2017-12-19]. Available from: `http://haskellstack.org`

[19] JONES, Mike, BRADLEY, John and SAKIMURA, Nanae. *JSON Web Token (JWT).* May 2015, [online], [cit. 2017-12-22]. Available from: `https://www.rfc-editor.org/rfc/rfc7519.txt`

[20] GITHUB, INC. *GitHub.* [software], [cit. 2017-12-19]. Available from: `http://github.com`

[21] DOCKER INC. *Docker Registry.* [software], [cit. 2017-12-22]. Available from: `https://docs.docker.com/registry/`

[22] NGINX, INC. *Nginx.* [software], [cit. 2017-12-19]. Available from: `https://www.nginx.com`

[23] LIPOVACA, M. *Learn You a Haskell for Great Good!* San Francisco, California, USA: No Starch Press, 2011, ISBN 978-1593272838.

[24] ORACLE CORPORATION. *Java.* [software], [cit. 2017-12-19]. Available from: `https://docs.oracle.com/javase/8/docs/index.html`

[25] O'SULLIVA, B. *Real World Haskell.* Sebastopol, California, USA: O'Reilly, 2008, ISBN 978-0596514983.

[26] ALLEN, C.; MORONUKI, J. *Haskell Programming from First Principles.* USA: Allen and Moronuki Publishing, 2016, ISBN 978-1945388033.

[27] ABRAHAMSON, Joseph. *A Little Lens Starter Tutorial.* 2014, [online], [cit. 2017-12-28]. Available from: `https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/a-little-lens-starter-tutorial`

[28] RHODECODE. *TestPyramid.* 2017, [online], [cit. 2018-01-05]. Available from: `https://rhodecode.com/insights/version-control-systems-2016`

[29] FOWLER, Martin. *FeatureBranch.* 2009, [online], [cit. 2018-01-05]. Available from: `https://martinfowler.com/bliki/FeatureBranch.html`

[30] CHACON, S.; STRAUB, B. *Pro Git.* New York, New York, USA: Apress, 2014, ISBN 978-1484200773.

[31] FOWLER, Martin. *Event Sourcing.* 2005, [online], [cit. 2017-12-30]. Available from: `https://martinfowler.com/eaaDev/EventSourcing.html`

[32] FOWLER, Martin. *CQRS.* 2011, [online], [cit. 2018-01-05]. Available from: `https://martinfowler.com/bliki/CQRS.html`

[33] STACK CONTRIBUTORS. *Stack LTS 9.11.* [software], [cit. 2017-12-19]. Available from: `https://www.stackage.org/lts-9.11`

[34] HENGEL, Simon. *Hpack.* [software], [cit. 2017-12-19]. Available from: `https://github.com/sol/hpack`

[35] FOWLER, Martin. *PresentationDomainDataLayering.* 2015, [online], [cit. 2017-12-31]. Available from: `https://martinfowler.com/bliki/PresentationDomainDataLayering.html`

[36] FOWLER, Martin. *TestPyramid.* 2012, [online], [cit. 2018-01-05]. Available from: `https://martinfowler.com/bliki/TestPyramid.html`

[37] GILL, Andy. *Hpc.* [software], [cit. 2017-12-22]. Available from: `https://wiki.haskell.org/Haskell_program_coverage`

[38] DOCKER INC. *Docker Compose.* [software], [cit. 2017-12-22]. Available from: `https://docs.docker.com/compose/`

Appendix **A**

API Specification

Info

Structures

- **`InfoDTO`**
  - `name` (`String`) – application name
  - `version` (`String`) – application version (from Git) – if a version is not available it is used SHA-1 hash of last commit
  - `builtAt` (`String`) – build time

Operations

- **GET /info** – Get information about the application
  - Security: Public endpoint
  - Request: `-`
  - Response: `InfoDTO`

Token

Structures

- **`TokenDTO`**
  - `token` (`String`) – JSON Web Token
- **`TokenCreateDTO`**
  - `email` (`String`)
  - `password` (`String`)

Operations

- **POST /tokens** – Get token
  - Security: Public endpoint
  - Request: `TokenCreateDTO`
  - Response: `TokenDTO`

Organization

Structures

- **`OrganizationDTO`**
  - `uuid` (`UUID`) – unique identification
  - `name` (`String`) – human-readable name for package
  - `groupId` (`String`) – identification of organization

Operations

- **GET /organization/current** – Get information about current used organization
  - Security: Secured endpoint (no special permission needed)
  - Request: `-`
  - Response: `OrganizationDTO`
- **PUT /organization/current** – Modify current used organization
  - Security: Secured endpoint (need: `ORG_PERM`)
  - Request: `OrganizationDTO`
  - Response: `OrganizationDTO`

Users

Structures

- **`User`**
  - `uuid` (`UUID`) – unique identificator
  - `name` (`String`)
  - `surname` (`String`)
  - `email` (`String`)

  - role (Role) – one of the admin, datasteward, researcher or custom
  - permissions ([Permission]) – list of permissions (available permissions are UM_PERM, ORG_PERM, KM_PERM, KM_UPGRADE_PERM, KM_PUBLISH_PERM, PM_PERM, WIZ_PERM and DMP_PERM)

- **UserCreateDTO**
  - uuid (UUID) – unique identificator
  - name (String)
  - surname (String)
  - email (String)
  - role – one of the admin, datasteward or researcher
  - password (String)
- **UserPasswordDTO**
  - password (String)

Operations

- **GET /users** – List all users
  - Security: Secured endpoint (need: UM_PERM)
  - Request: -
  - Response: UserDTO
- **POST /users** – Create user
  - Security: Secured endpoint (need: UM_PERM)
  - Request: UserCreateDTO
  - Response: UserDTO
- **GET /users/:userId** – Get user detail
  - Security: Secured endpoint (need: UM_PERM)
  - Request: -
  - Response: UserDTO
- **GET /users/current** – Get my profile
  - Security: Secured endpoint (no special permission needed)
  - Request: -
  - Response: UserDTO

- **PUT /users/:userId** – Modify user
  - Security: Secured endpoint (need: `UM_PERM`)
  - Request: `UserDTO`
  - Response: `UserDTO`
- **PUT /users/:userId/password** – Change password to user
  - Security: Secured endpoint (need: `UM_PERM`)
  - Request: `UserPasswordDTO`
  - Response: -
- **PUT /users/current** – Modify my profile
  - Type: Secured endpoint (no special permission needed)
  - Request: `UserDTO`
  - Response: `UserDTO`
- **PUT /users/current/password** – Change my password
  - Security: Secured endpoint (no special permission needed)
  - Request: `UserPasswordDTO`
  - Response: -
- **DELETE /users/:userId** – Delete user
  - Security: Secured endpoint (need: `UM_PERM`)
  - Request: -
  - Response: -

Branch

Structures

- **`BranchDTO`**
  - `uuid` (`String`) – unique identification
  - `name` (`String`) – human-readable name for package
  - `groupId` (`String`) – identification of organization
  - `artifactId` (`String`) – identification of concrete knowledge model in organization
  - `parentPackageId` (`Maybe String`) – unique identification of parent package (if exists)

    - `lastAppliedParentPackageId` (`Maybe String`) – merge properties

- **`BranchWithStateDTO`**
    - `uuid` (`String`) – unique identification
    - `name` (`String`) – human-readable name for package
    - `groupId` (`String`) – identification of organization
    - `artifactId` (`String`) – identification of concrete knowledge model in organization
    - `parentPackageId` (`Maybe String`) – unique identification of parent package (if exists)
    - `lastAppliedParentPackageId` (`Maybe String`) – merge properties
    - `state` (`BranchState`) – one of the five state – `Default`, `Edited`, `Outdated`, `Migrating` or `Migrated`

Operations

- **GET /branches** – List all branches
    - Security: Secured endpoint (need: `KM_PERM`)
    - Request: `-`
    - Response: `BranchWithStateDTO`
- **POST /branches** – Create branch
    - Security: Secured endpoint (need: `KM_PERM`)
    - Request: `BranchDTO`
    - Response: `BranchDTO`
- **GET /branches/:branchId** – Get branch detail
    - Security: Secured endpoint (need: `KM_PERM`)
    - Request: `-`
    - Response: `BranchWithStateDTO`
- **PUT /branches/:branchId** – Modify branch
    - Security: Secured endpoint (need: `KM_PERM`)
    - Request: `BranchDTO`
    - Response: `BranchDTO`
- **DELETE /branches/:branchId** – Delete branch
    - Security: Secured endpoint (need: `KM_PERM`)
    - Request: `-`
    - Response: `-`

Knowledge Model

Structures

- **`KnowledgeModelDTO`**
  - includes a knowledge model

Operations

- **GET /branches/:branchId/km** – Get current version of knowledge model
  - Security: Secured endpoint (need: `KM_PERM`)
  - Request: -
  - Response: `KnowledgeModelDTO`

Event

Structures

- **`EventsDTO`**
  - list of events

Operations

- **GET /branches/:branchId/events** – List all events which were created in this branch
  - Security: Secured endpoint (need: `KM_PERM`)
  - Request: -
  - Response: `EventsDTO`
- **POST /branches/:branchId/events/__bulk** – Add events from request to branch events
  - Security: Secured endpoint (need: `KM_PERM`)
  - Request: `EventsDTO`
  - Response: `EventsDTO`
- **DELETE /branches/:branchId/events** – Delete all events which were created in this branch
  - Security: Secured endpoint (need: `KM_PERM`)
  - Request: -
  - Response: -

Version

Structures

- **`VersionDTO`**
  - `description` (`String`) – short description about changes which this version brings

Operations

- **PUT /branches/:branchId/versions/:version** – Create package from branch
  - Security: Secured endpoint (need: `KM_PUBLISH_PERM`)
  - Request: `-`
  - Response: `PackageDTO` (defined in Package API Specification (A))

Migration

Structures

- **`MigratorStateDTO`**
  - `branchUuid` (`UUID`) – unique identifier
  - `migrationState` (`MigrationState`) – one of the four migration state – `RunningState`, `ConflictState`, `ErrorState` or `CompletedState`
  - `branchParentId` (`String`) – `uuid` of parent package of the branch
  - `targetPackageId` (`String`) – `uuid` of target package on which we migrate
  - `currentKnowledgeModel` (`Maybe KnowledgeModel`) – current build knowledge model
- **`MigratorStateCreateDTO`**
  - `targetPackageId` (`String`)
- **`MigratorConflictDTO`**
  - `originalEventUuid` (`UUID`) – `uuid` of solving event
  - `action` (`MigrationConflictAction`) – one of the three action – `Apply`, `Edited` or `Reject`
  - `event` (`Maybe Event`)

Operations

- **GET /branches/:branchId/migrations/current** – Get current migration
  - Security: Secured endpoint (need: `KM_UPGRADE_PERM`)
  - Request: -
  - Response: `MigratorStateDTO`
- **POST /branches/:branchId/migrations/current** – Create migration
  - Security: Secured endpoint (need: `KM_UPGRADE_PERM`)
  - Request: `MigratorStateCreateDTO`
  - Response: `MigratorStateDTO`
- **DELETE /branches/:branchId/migrations/current** – Cancel migration
  - Security: Secured endpoint (need: `KM_UPGRADE_PERM`)
  - Request: -
  - Response: -
- **POST /branches/:branchId/migrations/current/conflict** – Solve conflict
  - Security: Secured endpoint (need: `KM_UPGRADE_PERM`)
  - Request: MigratorConflictDTO
  - Response: -

Packages

Structures

- **`PackageDTO`**
  - `id` (`String`) – unique identification
  - `name` (`String`) – human-readable name for package
  - `groupId` (`String`) – identification of organization
  - `artifactId` (`String`) – identification of concrete knowledge model in organization
  - `version` (`String`) – package version

  - `description` (`String`) – short description of changes which package brings to knowledge model
  - `parentPackageId` (`Maybe String`) – unique identification of parent package (if exists)

- **`PackageSimpleDTO`**
  - `name` (`String`) – human-readable name for package
  - `groupId` (`String`) – identification of organization
  - `artifactId` (`String`) – identification of concrete knowledge model in organization

Operations

- **GET /packages** – List all packages
  - Security: Secured endpoint (need: `PKG_PERM`)
  - Request: -
  - Response: `PackageDTO`
- **GET /packages/unique** – List all packages (more versions of the same package are displayed as one record)
  - Security: Secured endpoint (need: `PKG_PERM`)
  - Request: -
  - Response: `PackageSimpleDTO`
- **GET /packages/:pkgId** – Get concrete package detail
  - Security: Secured endpoint (need: `PKG_PERM`)
  - Request: -
  - Response: `PackageDTO`
- **DELETE /packages** – Delete all packages
  - Security: Secured endpoint (need: `PKG_PERM`)
  - Request: -
  - Response: -
- **DELETE /packages/:pkgId** – Delete concrete package
  - Security: Secured endpoint (need: `PKG_PERM`)
  - Request: -
  - Response: -

IO

Structures

- **`IOPackageDTO`**
  - `id` (`String`) – unique identification
  - `name` (`String`) – human-readable name for package
  - `groupId` (`String`) – identification of organization
  - `artifactId` (`String`) – identification of concrete knowledge model in organization
  - `version` (`String`) – package version
  - `description` (`String`) – short description of changes which package brings to knowledge model
  - `parentPackageId` (`Maybe String`) – unique identification of parent package (if exists)
  - `events` – list of events (changes)

Operations

- **GET /export/:pkgId** – Export package
  - Security: Public endpoint
  - Request: -
  - Response: File including `IOPackageDTO`
- **POST /import** – Import package)
  - Security: Secured endpoint (need: `PKG_PERM`)
  - Request: File including `IOPackageDTO`
  - Response: `PackageDTO` (defined in Package API Specification (A))

Appendix B

Acronyms

API Application Programming Interface. 2, 6, 7, 21, 65, 66, 68, 70, 76, 77, 83, 84

BSON Binary JSON. 64, 70

CI Continuous Integration. 23, 73, 84

CORS Cross-origin resource sharing. 63

CRUD Create, Read, Update, Delete. 8

DAO Data Access Object. 64

DTL Dutch Techcentre for Life Sciences. 4

FIT CTU Faculty of Information Technology Czech Technical University in Prague. 3, 4, 6, 14, 83

GHC Glasgow Haskell Compiler. 22

HTTP Hypertext Transfer Protocol. 76

JSON JavaScript Object Notation. 3, 4, 23, 35, 63, 66

JWT JSON Web Tokens. 23, 63, 66

LTS Long Term Support. 62

REST Representational State Transfer. 21, 66

REST API Representational State Transfer Application Programming Interface. 7, 19, 62, 63, 83, 84

UI User interface. 5

UML Unified Modeling Language. 2

UUID Universally unique identifier. 11, 13

Appendix C

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ....................................... the directory of source codes
  impl ........................................ 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
  assignment.pdf ....................... assignment of master's thesis
```