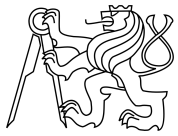
CZECH TECHNICAL UNIVERSITY IN PRAGUE

FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Data Stewardship Portal: Client-side Web Frontend
Student:	Bc. Jan Slifka
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 Portal (DSP) project
- Perform a brief review of state-of-the art Haskell-based solutions for web frontend development and select one for your task
- Design and implement a web frontend architecture of DSP.
- Design and implement a web UI for the Knowledge Model Editor and Migration Module.
- Elaborate a technical documentation of your solution in the form of descriptive text and generated API docs.
- Test your solution appropriately and document your results.

References

Will be provided by the supervisor.

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

Data Stewardship Portal: Client-side Web Frontend

Bc. Jan Slifka

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

January 7, 2018

Acknowledgements

I would like to thank to my supervisor Ing. Robert Pergl, Ph.D. for his guidance and enthusiasm for the project that motivated me greatly. I would also like to thank to the whole Faculty of Information Technology at CTU for everything I could learn there. I am profoundly grateful to my parents who supported me all the time throughout my studies.

Declaration

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

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 7, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Slifka, Jan. *Data Stewardship Portal: Client-side Web Frontend.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Práce popisuje proces návrhu, vývoje, testování a nasazení webového frontendu pro Data Stewardship Portal, který je součástí Interoperability platformy v ELIXIR CZ projektu. Portál zahrnuje rozsáhlý editor knowledge modelů a modul pro migrace. Pro webový vývoj je použito řešení založené na Haskellu.

Klíčová slova ELIXIR, Data Stewardship, Elm, The Elm Architecture, Webový vývoj, Webový klient, webpack, Haskell, PureScript, Docker

Abstract

The thesis goes through the process of design, development, testing and deployment of the client-side web frontend for the Data Stewardship Portal which is part of the Interoperability platform at ELIXIR CZ project. The portal includes extensive Knowledge Model Editor and Migration Module. Haskell-based solution is used for the web development.

Keywords ELIXIR, Data Stewardship, Elm, The Elm Architecture, Web Development, Web Client, webpack, Haskell, PureScript, Docker

Contents

List of Figures

List of Tables

Introduction

ELIXIR coordinates and develops life science resources across Europe so that researchers can more easily find, analyse and share data, exchange expertise, and implement best practices. This makes it possible for them to gain greater insights into how living organisms work. [1]

ELIXIR-CZ is working on Interoperability Platform[2]. There is a defined Knowledge Model and some tools for working with it. The Knowledge Model can be used to create a Data Management Plans using the Data Stewardship Wizard tool. Now, there is a need to create a complex portal around Knowledge Models management, data management plan generation and the data stewardship in general.

Goals

The main goal of the thesis is to create a client side web frontend for the Data Stewardship Portal. I will go through several tasks in order to successfully finish the project.

The first task is to understand the needs of the Data Stewardship Portal, analyze the requirements and functionality of individual modules.

The portal should be developed using some Haskell-based solution for web frontend development. I will perform a brief review of current possibilities and choose the most suitable one.

I will design and implement the architecture of the Data Stewardship Portal web fronted using the selected technology. The architecture should ensure the maintenance and future development of the project will be easy.

The portal should focus on knowledge models now. The most important parts are Knowledge Model Editor and Migration Module. My task is to design and implement the UI and the functionality of the web frontend for these modules.

The whole solution should have a good documentation to make it easy for other developers to collaborate in the future. It should also be tested

appropriately.

Methodology

The structure of the thesis follows the common software development process. At the beginning I will start with analyzing the state-of-the-art Haskell based solutions for web. I will focus on their maturity, usability, how well they are maintained and documented and how ready they are to be used for the real world projects. The output of the chapter will be the choice of the most suitable solution for the Data Stewardship Portal.

The next chapter is Analysis and Design. I will analyze what the current state is, what the requirements for the portal are, what should be the functionality of individual modules and how they should be designed. For each module, I will create a UML Use Case diagram, go through the individual screens needed and create a list of tasks that the user can do on each screen. Then I will use UML Activity Diagrams for user interface navigation. It is possible to use UML's extension mechanism to add new stereotypes. I will use the similar approach as Benjamin Lieberman suggests in his article UML Activity Diagrams: Detailing User Interface Navigation[3]. I will use the following elements and stereotypes:

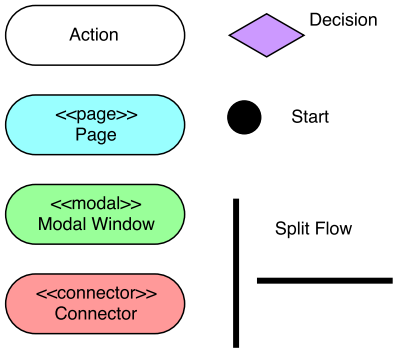


Figure 0.1: Activity Diagram Stereotypes and Elements

- **Action** represents activities performed by the user.
- **Page** represents a separate page.
- **Modal** represents a modal window.
- **Connector** refers to another diagram. It is used to connect more diagram flows together.
- Then I will use the standard UML Activity Diagram elements for the decision, flow start and flow split.

I don't need to analyze the data model or create the diagrams about that. All these things are handled on the server side, the client is just using the API to get the JSON data from the server but is not working with the database directly.

The Implementation chapter will be about the solution I picked. I will go more into details how the solution works and how the whole architecture is implemented within the solution. I will also describe how the development process works and what build tools are used.

The next chapter is about Testing and Deployment. I will describe what testing methods are used in the project and how they work. I will also evaluate the quality of the UI and how it could be improved. Then I will go more into details about how the portal fronted is deployed (I will also create a UML Deployment Diagram representing that) and how the deployment process works and how it is automated.

At the end I will evaluate what I have done through my thesis and what is the project state. I will take a look in the future and describe the next steps of project development and its possible improvements.

Chapter 1

State-of-the-art Haskell Based Solutions for Web

In this chapter I will describe why I want to use Haskell based solution and introduce some tools and terms connected with web development. Then I will go through the options how to use Haskell based solution and describe which one I choose and why.

1.1 Why Haskell Based Solution

Haskell[4] is a purely functional programming language that allows to write declarative, statically typed code. Using the functional programming we can gain a lot of advantages. The code itself is declarative which means it describes what result we want instead of how to get it, usually with less code. Pure functions has no side effects – regardless the context we run them in, the result is always the same. The state is immutable, we can't change it. We can only create a new state. Also the language is statically typed which should help to eliminate runtime exceptions.

Besides the advantages the other Data Stewardship projects (e.g. the backend for the DSP) are also written in Haskell. So it would be good to keep close.

The problem here is that the Data Stewardship Portal should be a web application. Which means it has to be a JavaScript code at the end of the day, because that's the only option how to run the code in the browser.

However, JavaScript is not the language we want to have the portal written in. It is not purely functional, nor statically typed. We cannot be sure whether the code will work or not without a lot of tests. Runtime exceptions are very common. There is no compilation so there are no real checks whether the functions or properties we are using really exist.

Nothing is lost though. There are many languages that actually compiles

to JavaScript, some of them also have the advantages similar to Haskell language. I will explore how applicable they are for the Data Stewardship Portal use case in this chapter.

1.2 Tools Introduction

Before exploring what Haskell-based solution I should use, I will shortly introduce the tools that are common for Haskell world and the tools that are common for web development world. I will refer to these tools in the following sections.

1.2.1 Haskell Tools

Haskell is usually using GHC[6] (Glasgow Haskell Compiler) to compile the source code into binary. It can be also used as an interactive environment for Haskell. Open source Haskell software is published in a central package archive called Hackage[7].

Cabal[8] is the Common Architecture for Building Applications and Libraries. It is used for building, packaging, distributing and cataloging Haskell packages. Authors can use Cabal to upload their libraries to Hackage. Users of the libraries can then install them from Hackage, also using Cabal.

1.2.2 Web Development Tools

When working on a web project, the most common package manager used is called npm[9]. It is a package manager for JavaScript. It has a global online repository (or private instances can be used) and a tool for installing the packages from the repository. It contains modules for web frontend and also Node.js.

Bower[10] is something similar to npm. It also has a global online repository with packages that can be installed by users. However, it focuses on frontend packages only.

Webpack[11] is a tool for bundling frontend modules for usage in a web browser. While creating the modules, webpack is loading different file types with appropriate loaders that can, for example somehow transform the code. Therefore it can be used for other languages that compiles to JavaScript. It can be also used for other resources like styles or images. So we can have a single tool to create the distribution build for the whole frontend application.

Google Closure Compiler[12] is a tool for making JavaScript code smaller by analyzing the code, removing unused parts and minimizing the rest (it changes the symbol names to shorter ones, remove unnecessary whitespaces and comments, etc.)

1.3 Haskell

The first option is to compile Haskell itself into JavaScript. There are several tools that provide this functionality.

1.3.1 Haste

Haste[13] is a tool for compiling Haskell code into JavaScript that can be used in a web browser. It is also possible to compile the program not only into JavaScript but also into binary that can be used as a server. Both programs can than talk to each other using some kind of internal Haste protocol. This ensures the communication is type safe. On the other hand, it requires to use Haste on both ends which can be very limiting.

1.3.1.1 Code & Libraries

The code in Haste is quite straightforward. We can use equivalents to native JavaScript API like `getElementById` or setting `innerHTML`. Haste provides these helpers into the Haskell code. All the business logic behind is then written purely in Haskell. Everything is wrapped into `main` function that is called once the document is loaded.

Haste supports the full Haskell language and brings its own packages for the common things used in JavaScript – DOM, JavaScript Events, AJAX, JSON parsing, Canvas manipulations and more.

Haste doesn't come with any templating system by default though. It has packages to wrap the JavaScript API and use it in Haskell code. There is no framework or other tools that would help with writing complex applications though. Everything has to be created from scratch.

Some libraries extending the basic Haste functionality exist. They are usually trying to improve the DOM manipulations or port existing JavaScript libraries like React[5]. However, these are usually outdated, not actively maintained and not well documented. Therefore it would not be a good idea to use them in production.

1.3.1.2 Documentation & Other Materials

Haste has an online API documentation that covers all the functionality. It is very brief though and contains no examples of usage. There are also several examples in the Github repository and links to few tutorials can be found on the website.

1.3.1.3 Tools

We need Haste compiler to be able to compile the code into JavaScript. The compiler can be either downloaded as a binary or installed using Cabal.

The result JavaScript generated by the compiler is according to the Haste homepage normally less than 3x the size of an equivalent hand-written program. However, the compiler comes with the option to use Google Closure Compiler (which is slightly modified and included in Haste, because the default one is breaking the builds) and then the size of the minified build is reasonable.

There are no usable integration with other build tools used for web development like webpack. The compiler has to be manually run every time or some ad hoc solution has to be created.

1.3.2 Fay

Fay[14] allows the users to use a subset of Haskell language and compile it to JavaScript. It supports the fundamental data types that can be supported in JavaScript and it has simple foreign function interface (FFI) to interact with the JavaScript API.

1.3.2.1 Code & Libraries

The code itself is conceptually something similar to Haste. It is again similar to using plain JavaScript. However, there is one important difference. Fay doesn't have any default packages that would wrap the JavaScript API. Everything has to be written from scratch using foreign function interface (FFI). FFI is using a `Fay` monad for side effects.

Listing 1.1: Example of FFI usage in Fay [15]

```
1 setBodyHtml :: String -> Fay ()
2 setBodyHtml = ffi "document.body.innerHTML = %1"
```

As we can see, Fay brings a way to wrap JavaScript API in Haskell code using FFI. However, we need to actually write everything by ourselves. There are some official packages with the functionality, e.g. for manipulating the DOM or even wrapping the whole JavaScript libraries like jQuery. These are incomplete, not actively maintained and not very well documented though.

In terms of some high level libraries that would provide more advanced things (like generating views from templates) there is nothing available for Fay. We should be able to port existing JavaScript libraries or implement them ourselves using FFI.

1.3.2.2 Documentation

Fay itself has a basic documentation explaining everything from the installation to the usage, including also some useful tips how to reduce the output size. Since the only API provided by Fay is FFI and the `Fay` monad the documentation doesn't have to include much.

1.3.2.3 Tools

The Fay compiler can be simply installed using Cabal. If we want to integrate the compiler with other build tools used for building web frontend (e.g. webpack) we need to create the integration by ourselves.

The output JavaScript from the Fay compiler is pretty small. We can also use Google Closure Compiler to make it even smaller.

1.3.3 GHCJS

GHCJS [16] is a very complex Haskell to JavaScript compiler that uses GHC API. It supports large variety of Haskell features, including all type system extensions supported by GHC.

1.3.3.1 Code & Libraries

The code itself works in a similar way to Haste. GHCJS comes with a collection of packages that provides types for communication with JavaScript or wrapping the DOM API.

There is an interesting project called Reflex[17] which is functional reactive programming interface and engine that can be use with GHCJS. It also has a framework called Reflex-DOM[18] which is suitable for development of single-page applications.

Listing 1.2: Example of Reflex usage [19]

```
1 {-# LANGUAGE OverloadedStrings #-}
2 import Reflex.Dom
3
4 main = mainWidget $ el "div" $ do
5   t <- textInput def
6   dynText $ _textInput_value t
```

This example code creates a text input and dynamic text element showing the value of the text input.

There is also a project called Reflex-Platform[19] which helps to set up an environment for building the Reflex application using GHCJS.

1.3.3.2 Documentation & Other Materials

GHCJS Github repository contains extensive Readme describing the installation process and the usage of the compiler. There is also Wiki with user guide which is not yet completed. It describes more in detail how to use GHCJS packages in Haskell code, how to do the deployment and code minification and more.

There is another repository with several complete examples[20] of how to use GHCJS for the real world problems.

1.3.3.3 Tools

We need to clone the GHCJS repository and then run install script using Cabal. The installation takes very long time (it can be more than one hour).

Since GHCJS is trying to support every possible Haskell feature, the output JavaScript code is huge. Google Closure Compiler can be used to minify it, however, even the minified code from GHCJS is bigger than builds that are not minified from the other tools.

There is also unofficial ghcjs-loader[21] for webpack. Unfortunately the project is outdated and not working with the recent version of webpack.

1.4 PureScript

PureScript[22] is a strongly-typed functional language that can be compiled to JavaScript. The syntax is similar to Haskell with some differences[23].

Listing 1.3: Example PureScript code [22]

```
1 import Prelude
2 import Control.Monad.Eff.Console (log)
3
4 greet :: String -> String
5 greet name = "Hello, " <> name <> "!"
6
7 main = log (greet "World")
```

There is a Pursuit package database[24] which contains a list of available PureScript packages with their documentation. The packages are usually published in the Bower repository and can be easily installed from there.

A lot of PureScript libraries providing different kinds of utility or wrapping existing JavaScript libraries can be found in Pursuit. There are also some frameworks for web applications development. Some of the popular ones are `thermite`, `halogen` and `pux`.

1.4.1 Thermite

Thermite[25] is more high level wrapper for `purescript-react`[26] (which is a library with React bindings to PureScript). It has a brief documentation[27] describing the API and showing example usage.

1.4.2 Halogen

Halogen [28] is declarative UI library. The application created with Halogen is made of a tree of components. These components are self-contained units that has a state and can render themselves. Each component defines queries and messages. Queries are used when interacting with the component – they

can be used to change the state or ask about the component state. Messages are used to notify listeners about the activity within the component.

Halogen has a detailed guide[29] describing how to use the framework. Unfortunately, not all chapters are finished yet. It also has several examples[30] of usage from simple to more complex.

1.4.3 Pux

Pux[31] is another PureScript UI library. Pux application consists of a type for application `State`, type for `Events`, a function called `foldp` that produces new state from previous state and event, and a `view` function that produces HTML based on the current application state.

Listing 1.4: Pux example [31]

```
1  data Event = Increment | Decrement
2
3  type State = Int
4
5  -- | Return a~new state (and effects) from each event
6  foldp :: ∀ fx. Event -> State -> EffModel State Event fx
7  foldp Increment n = { state: n + 1, effects: [] }
8  foldp Decrement n = { state: n - 1, effects: [] }
9
10 -- | Return markup from the state
11 view :: State -> HTML Event
12 view count =
13   div do
14     button #! onClick (const Increment) $ text "Increment"
15     span $ text (show count)
16     button #! onClick (const Decrement) $ text "Decrement"
```

Pux also has a detailed guide[32] explaining everything from the basics to more advanced things and API Reference. Pux has a repository with examples[33] showing how to solve common problems (e.g. AJAX, nested `foldp` function for more complex applications, forms and more).

Another important thing in web applications is routing. Pux provides the functionality for routing out of the box in `Pux.Router` module.

Unlike the other frameworks, when rendering the view Pux has an intermediate step. It is using React under the hood and it has to first render from Smolder Markup (which is Pux representation) to React Virtual DOM and then the React Virtual DOM is rendered. This makes Pux slower compared to other frameworks. It should be resolved in the future but performance is not the priority for Pux yet.

1.4.4 PureScript Tools and Development

PureScript compiler can be installed using npm. All the libraries and frameworks are published in Bower repository. That makes the installation very easy.

There is also a webpack loader for PureScript so it is easy to use webpack and all its features (like webpack dev server with live reload) during the development process.

1.5 Elm

Elm[34] is a reactive purely functional statically typed programming language for web development inspired by Haskell. Elm compiles to JavaScript. Because of the types, problems are detected during compilation, therefore there are no runtime errors in practice.

Elm is not just a language but also a framework to build a web application. It comes with a simple pattern called The Elm Architecture[36] that helps to make the architecture of a web application consistent, well-defined and scalable.

Elm has its own highly optimized Virtual DOM implementation. In benchmarks [37], it beats current popular frameworks like React or Angular[35].

1.5.1 Tools

Elm comes with several command line tools. They can be easily installed on Mac and Windows or anywhere using npm. The tools are:

- `elm-repl` – REPL where we can run Elm expressions.
- `elm-reactor` – It is used to build an Elm project and start a local server where it serves the application.
- `elm-make` – Build tool for Elm projects.
- `elm-package` – Download and publish Elm packages.

Besides that, it is also possible to build Elm using webpack with appropriate webpack loader[47]. Then we don't even need to install previous packages explicitly.

1.5.2 Libraries

Elm has its own package catalog[38] where all the packages are published. The packages can be installed using `elm-package` tool. When publishing a new package to the catalog Elm can detect API changes and force semantic versioning.

1.5.3 Documentation and Tutorials

Elm has a very extensive tutorial[39] which leads the reader from the language basic through the process of creating the whole application, setting up the project and using The Elm Architecture.

There is also a language reference for the modules that describes all the functions and types provided by the module. The libraries in package catalog also usually have everything well documented.

1.5.4 Elm in Production

Elm is not just experimental language, there are a lot of companies that are using it in production[40].

1.6 Comparison

1.6.1 Installation and Development

It is important that the tools needed to use the language are easy to install and use. It should also be easy to integrate the tools into continuous integration.

Haskell. The tools for compiling Haskell to JavaScript can be installed using Cabal, which is standard for Haskell developers. The problem here is that there are no usable integration with web development tools out of the box, e.g. live reload during development.

PureScript. Everything needed can be easily installed using npm and bower. There is also a webpack integration.

Elm. Elm is easy to install with installation package or via npm. It has also a webpack integration.

While PureScript and Elm are very easy to start developing in, the Haskell tools needs more time and effort to make everything work and the development itself is not so smooth due to missing webpack integration.

1.6.2 Libraries and Frameworks

We don't want to reinvent the wheel during the development so we want to have some libraries available. We also want to have a framework to keep the application structure organized and clean.

Haskell. In terms of libraries we can theoretically used everything available in Haskell (not everything is supported by all compilers though). None of the tools for compiling Haskell to JavaScript has a solid framework that would be ready to use for a complex application.

PureScript. There is a Pursuit package database for PureScript libraries that contains a lot of packages. The frameworks for web development in PureScript exist but they are not yet mature enough to be used for complex applications in production.

Elm. There are many Elm packages available covering a lot of different things. The Elm Architecture is a framework included in the language. Following its patterns, it is easy to maintain even complex web applications.

There are a lot of libraries available for different tasks for PureScript and Elm that should cover most common problems. However the frameworks for PureScript are not documented and/or maintained enough to be used.

1.7 Conclusion

The Haskell solutions don't have the tooling to make the development process comfortable enough. There are no frameworks that would be mature enough to be used for serious application (with the exception of Reflex, which is also in very early stage).

PureScript is a general purpose language that can not only be used for web but also in node environment for backend. The available frameworks are not yet ready for the production though – they lack enough quality documentation or its performance is not yet the top priority.

Elm doesn't have all the features that Haskell has, however, it focuses on web development and has everything needed for that. The Elm Architecture makes creating of complex web applications easy and helps to keep them clean and healthy while new features are added. The tooling provided for Elm is easy to install and use. In my opinion, it is the best option for using the purely functional programming language for the web frontend these days. That's why I decided to choose Elm for the Data Stewardship Portal.

Chapter 2

Analysis and Design

In this chapter, I will first talk about what the current state is and why the Data Stewardship Portal is needed. Then I will go through the planned portal structure and the individual modules the portal will have and describe what functionality should be provided by them.

2.1 Current State

The idea behind the Knowledge Model is that it consists of large core which is applicable everywhere and localizations which extend or overwrite the core. Localizations can be stacked – one localization can extend another localization [41].

The Knowledge Models for the core and localizations are written in a JSON format and are managed in a Github repository. The JSON files are edited and updated manually. This is not an ideal situation because manual editing of large JSON files can be tedious, confusing and error-prone. Moreover, it requires the editor to understand the JSON format.

The Knowledge Models are then used within Data Stewardship Wizard in order to create Data Management Plans.

2.2 Requirements

The Data Stewardship Portal (DSP) should be created.

2.2.1 Functional Requirements

Portal management. Each organization using the Knowledge Models will run its own instance of DSP. Therefore it should provide an interface to edit organization details and manage organization users.

Knowledge Model editor. DSP should provide a user-friendly editor for the Knowledge Models. The editor should allow the users to edit the Knowledge Model using clean and well-arranged user interface and without the need to understand the data structures that are used to save the Knowledge Model.

Import and export. DSP should allow the import and export of the Knowledge Models. Each organization can produce its own Knowledge Models and these can be exported and imported into the portal of another organization and used. Also the new version of the core Knowledge Model can be imported this way.

Versions. DSP should provide a way to manage the versions of the Knowledge Models. When the Knowledge Models can be shareable it is necessary to version them.

Migrations. DSP should provide a way to update the localization when a new version of the core or the parent localization is released. The migration process should allow the users to see all the changes and pick what changes they need in their localization.

2.2.2 Non-functional Requirements

Extensibility. DSP should be ready to also integrate the Data Stewardship Wizard and generate Data Management Plans in the future.

Platform. DSP should be a web application that works in a recent versions of the major desktop browsers.

Technical documentation. DSP should be documented well enough to make it easy for other developers to collaborate.

Deployment. It should be easy for the organizations to deploy DSP into their infrastructure.

2.3 Definition of Terms

2.3.1 Data Stewardship Portal

The Data Stewardship Portal covers everything that is described later. One instance of the portal is used per Organization. Organization Users are using the portal to manage the Knowledge Models and Knowledge Model Packages that can be used to generate Wizards and Data Management Plans.

2.3.2 Organization

Defines the organization that is using the Data Stewardship Portal.

Properties

- **Organization name**
- **Organization Group ID.** It is used with packages produced by the organization to unambiguously define them.

2.3.3 User

Defines the user that is using the Data Stewardship Portal.

Properties

- **Email.** User's email is used for logging in the portal.
- **Name.** User's first name (and middle name if any).
- **Surname.** User's surname.
- **Role / Permissions.** The role and permissions that the user has assigned within the portal.
- **Password.** User's password that is used when logging in the portal.

2.3.4 Role

The role is a set of permissions. It defines what permissions the user has.

2.3.5 Permission

Permissions are used to determine whether the user is allowed to perform certain actions.

2.3.6 Knowledge Model

Knowledge Model is a complex data structure that contains some metadata and Chapters, Questions, Answers, Follow-up Questions, References and Experts in a tree-like structure.

Knowledge Model is used to generate a complex form about the data needed that can be later filled in order to get the Data Management Plan.

Properties

- **Name.** Defines the name of the Knowledge Model.
- **UUID.** Universally unique identifier of the Knowledge Model.
- **Artifact ID.** Human readable identifier of the Knowledge Model. It should be unique within the organization.
- **Parent Knowledge Model.** Every Knowledge Model can be based on another Knowledge Model. That means it has all the data from its parent and then its own local changes. Parent Knowledge Model is defined as `<Organization Group ID>:<Artifact ID>:<Version>`. Knowledge Model doesn't have to have the parent.
- **Chapters.** a list of Chapters that are included in the Knowledge Model.

2.3.7 Chapter

Knowledge Model is divided into Chapters grouping the Questions that belongs together.

Properties

- **Title.** Chapter title (e.g. *Design of experiment*).
- **UUID.** Universally unique identifier of the Chapter.
- **Text.** Text that describes the Chapter.
- **Questions.** a list of Questions that belongs to the Chapter.

2.3.8 Question

Defines the Question that should be asked.

Properties

- **Title.** This is the actual question (e.g. *Is there any pre-existing data?*).
- **UUID.** Universally unique identifier of the Question.
- **Short UUID.**
- **Text.** Text that describes more details about the Question and how to answer it
- **Answers.** List of Answers to the Question.
- **References.** List of References that belongs to the Question.
- **Experts.** List of Experts that belongs to the Question.

2.3.9 Answer

Each Question has a set of Answers that the respondent can choose from.

Properties

- **Label.** This is the actual answer (e.g. *No*).
- **Advice.** Answers can have some advice text that should help choosing the answer.
- **Follow-up Questions.** List of the Follow-up Questions that should be asked if the Answer was selected.

2.3.10 Expert

Questions can have Experts assigned to them. Expert is a person who should know how to answer the specific question and could be asked by respondents.

Properties

- **Name.** Full name of the Expert for the Question.
- **Email.** Contact email for the Expert.

2.3.11 Reference

Question can have a reference to the book Data Stewardship for Open Science: Implementing FAIR Principles by Barend Mons[42] where the respondent can find more information.

Properties

- **Chapter.** The Chapter in the book.

2.3.12 Follow-up Question

Defines the Question that is asked only if a specific Answer was selected. Otherwise it is the same as a top-level Question.

2.3.13 Knowledge Model Package

The Knowledge Model can be exported out of the portal or imported into the portal as a Package. The Package is a special file type that contains the Knowledge Model information. The Package contains the Knowledge Model in a specific Version. The whole package with version is defined by `<Organization Group ID>:<Artifact ID>:<Version>`.

2.3.14 Knowledge Model State

Knowledge Models created within the portal can be in various states.

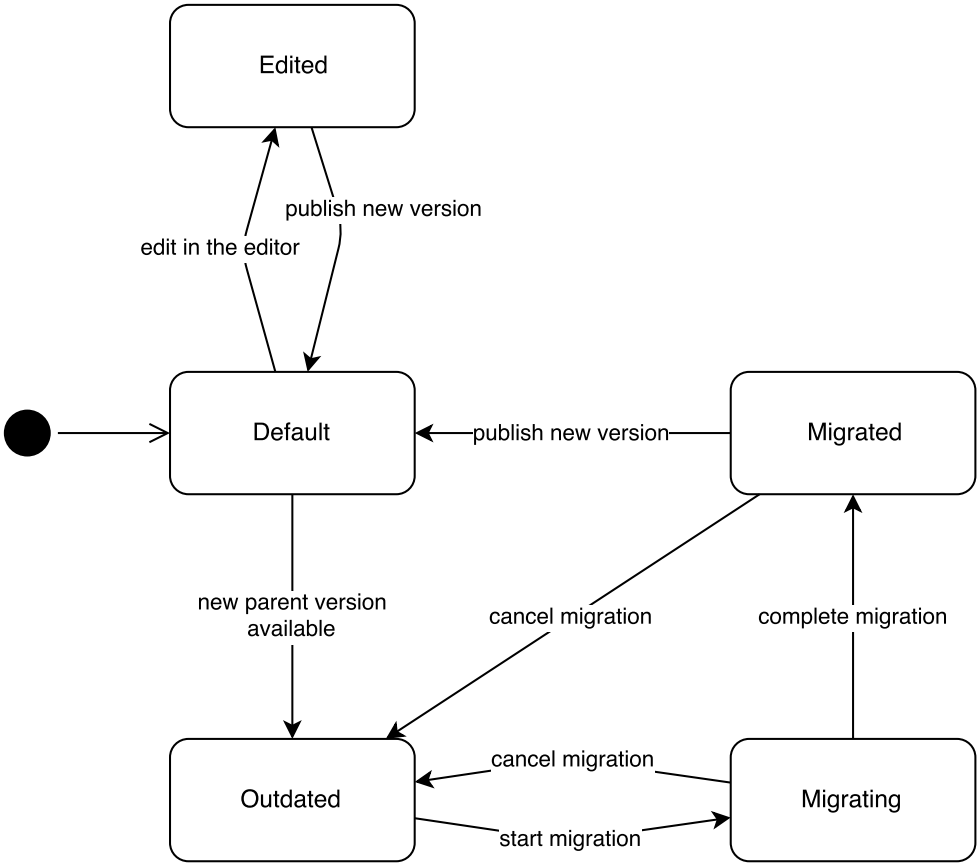


Figure 2.1: Knowledge Model State Diagram

States

- **Default.** The default state. Nothing has been done with the Knowledge model since the last version.
- **Edited.** There are changes made in the Knowledge Model that has not been published as a new version yet.
- **Outdated.** There is a newer version of the Knowledge Model Parent.
- **Migrating.** The migration to a newer version of the Knowledge Model Parent is in progress.

- **Migrated.** The migration to a newer version of the Knowledge Model Parent has been completed but the new version of the Knowledge Model has not been published yet.

2.3.15 Knowledge Model Migration

Knowledge Model can become outdated when a new version of its Parent Knowledge Model is published or imported to the portal. Knowledge Model Migration is the process of applying the new changes from the Parent Knowledge Model to the local Knowledge Model.

Users who are doing the migration go through the changes and decides whether they want to apply the new changes to their local Knowledge Model or reject them.

2.4 Project Overview

The whole portal is divided into 4 modules for now, each for different set of tasks. These modules cover the system management in terms of users and organization and working with knowledge models (editing, migrating, importing, exporting, etc.).

There will 2 more modules in the future. They will be for the tasks that are using the Knowledge Models.

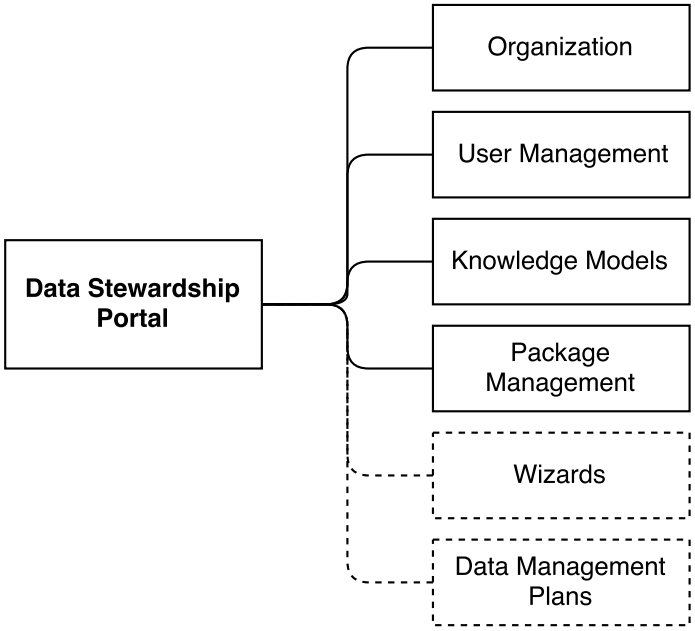


Figure 2.2: Project Modules

2.4.1 Modules

Organization This is a simple module for managing the organization settings.

User Management The portal is used by the users, therefore we need a way to manage user accounts. That's what this module is for.

Knowledge Models This module is for working with Knowledge Models created by the Organization.

Package Management This module is for importing and exporting Knowledge Model Packages and share them between different portals.

Wizards Wizards will be generated from the Knowledge Models. The user will be able to choose the Knowledge Model and fill in the wizard according to their project.

Data Management Plans When the users fills a wizard, they will be able to use the answers to generate a Data Management Plan.

2.5 Organization Settings Module

Organization settings module is used by admins to edit organization details which is the only use case for the module.

2.5.1 Use Cases

The only use case for this module is editing organization detail.

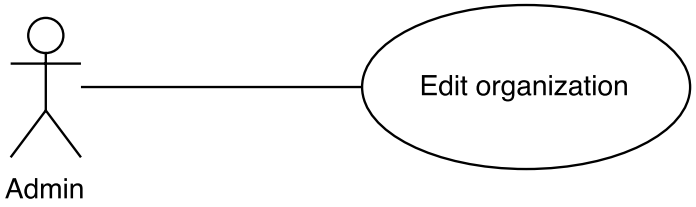


Figure 2.3: Organization Use Cases

2.5.2 Tasks

Organization module is actually only one form with organization details. The only task the admin can do is to save the changes.

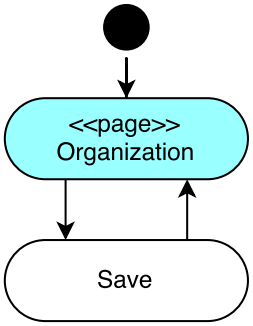


Figure 2.4: Organization Settings Module

2.6 User Management Module

Users with ADMIN role can use this module to manage user accounts for other users that will use the portal.

2.6.1 Use Cases

Within the user management module admin can:

- List users
- Create new user account
- Edit existing user account
- Delete existing user account

2.6.2 Tasks

User management module consists of several screens where admins can perform various tasks.

User Management The main screen shows a list of existing user accounts with their details.

Task list

- Create User
- Edit User
- Delete User

Create User Create user screen shows a form where an admin can fill in the new user account details and role.

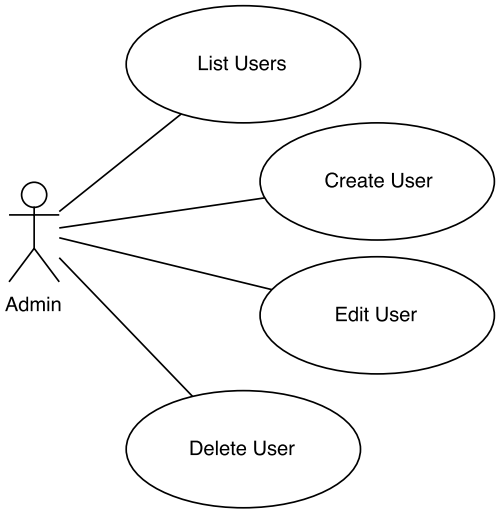


Figure 2.5: User Management Use Cases

Task list

- Save – create new account and return to the main screen.
- Cancel – return back to the main screen without creating the account.

Delete User Modal When an admin wants to delete the user account, the modal window is shown to confirm the deletion.

Task list

- Delete – confirm the deletion and close the modal.
- Cancel – close the modal only.

Edit User Profile The form for editing user profile is split into two parts – one contains the form for user details (email, name, etc.) and role and the other the form for changing the password. When an admin clicks on edit actions the form for editing user details is presented.

Task list

- Save – save the changes in user profile form.
- Password – navigate to the other form for changing user password.

Edit User Password The other part of the editing user profile. This one for changing the password.

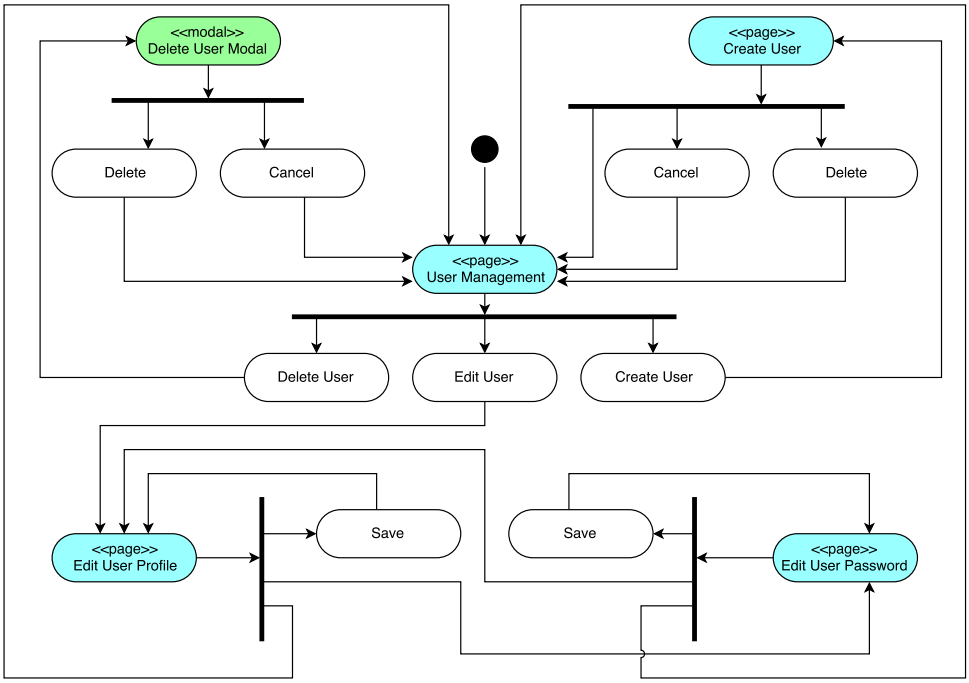


Figure 2.6: User Management Module

Task list

- Save – save the password change.
- Profile – navigate to the form for changing user details.

2.7 Knowledge Models Module

This module is used by Data Stewards to manage the internal Knowledge Models.

2.7.1 Use Cases

- List knowledge models
- Create new knowledge model
- Edit existing knowledge model
- Upgrade knowledge model
- Delete knowledge model

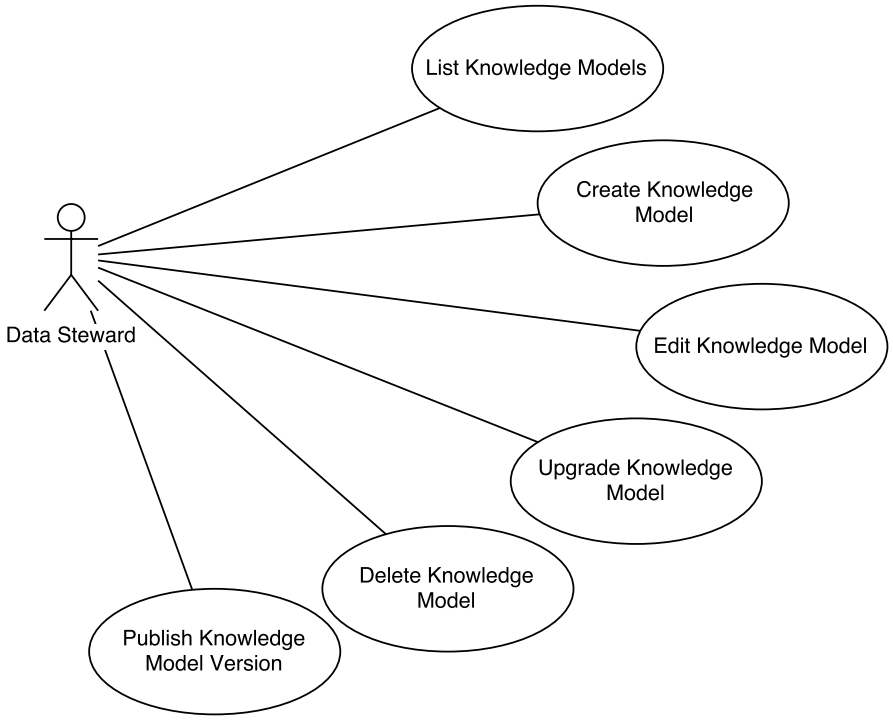


Figure 2.7: Knowledge Models Use Cases

- Publish new knowledge model version

2.7.2 Tasks

There are several screens and modal windows where Data Stewards can perform various actions.

Knowledge Models The main screen shows a list of Knowledge Models created in the portal. Based on the Knowledge Model state, Data Stewards can do different actions.

Task list

- Delete Knowledge Model – This action can be performed in every Knowledge Model state.
- Edit – If the Knowledge Model is in Default, Edited or Outdated state, Data Stewards can open the Knowledge Model editor and make some changes.
- Upgrade – If the Knowledge Model is in Outdated state, it can be upgraded.

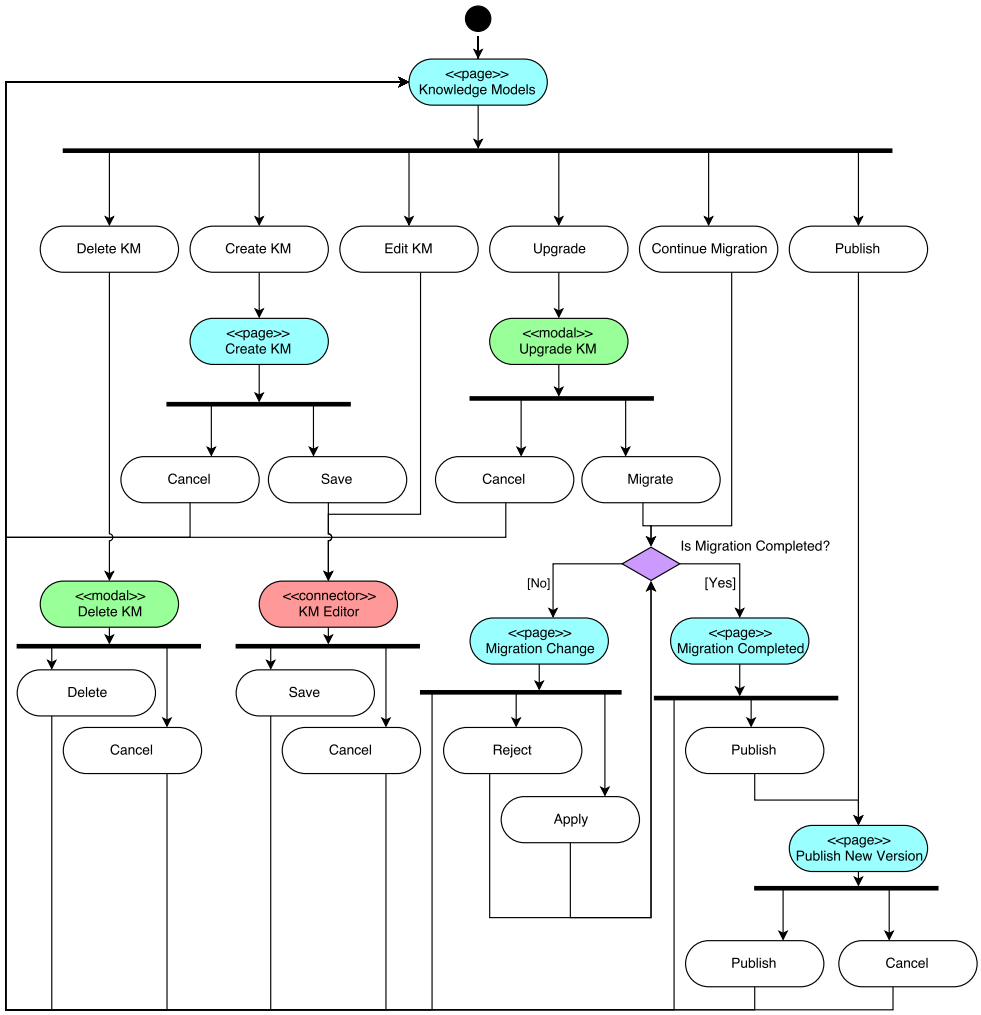


Figure 2.8: Knowledge Models Module

- Continue migration – If the Knowledge Model is in Migrating state, Data Stewards can choose to continue the migration.
- Cancel migration – If the Knowledge Model is in Migrating or Migrated state, the migration can be also canceled.
- Publish – If the Knowledge Model is in Edited or Migrated state, it can be published as a new version.

Delete Knowledge Model Modal Confirmation modal window that is displayed when Data Stewards want to delete the Knowledge Model.

Task list

- Delete – Confirm the deletion of the Knowledge Model.
- Cancel – Just close the modal window.

Knowledge Models Editor Knowledge Model editor contains screens with forms for editing all the entities within the Knowledge Model.

Task list

- Save – Save all the changes made in the editor.
- Cancel – Cancel all the changes made in the editor.

Editor tasks for individual editor screens and entities are described in the next section.

Upgrade Knowledge Model Modal a modal window that is shown when Data Stewards want to upgrade the Knowledge Model. It contains a select box for selecting a new parent Knowledge Model that the current Knowledge Model should be migrated to.

Task list

- Migrate – Confirm the new parent selection and start a migration.
- Cancel – Close the modal window.

Migration View This is the view used during the migration of the Knowledge Model. What user can see depends on whether the migration is completed or not.

Migration Change View When the migration is not yet completed it shows the current change (some entity from the parent Knowledge Model was added, edited or deleted) and the Data Steward needs to decide what to do with the change.

Task list

- Reject – Data Steward doesn't want to apply the change from the parent Knowledge Model to their Knowledge Model.
- Apply – Data Steward wants to apply the change from the parent Knowledge Model.

Migration Completed View When the last change is resolved this view will inform the user that the migration is completed.

Task list

- Publish – Go to the Publish new version view.

Publish new version view This view is used for publishing new versions of Knowledge Models. Data Steward needs to fill in the new version number and write the description about the version.

Task list

- Publish – Confirm the creation of the new Knowledge Model Package.
- Cancel – Cancel the publishing of a new version.

2.7.3 Editor Tasks

This section describes the tasks users can perform in the Knowledge Model Editor.

Knowledge Model Editor The default screen when the editor is opened. Data Stewards can edit Knowledge Model details and manage the Chapters here.

Task list

- Add Chapter – Create a new Chapter and open the Chapter Editor.
- Edit Chapter – Open an existing Chapter in the Chapter Editor.
- Reorder Chapters – Change the order of the Chapters in the Knowledge Model.

Chapter Editor This editor is used for editing the Chapter details and managing the Questions that the Chapter contains.

Task list

- Done – Confirm the changes and return back to the Knowledge Model Editor.
- Cancel – Cancel the changes and return back to the Knowledge Model Editor.
- Delete – Delete the Chapter and all its Questions.
- Add Question – Create a new Question and open the Question Editor.
- Edit Question – Open an existing Question in the Question Editor.

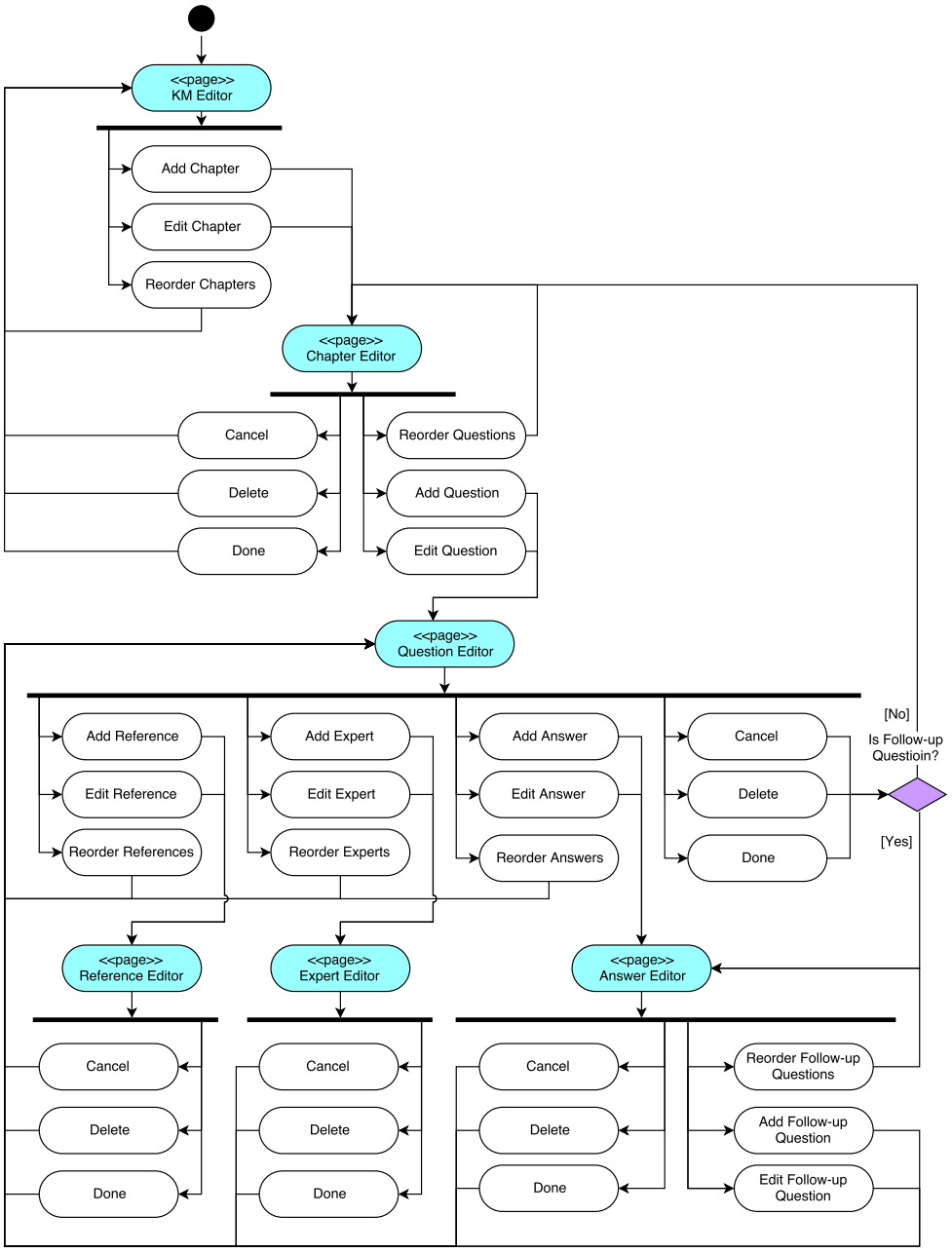


Figure 2.9: Knowledge Model Editor

- Reorder Questions – Change the order of the Questions in the Chapter.

Question Editor This is the editor for editing Questions and Follow-up Questions details and their children (Answers, References and Experts).

Task list

- Done – Confirm the changes and return back to the Chapter Editor in case of top level Question or to the Answer editor in case of Follow-up Question.
- Cancel – Cancel the changes and return back to the appropriate editor.
- Delete – Delete the question and all its children.
- Add Answer – Add a new Answer and open the Answer Editor.
- Edit Answer – Open an existing Answer in the Answer Editor.
- Reorder Answers – Change the order of the Answers in the Question.
- Add Reference – Add a new Reference and open the Reference Editor.
- Edit Reference – Open an existing Reference in the Reference Editor.
- Reorder References – Change the order of the References in the Question.
- Add Expert – Add a new Expert and open the Expert Editor.
- Edit Expert – Open an existing Expert in the Expert Editor.
- Reorder Experts – Change the order of the Experts in the Question.

Answer Editor The editor for editing Answers and manage their Follow-up Questions.

Task list

- Done – Confirm the changes and return back to the Question editor.
- Cancel – Cancel the changes and return back to the Question editor.
- Delete – Delete the Answer and all its Follow-up Questions.
- Add Follow-up Question – Add a new Follow-up Question and open the Question Editor.
- Edit Follow-up Question – Open an existing Follow-up Question in the Question Editor.

- Reorder Follow-up Questions – Change the order of the Follow-up Questions in the Answer.

Expert Editor Editor for editing Experts.

Task list

- Done – Confirm the changes and return back to the Question editor.
- Cancel – Cancel the changes and return back to the Question editor.
- Delete – Delete the Expert.

Reference Editor Editor for editing References.

Task list

- Done – Confirm the changes and return back to the Question editor.
- Cancel – Cancel the changes and return back to the Question editor.
- Delete – Delete the Reference.

2.8 Package Management Module

Package Management module is used by Data Stewards to export internal Knowledge Model Packages and share them with other organizations or import external Knowledge Model Packages from other organizations.

2.8.1 Use Cases

- List Packages
- View Package Detail
- Import Package Version
- Export Package Version
- Delete Package
- Delete Package Version

2.8.2 Tasks

There are several screens and modal windows used for managing Knowledge Model Packages in the portal.

Package Management The main screen shows a list of Knowledge Model Packages.

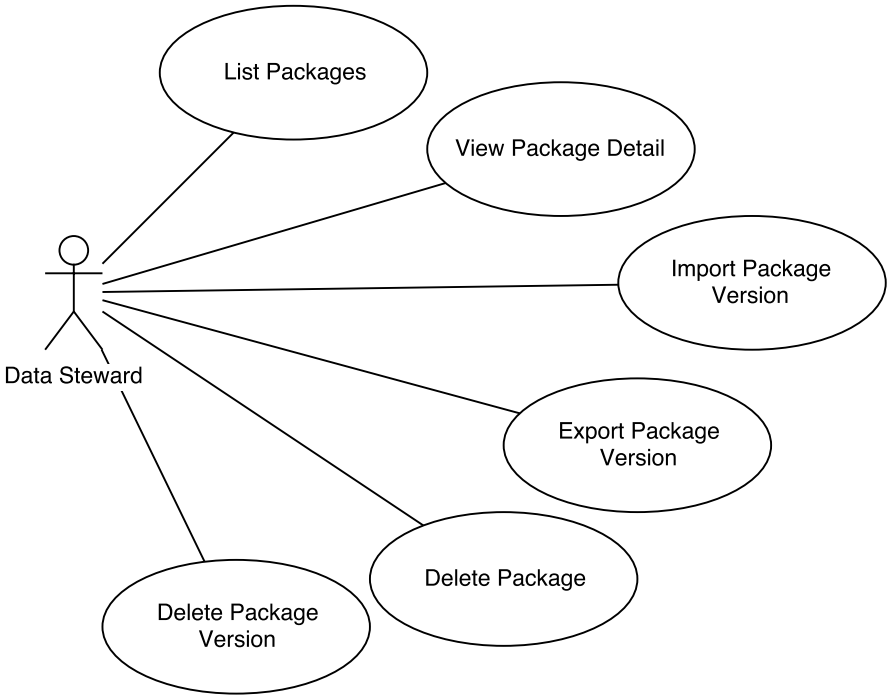


Figure 2.10: Package Management Use Cases

Task list

- Show Detail – Navigate to Package Detail View.
- Import – Navigate to Import Package.

Import Package: File Selection This is the first step of importing a Package to the portal. Data Stewards have to select the file from their computer and then they move to the next view.

Task list

- Choose file – Open the browser file selection window where the user can select the file.
- Drag & Drop File – Instead of selecting file in the window users can simply drag & drop the file from their local file system.

Import Package: Selected File The second step after the user has selected the file.

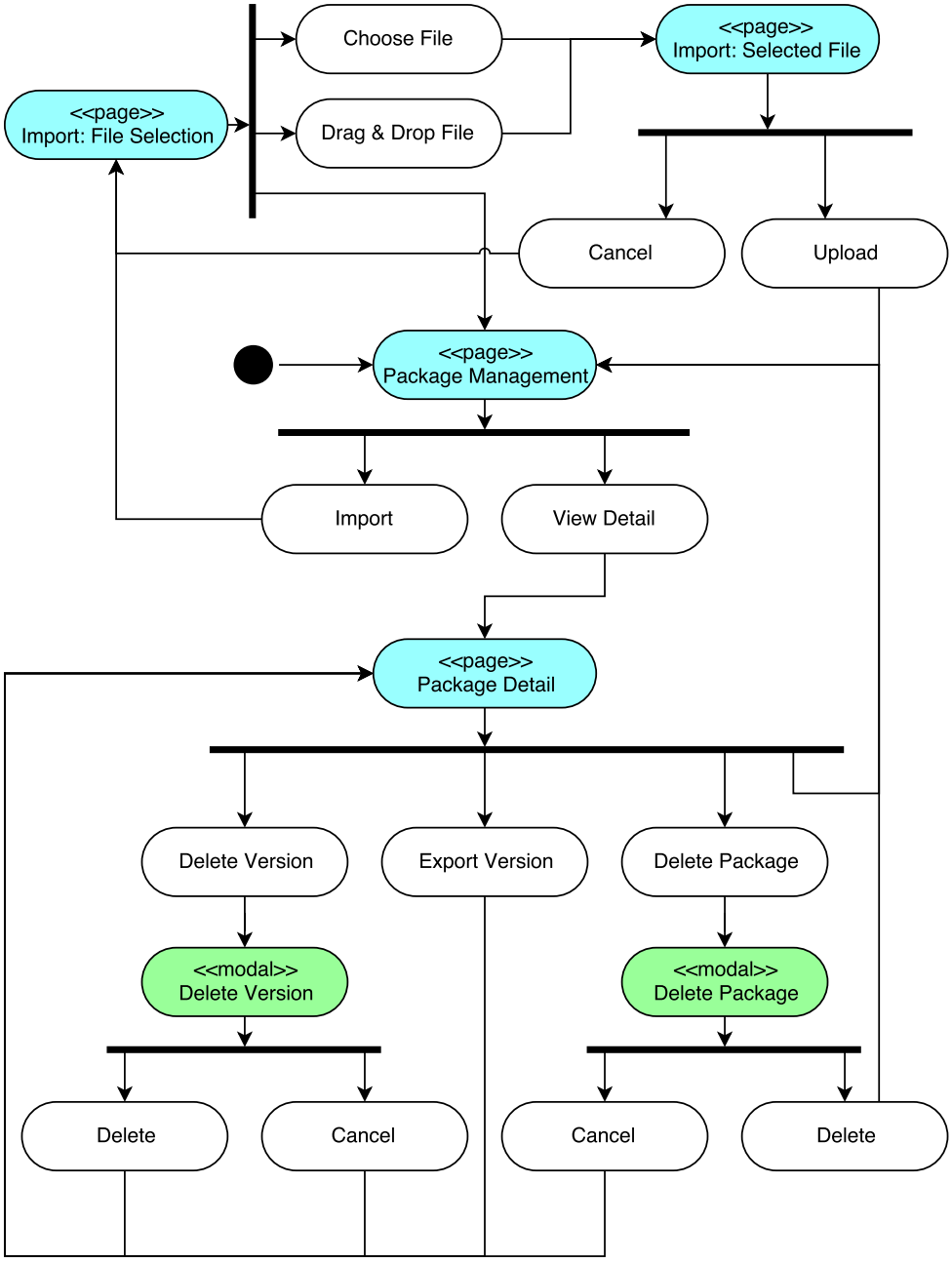


Figure 2.11: Package Management Module

Task list

- Upload – Upload the selected file and then move to the Package Management view.
- Cancel – Cancel the selection and return back to the File Selection.

Package Detail The detail screen shows information about the package and a list of its versions.

Task list

- Delete Package – Delete the whole Package with all its versions.
- Delete Version – Delete just a specific version of the Package.
- Export Version – Download the Package file that can be then imported to another portal.

Delete Package Modal Before the Package is deleted the user has to confirm it in this modal window.

Task list

- Delete – The whole package will be deleted.
- Cancel – Close the modal window.

Delete Version Modal Modal window where the user has to confirm deletion of a package version.

Task list

- Delete – Delete the selected version.
- Cancel – Close the modal window.

2.9 User Roles & Permissions

All the action in the system are allowed based on user permissions. User roles defines what permission the user has. Because of that it is easy to define new role as a set of different permissions if necessary.

2.9.1 Roles

We now have 3 roles defined in the system.

Admin This is the person responsible for managing the users and the organization settings.

Data Steward Data Steward is responsible for managing the Knowledge Models. He should edit them according to organization needs and keep them up to date.

Researcher Researcher is the person who is actually working on an experiment. He is using the Knowledge Models in wizards to generate Data Management Plans.

2.9.2 Permissions

There are several user permissions in the system that allows to perform different actions.

UM_PERM Permission that allows the user to access the User Management module and perform all the actions there.

ORG_PERM Permission that allows the user to edit the organization settings.

KM_PERM Allows the user to access the Knowledge Model module, create new and edit or delete existing knowledge models.

KM_UPRGADE_PERM Allows the user to start the migration and upgrade the Knowledge Model if the newer parent is available.

KM_PUBLISH_PERM Allows the user to publish a new version of the Knowledge Model which then becomes available in the Package Management module for export and in the Wizards module to be filled.

PM_PERM Allows the user to access the Package Management module and import/export knowledge model packages there.

WIZ_PERM Allows the user to access the Wizards module.

DMP_PERM Allows the user to access the Data Management Plans module.

Table 2.1: Roles and permissions matrix

	Admin	Data Steward	Researcher
UM_PERM	x		
ORG_PERM	x		
KM_PERM	x	x	
KM_UPRGADE_PERM	x	x	
KM_PUBLISH_PERM	x	x	
PM_PERM	x	x	
WIZ_PERM	x	x	x
DMP_PERM	x	x	x

Chapter 3

Implementation

3.1 Elm Language

Elm offers a lot of language features. Some of them are similar to Haskell, some of them are inspired by other functional languages.

3.1.1 Functions

Functions are core thing in Elm language. Elm has two types of functions – anonymous and named.

An anonymous function has a list of arguments and function body. a named function has a function signature that defines the function name and argument types.

Listing 3.1: Elm functions

```
1 -- Anonymous function
2 \x y -> x + y
3
4 -- Named function
5 add : Int -> Int -> Int
6 add x y = x + y
7
8 -- Function usage
9 add 1 2 == 3
```

3.1.2 Partial Function Application

Functions in Elm can be partially applied. We don't have to pass all the arguments to the function but only first few and the result is new function that takes the rest of the arguments.

Listing 3.2: Partial function application

```
1 -- using add function from previous example
2 increment : Int -> Int
3 increment = add 1
4
5 increment 5 == 6
```

3.1.3 Function Composition

Sometimes it is handy to create a function as a composition of multiple functions. Elm has operators `<<` and `>>` for that. Operators define the direction of composition.

Listing 3.3: Function composition

```
1 (f >> g) == (\x -> g(f(x)))
2 (f << g) == (\x -> f(g(x)))
```

3.1.4 Avoiding Parentheses

There can be a lot of parentheses in the code if we need to apply multiple functions . Elm has a backward function application `<|` and forward function application `|>` to avoid a lot of parentheses.

Listing 3.4: Backward function application [43]

```
1 -- with parentheses
2 leftAligned (monospace (fromString "code"))
3
4 -- with <|
5 leftAligned <| monospace <| fromString "code"
```

Listing 3.5: Forward function application [43]

```
1 -- with parentheses
2 scale 2 (move (10,10) (filled blue (ngon 5 30)))
3
4 -- with |>
5 ngon 5 30
6   |> filled blue
7   |> move (10,10)
8   |> scale 2
```

It is very common to see the usage of these operators in the Elm code. Especially `|>` is used quite often when processing some data and multiple functions need to be used. Therefore it is recommended to have the data as a last argument when designing functions.

3.1.5 Union Types

Union types (also called Algebraic Data Types in other functional languages) are extremely useful. They consist of a type name and constructors. The name is then use as a type in functions that are using it. Constructors are used to create new instances.

Listing 3.6: Union type example

```
1 -- union type definition
2 type Color = Red | Green | Blue
3
4 -- union type usage
5 redColor : Color
6 redColor = Red
```

Union type constructors can also have some data associated with them. Then we use the constructor as a function. In the following example, there is now new constructor for other color that has one string argument for the color hex code.

Listing 3.7: Union type constructor with data

```
1 type Color = Red | Green | Blue | Other String
2
3 yellowColor : Color
4 yellowColor = Other "#ffff00"
```

We can also use type variable to create a generic data structure. For example if we want to create a tree structure we don't want to specify what type should be saved in the tree. Instead, we use type variable and then we can define the used type when using the tree.

Listing 3.8: Union type with type variable

```
1 type Tree a = Node a (Tree a) (Tree a) | EmptyTree
2
3 intTree : Tree Int
4 intTree = Node 2 EmptyTree EmptyTree
```

3.1.6 Pattern Matching

Pattern matching is a common feature in functional languages. It consists of defining the patterns the data should match, checking which pattern they match and deconstructing the data using the pattern.

It is very common to use pattern matching with union types to find out which constructor has been used. If the constructor has some additional

data associated with it we can access them when deconstructing the data using the pattern.

Listing 3.9: Pattern matching with union type

```
getColorCode : Color -> String
getColorCode color =
    case color of
        Red -> "#ff0000"
        Green -> "#00ff00"
        Blue -> "#0000ff"
        Other code -> code
```

3.1.7 Maybe

`Maybe` is a generic union type that can be either `Just something` or `Nothing`. It is used when a function can return something but the operation might not be successful, e.g. getting the first element of a list – if the list is empty, it returns `Nothing` and if the list is not empty, it returns `Just <first element>`. It can also be used to handle optional values.

Listing 3.10: Definition of Maybe [44]

```
type Maybe a = Just a | Nothing
```

When working with Maybes we can use pattern matching to check if it is `Just value` or `Nothing`. This can be a lot of code that is usually very similar, therefore Elm comes few useful functions to work with Maybes.

In the following example we have a function that will return user image URL. However a user doesn't have to have the image. In that case we want to use the default image url. Elm has a function `Maybe.withDefault` that returns the value packed in Maybe or the default value in case of `Nothing`.

Listing 3.11: Example of Maybe.withDefault usage

```
-- using pattern matching
userImageUrl : Maybe String -> String
userImageUrl maybeUrl =
    case maybeUrl of
        Just url -> url
        Nothing -> "default.png"

-- using Maybe.withDefault
userImageUrl : Maybe String -> String
userImageUrl = Maybe.withDefault "default.png"
```

Another helpful function is `Maybe.map` that applies given function to a value wrapped with Maybe.

Listing 3.12: Maybe.map example [44]

```
1 Maybe.map sqrt (Just 9) == Just 3
2 Maybe.map sqrt Nothing == Nothing
```

When we have more functions that returns Maybe we can chain them together using `Maybe.andThen` function. If a Maybe contains a value `andThen` applies the given function to the value, otherwise it returns `Nothing` without using the function.

Listing 3.13: Example of chaining functions with Maybe.andThen

```
1  fn1 : Int -> Maybe String
2  fn1 = ...
3
4  fn2 : String -> Maybe String
5  fn2 = ...
6
7  result : List Int -> Maybe String
8  result lst =
9      head lst
10         |> Maybe.andThen fn1
11         |> Maybe.andThen fn2
```

3.1.8 Result

`Result` is another generic union type. It is used for computations that might not be successful and we want to return an error. `Result` can be either `Ok` and contains the result value or `Err` and contains the error.

Listing 3.14: Definition of Result [45]

```
1 type Result error value = Ok value | Err error
```

In Elm we have again `withDefault`, `map` and `andThen` functions in the `Result` module that works in a similar fashion as these functions for `Maybe`.

3.1.9 Record

Records are lightweight labeled data structures, similar to JavaScript objects. Records can contain data of different types. Record field values can be retrieved using accessor.

Record fields can be also updated. Since everything is immutable in Elm, the record is not really updated but a new record with updated value is created.

Listing 3.15: Record

```
-- creating a record
user = { name = "John", age = 21 }

-- accessing a record field
user.name  -- "John"

-- updating a record
{ user | age = 22 }  -- { name = "John", age = 22 }
```

When using records with a function, we need to write a record type annotation – name the fields and their types. Records can be also used with pattern matching to deconstruct the field values instead of getting the values with accessors.

Listing 3.16: Records usage

```
userLabel : { name : String, age : Int } -> String
userLabel {name, age} =
        name ++ ", " ++ age
```

3.1.10 Type Aliases

We can use type aliases to create a name for existing type that can be used within the type annotations. This is especially useful with records.

```
-- create type alias
type alias User =
    { name : String
    , age : Int
    }

-- usage
userLabel : User -> Html msg
userLabel user =
    user.name ++ ", " ++ user.age
```

Type aliases can be also used to create an alias for existing type and making the code more clear.

```
type alias Minutes = Int
type alias Seconds = Int

toSeconds : Minutes -> Seconds
toSeconds minutes =
    minutes * 60
```

3.2 The Elm Architecture (TEA)

The Elm Architecture is a pattern used by Elm applications to define the architecture. It is very good for modularity, refactoring, code reuse and testing. It is easy to keep even the complex applications clean and maintainable with the TEA.

The Elm application is divided into three main parts:

- **Model.** The state of the application.
- **Update.** How to change the state.
- **View.** How to display the state.

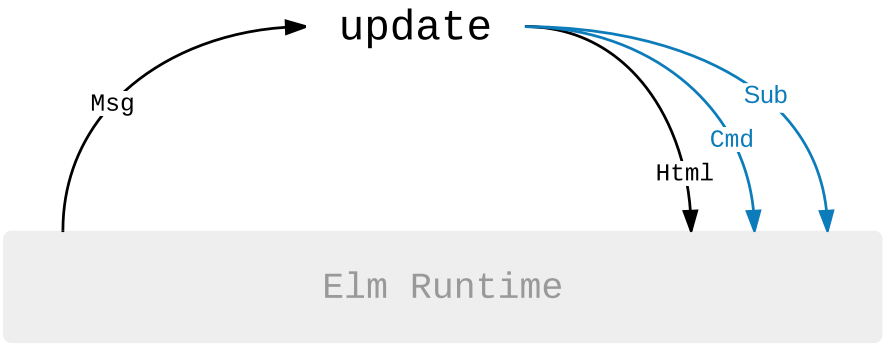


Figure 3.1: The Elm Architecture [46]

3.2.1 Model

Model is usually defined as a type alias for a specific type, usually a record type.

Listing 3.17: Example model

```
1 type alias Model =
2     { name : String
3     , age : Int
4     }
```

3.2.2 View

View in Elm is written in a declarative way. It simply takes the model and returns the Html type. Elm runtime is then responsible for the actual rendering of the HTML code.

Elm has a packages with functions for HTML elements and HTML attributes. HTML element functions has two parameters – the first one is a list of HTML attributes, the second one is a list of children HTML elements.

Listing 3.18: Example view

```
1 view : Model -> Html Msg
2 view model =
3     div [ class "container" ]
4         [ h1 [] [ text "My Elm Application" ]
5         , p [] [ text ("Hello " ++ model.name) ]
6         ]
```

3.2.3 User Input and Messages

The user input is handled in form of so called Messages. Messages are defined as a union type specific for the application.

Listing 3.19: Example messages definition

```
1 type Msg
2     = Increase
3     | Reset
```

View elements (e.g. buttons) can then produce messages.

Listing 3.20: Example of HTML button with message

```
1 button [ onClick Increase ] [ text "Increase" ]
```

3.2.4 Update

Update function maps message and model to a new model. In other words, it decides based on the message how to change the model.

Listing 3.21: Example of update function

```
1 -- Model is just Int in this example
2 type alias Model =
3     Int
4
5 update : Msg -> Model -> Model
6 update msg model =
```

```
7       case msg of
8           Increase ->
9               model + 1
10
11          Reset ->
12              0
```

3.2.5 Effects

Sometimes we need other input than just a user input from the application itself (e.g. HTTP request or using Web Sockets). There are two types of effects in Elm:

- **Commands.** Commands are used to do something (HTTP request, generate random number etc.).
- **Subscriptions.** Subscriptions are used to register that we are interested in something (time change, location change, Web Socket message etc.).

Both, Commands and Subscriptions are just the data that describes what to do. We are not doing anything, we just tell the Elm runtime what to do.

When we are using commands we need to change the update function to return not only the new model but also the command.

3.2.6 Commands Example

Let's see how the commands are used on an example with HTTP request. First, we define two messages.

The first one tells to get the user data and has user id string parameter. The second one is used when the get user profile request is complete. It has one parameter of Result type. Result can be either `Http.Error` when the request failed or `Ok <some type>` when the request is successful. In this example there is `User` type. The definition is not important but we can assume it is a record type with some user data.

Listing 3.22: Message type for getting user data

```
1 type Msg
2     = GetUser String
3     | GetUserComplete (Result Http.Error) User
```

Then we define `getUser` function that creates HTTP request. First we create the request itself using `Http.get` function that takes the url as a String and decoder. Decoder is a function that can convert the HTTP response into desired type, `User` in this case.

The `Http.send` function creates a command from the request. The first parameter is the message that should be used when the request is complete.

Listing 3.23: Example function to create HTTP request

```
1 getUser : String -> Cmd Msg
2 getUser userId =
3     let
4         request =
5             Http.get (apiUrl ++ userId) decodeUser
6     in
7     Http.send GetUserComplete request
```

The update function looks different now. Instead of just model, it returns a tuple with model and command. We can see that in case of `GetUser` message, it doesn't change the model but it creates a command using `getUser` function. On the other hand, when the request is complete and update function receives `GetUserComplete` message it updates the model (if there was no error) and returns `Cmd.none` (which means no command for the Elm runtime).

Listing 3.24: Example update function with commands

```
1  update : Msg -> Model -> ( Model, Cmd Msg )
2  update msg model =
3      case msg of
4          GetUser userId ->
5              ( model, getUser userId )
6
7          GetUserComplete (Ok newUser) ->
8              ( { model | user = newUser }, Cmd.none )
9
10         GetUserComplete (Err _) ->
11             -- handle the error here
12             ( model, Cmd.none )
```

3.2.7 Subscriptions Example

Let's say we want to do something every 10 seconds. In that case we can use subscriptions function. It takes one parameter – the model – and returns `Sub` type that Elm runtime can handle.

In this example, we use function `every` from the `Time` package. The first parameter is an interval how often we want updates, the second defines what kind of message we want to receive in update function.

It means that every 10 seconds `update` function will be called with `MyMsg` as a message.

Listing 3.25: Example subscriptions function

```
1 subscriptions : Model -> Sub Msg
2 subscriptions model =
3     Time.every (10 * second) MyMsg
```

It is of course possible to use different subscriptions based on model. For example, we want to receive time updates only if `foo` property of the model is true.

Listing 3.26: Example subscriptions function

```
1 subscriptions : Model -> Sub Msg
2 subscriptions model =
3     if model.foo then
4         Time.every (10 * second) MyMsg
5     else
6         Sub.none
```

3.3 Project Structure in Elm

As we could see in the previous section, the Elm Architecture offers a solid way to split the application logic into appropriate parts and put everything where it belongs.

However, the Data Stewardship portal is very complex and having for example all update code in a single update function would not be the best solution. Luckily, the Elm Architecture is perfect for modularity. We can simply create separate functionality for models, view and update for each module and connect them in their top level alternatives. Let's see an example of how this could work.

3.3.1 Model

Each module has its own model and the top level model consists of some top level data (e.g. current route or logged in user information) and the data for each nested module.

The nested model simply contains the data needed for the module itself.

Listing 3.27: Nested model

```
1 module NestedModule.Models exposing (..)
2
3 type alias Model =
4     { name : String
5     , age : Int
6     }
```

Then the top level model imports all the nested models and includes them. Besides other top level app data, it also contains a route. Route defines what page in the browser is opened. It will be necessary for other parts of the application.

Listing 3.28: Top level model

```
module Models exposing (..)

import NestedModule.Models

type alias Model =
    { -- some top level app data
    , route : Route
    , nestedModuleModel : NestedModule.Models.Model
    }
```

3.3.2 Messages

The nested module contains simple definition of messages that are used by the module's update function.

Listing 3.29: Nested module messages

```
module NestedModule.Msgs exposing (..)

type Msg
    = Message1
    | Message2
```

The top level messages import all the messages from the nested modules. The top level `Msg` type has constructor that wraps all the messages types from the nested modules. This will be important for the top level update function.

Listing 3.30: Top level messages

```
module Msgs exposing (..)

import NestedModule.Msgs

type Msg
    = NestedModuleMsg NestedModule.Msgs.Msg
    -- messages from other modules
    -- and some top level messages
```

3.3.3 Update

The update function in the nested module takes the `Msg` and `Model` type from the nested module and produces a tuple with new nested model and command. The important thing is that it returns command with top level message and not a module message. This is necessary because this way the module can produce also commands that are changing the global application state (e.g. changing the application route).

Listing 3.31: Nested module update function

```
1  module NestedModule.Update exposing (..)
2
3  import Msgs
4  import NestedModule.Models exposing (Model)
5  import NestedModule.Msgs exposing (Msg(..))
6
7  update : Msg -> Model -> ( Model, Cmd Msgs.Msg )
8  update msg model =
9      case msg of
10         Message1 ->
11             -- do something and return model and cmd
12             ( model, cmd )
13
14         -- Handle other module messages
```

The top level update function simply decides based on the message type which module update function should be used and send the module message and module model to the correct update function. Then it updates the nested model in the top level model and returns a new top level model with a command returned from the module update function.

Listing 3.32: Top level update function

```
1  module Update exposing (..)
2
3  import Msgs exposing (Msg(..))
4  import NestedModule.Update
5
6  update : Msg -> Model -> ( Model, Cmd Msg )
7  update msg model =
8      case msg of
9          NestedModuleMsg nestedModuleMsg ->
10             let
11                 ( newModel, cmd ) =
12                     NestedModule.Update.update
13                         nesteModuleMsg
```

```
14                          model.nestedModuleModel
15              in
16              ( { model
17                    | nestedModuleModel = newModel
18                }
19              , cmd
20              )
21
22          -- Handle msgs from other modules
23          -- and other top level messages
```

3.3.4 View

The view from the nested module knows what to render based on the model from the nested module. It doesn't know anything about top level model or view. The Html it returns is also using the top level message instead of module message. It is because some elements can produce messages that can change the global application state (e.g. link to a different view).

Listing 3.33: Nested module view

```
1  module NestedModule.View exposing (..)
2
3  import Msgs
4  import NestedModule.Models exposing (Model)
5  -- other imports for Html, etc.
6
7  view : Model -> Html Msgs.Msg
8  view model =
9      div []
10         [-- code handling what should be
11          -- visible in the module view
12         ]
```

The top level view imports views from all the nested modules. It uses the top level model and based on the route it decides what nested view should be used. Then it sends only the nested model to the nested view. The nested view renders only the content of the module itself. The result is then send to `appView` function which wraps it with the app layout, menu etc.

Listing 3.34: Top level view

```
1  module View exposing (..)
2
3  import Models exposing (Model)
4  import Msgs
```

```
5  import NestedModule.View
6  -- other imports for Html, etc.
7
8  view : Model -> Html Msg
9  view model =
10     case model.route of
11         NestedRoute ->
12             model.nestedModuleModel
13                 |> NestedModule.View.view
14                 |> appView
15
16         -- Views for other routes
```

3.4 Development & Build Tools

The development of Elm application is very easy and convenient. There is a webpack loader for Elm[47] which means we can simply use webpack to build everything from compiling Elm into JavaScript to compiling CSS styles from SCSS. We create so called bundle using webpack which is the bundled application ready to be used in the browser.

There is also webpack dev server[48] with a live reload feature. The dev server is using the same configuration as the webpack is using for building the application. It starts a local server which serves the bundled application. When something is changed in source files, the application is automatically rebuild and reloaded in the web browser.

Elm loader is using elm-make[49] to build elm into JavaScript and elm-package[50] to download additional Elm packages needed for the application build.

All of these tools can be simply installed using npm. Project dependencies that should be installed from npm are defined in a file called `package.json`.

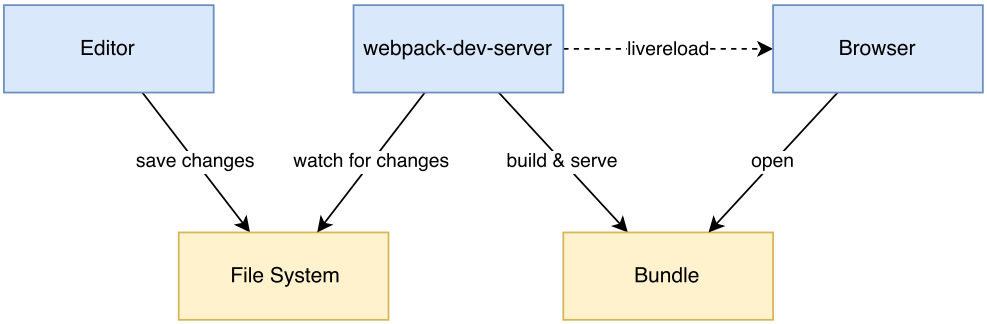


Figure 3.2: Development & Build Tools

Elm has its own file called `elm-package.json` that defines the Elm dependencies. With these two files it is simple to install all the project dependencies or build the application with the single command.

Chapter 4

Testing and Deployment

This chapter will be about different kind of tests that are used in the Data Stewardship Portal. Then I will analyze how good the UI of the portal is. After that I will talk about how the portal is deployed and how the deployment process works.

4.1 Unit Tests

Unit testing is an essential part of software development. It focuses on testing the smallest units of source code like functions or objects. Sometimes, it is necessary to simulate the context using mocks.

However, it is quite simple in Elm. The source code contains only pure functions, so there is no need for mocking the context. We only need to pass some parameters to the function that is being tested and check if the output is as expected.

Listing 4.1: Example test in elm-test [51]

```
1  suite : Test
2  suite =
3      describe "The String module"
4          [ describe "String.reverse"
5              [ test "reverses a known string" <|
6                  \_ ->
7                      "ABCDEFG"
8                          |> String.reverse
9                          |> Expect.equal "GFEDCBA"
10             , fuzz string "restores the origi..." <|
11                 \randomString ->
12                     randomString
13                         |> String.reverse
14                         |> String.reverse
```

```
15                              |> Expect.equal randomString
16                ]
17            ]
```

Elm has a library called `elm-test`[51] that provides some functions for writing unit tests and also a test runner. There are 3 most important functions provided by the `elm-test`:

- `describe` adds description to a list of tests. It can be also nested to create clear structure of tests.
- `test` simply runs a test which is a function that returns `Expectation` that is evaluated when the test is run.
- `fuzz` is similar to `test`, however, the function is run several times with randomly generated input.

Using these functions we can create tests for everything we need. The runner can be used from the command line and is easy to integrate into continuous integration solutions.

4.2 End-to-End Tests

Unit tests covered the smallest units of the code. However, it is also important to test that the flows in the application works from the start to the end from the user perspective as expected. This type of tests are called end-to-end tests.

I decided to use Test Scenarios for testing the end-to-end functionality of the application flows. The scenario simply describe step by step what the user and the system are doing during the activity and what is the expected outcome.

The scenarios are based on the use cases for each module. They should cover all the possible actions shown in the activity diagrams.

They consist of the following parts:

The Goal. What is the user trying to achieve in the scenario. E.g. *User wants to log in.*

Precondition. What are the preconditions for the scenario, whether we should be in a specific application state or the user should have specific role etc. E.g. *User is not logged in the portal.*

Main Success Scenario. This part describe step by step the scenario if everything is going well. Example:

Step	Actor	Action Description
1	User	User navigates to the login page
2	User	User fills in his email and password and clicks the Login button
3	System	System logs the user in and redirect him to the Homepage

Scenario Extensions. This part can extend the steps from the Main Success Scenario. It can describes what happens when some specific condition happens, user fills in invalid data or clicks different button than the one that leads to the successful end etc. Example:

Step	Condition	Action Description
3a	User filled in incorrect combination of username and password	User is not logged in and an error message is shown instead

Success End Condition. This part describes what should be the state of the application if the scenario was successful. E.g. *User is logged in and see the homepage.*

The test scenarios are described in Appendix C.

4.3 UI Heuristic Evaluation

I will use 10 Usability Heuristics for User Interface Design by Jakob Nielsen[52] to analyze the user interface of the Data Stewardship Portal and find out what could be improved.

4.3.1 Visibility of system status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

- When getting the data from or posting to the server and it takes long time, the loading indicator is shown.
- All the pages contain a title describing the user where he is, also the appropriate menu item is highlighted.

Possible flaws

- During the Migration there is no information about which Knowledge Model and to what version is migrating.

4.3.2 Match between system and the real world

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

- The system is using the terminology already used within the Knowledge Models and everything should make sense to the users.

4.3.3 User control and freedom

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

- All forms and modal windows has also a cancel button so the user can just step back.
- Confirmation is always required before something is deleted.
- User can always use the menu to navigate back where he wants.

4.3.4 Consistency and standards

Users should not have to wonder whether different words, situations, or actions mean the same thing.

- There is always the same terminology and/or same icons for the same actions regardless the entity.
- Also the UI of the same type of the pages for different entities has always the same layout.

4.3.5 Error prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

- When the user is importing a new version to the portal, the file has to be sent to the server first to find out whether the file is valid.
- When the user is entering values in forms, he can fill in invalid values. The error message is shown immediately before the form is submitted.
- Some values from the forms has to be validated on the server after the form submission, e.g. whether the email address is used in the portal.

- Besides the previous cases, there are no other situations where the user should see an error message.

4.3.6 Recognition rather than recall

Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

- Every information the user need for the task he wants to do should be always visible on the screen. E.g. when the user is publishing a new version of the knowledge model, he can also see what was the last published version.

Possible flaws

- The problem that could be here was already mentioned before – during the Migration there is no information about which Knowledge Model and to what version is migrating. User has to remember that from the list view where he started the migration.

4.3.7 Flexibility and efficiency of use

Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

- Most of the actions are reachable in two clicks, all of them in three clicks. It should be fast enough to do anything in the system.

Possible improvements

- There could be some keyboard shortcuts to eliminate the need to use the mouse whenever possible.

- There could be a customizable dashboard at the homepage where the user could set up some widgets to bring the information he needs the most to the homepage.

4.3.8 Aesthetic and minimalist design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

- Every screen contains only the information that is needed for the task within the screen.

- Buttons for primary actions has always distinctive color to catch user's attention. There is maximum of one primary action per screen.

4.3.9 Help users recognize, diagnose, and recover from errors

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Possible flaws

- Error messages used in the forms could be more descriptive, now it shows only e.g. *InvalidEmail*.

- When the server returns an error the client just show a general error without trying to read the error returned by the server. It should show the error from the server whenever possible.

4.3.10 Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Possible flaws

- Data Stewardship Portal has no user documentation. It should be easy enough to use for the users though.

4.3.11 Conclusion

The user interface of the Data Stewardship Portal is following most of the points with minor problems. The future improvements should include:

- Improve Migration screen to include the information about the Knowledge Model.

- Error messages should more precisely and descriptively reflect what happened.

- The portal should have user documentation.

4.4 Deployment

4.4.1 Docker

The application is deployed using Docker[53]. Docker allows to bundle applications into images that defines all the dependencies and requirements the application has – operating system, installed packages, etc. The image contains everything needed for the application to run properly. Docker can start containers from the images with a specific configuration (e.g. environment variables).

Containers doesn't have any persistent data inside them. That means they can be safely restarted or removed and started again without loosing the data. If we need to store some data permanently, we need to use a volume. Volumes can be mounted into containers and their data are stored on the host filesystem. When the container is removed, a new one can be started with the same volume.

It is also easy to scale dockerized application. Simply more containers can be started with a load balancer in front of them. Docker containers can be used not only on a single host but also with various infrastructure tools like Kubernetes[54] or Google Cloud Platform[55]. If there is a need for that in the future, it will be easy and smooth to do that because the images will be already prepared.

4.4.2 DSP Client Docker Image

The Docker image for DSP Client is quite simple. The whole application is just a couple of static files (JavaScript scripts, CSS styles, images, etc.). We just need something to serve those files. The image is based on nginx[56]. It is a web server with high performance and low requirements. After the source of the DSP Client is build it is copied into the image and served by nginx.

4.4.3 Reverse Proxy

DSP Client is using simple HTTP protocol for communication. However, to connect it to the outside world we should use HTTPS to make the connection secure with encrypted requests.

There will be more services running on the same Docker host (e.g. the whole backend for DSP). So it is a good idea to separate the HTTPS communication and the certificates management from the applications.

We can configure another nginx as a reverse proxy. It will be the only service that will be exposed to the outside world. Its responsibility will be providing a secure HTTPS connection and pass requests through to the requested services.

The other services are available only in the internal Docker network and cannot be reached from outside directly, only through the reverse proxy.

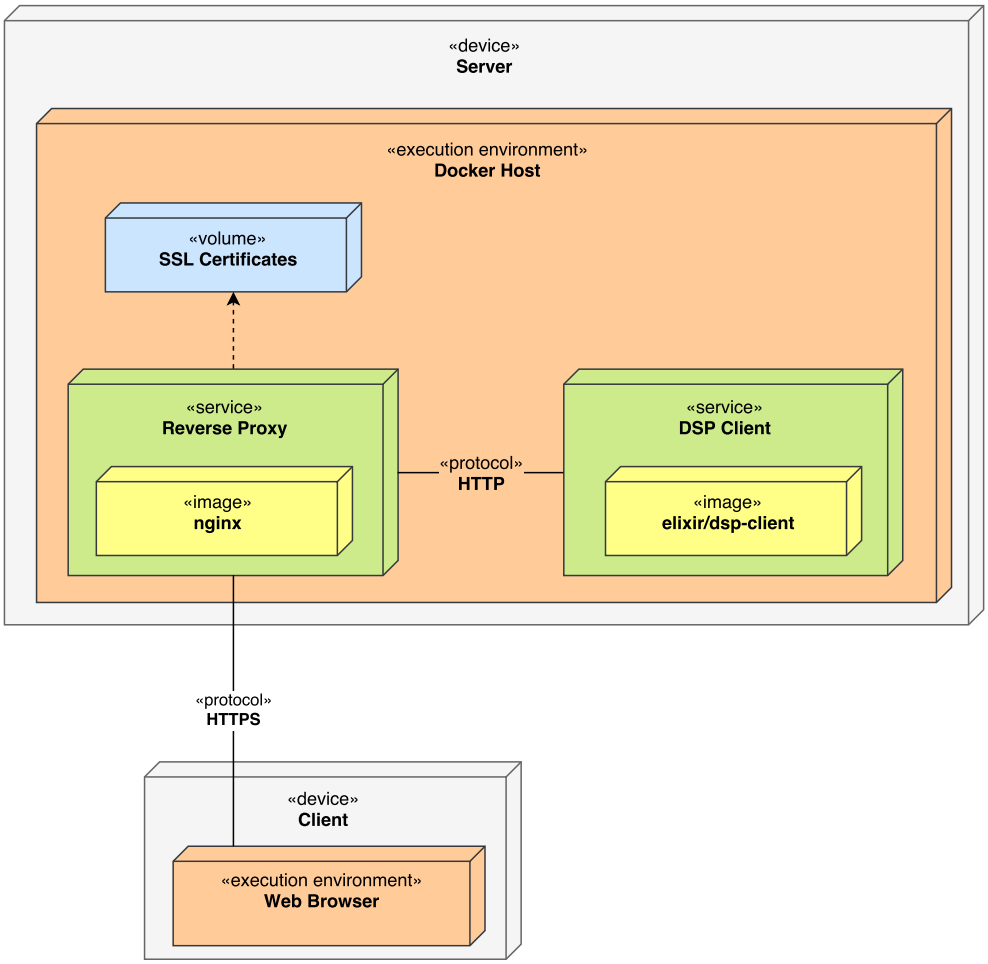


Figure 4.1: Deployment Diagram

Therefore we don't need to configure HTTPS and handle certificates for each service, we can simply use HTTP within the internal network.

The HTTPS certificates are generated using Let's encrypt service[57]. They are stored in the volume on the host filesystem and are mounted into the reverse proxy container and used by nginx.

4.5 Deployment Process

Git[58] is used as a source control management tool. The source code is stored in Github[59]. It is a service that supports software development with git and is used by most open source projects.

Travis CI[60] is a service for continuous integration that can be connected with a Github repository. We have to create a configuration file for Travis CI

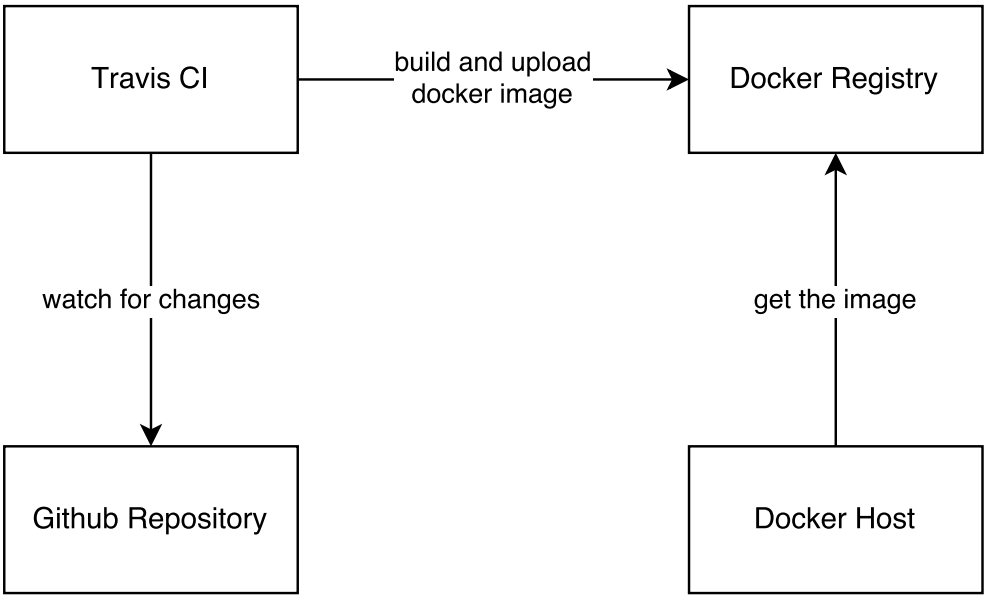


Figure 4.2: Deployment Process

describing what tasks should be performed in order to build, test and deploy the project. Travis CI then watches for the changes in the Github repository and perform the tasks.

The whole project is built into a Docker image, so Travis CI run tests, build the Elm source code into JavaScript and creates the Docker image. When the image is created it is pushed into a Docker registry. Docker registry is a service where Docker images can be pushed into and pulled from. We use our own private Docker registry for all Elixir DSP images so the frontend image is also pushed there.

The server where the whole project runs is using Docker to run all the images. It can get all the images from our private Docker registry and run them with appropriate configuration.

Chapter 5

Results

I will describe the state of the Data Stewardship Portal and what features it provides now. Then I will look in the future of the project – what will be the next steps and what could be done to improve the current state.

5.1 Current State

The Data Stewardship Portal has now well-defined and manageable architecture in Elm language using TEA. It is very easy to add new parts and keep everything clear.

The portal contains essential parts to be used by organizations and their members – it contains organization and user management modules. Users can be added with appropriate roles to have access to the parts of the portal they need to.

All the functionality needed for working with Knowledge Models has been developed. The portal contains Knowledge Model editor where the data stewards can make changes. Knowledge Models can be exported out of and import into the portal, also between different organizations. When a new parent Knowledge Model appears in the portal, it is possible to migrate local knowledge models dependent on it to apply all the new changes.

To sum it up, the functionality needed for portal administrators and data stewards has been created and is ready to use.

The portal is ready to be used in production. It is distributed in form of Docker image. The project is tested and and the image is build automatically using Travis CI.

5.2 Project Future

This theses was just the beginning for the Data Stewardship Portal. The development of the portal will continue and it will eventually help scientists

to plan the data management for their experiments.

5.2.1 New Modules

The portal now handles the organization and user management and all the processes with the knowledge models. There are two more modules planned for the researchers to plan their projects:

- **Wizards module** – for answering the questions from the knowledge models.
- **Data Management Plans module** – for generating data management plans from filled wizards.

5.2.2 Usability Improvements

There are some usability details that can be improve in the current state. UI heuristic evaluation revealed few flaws:

- Add more details about the Knowledge Model to the Migration view
- Make error messages more understandable for the user
- Create user documentation

The project is now targeting desktop browsers but there might be the need to make it also mobile friendly.

5.2.3 Other Improvements

Currently most of the deployment process and testing is automated. The only thing that needs to be done manually are the test scenarios. It would be good to automate them and include them to the deployment process.

Conclusion

I acquainted myself with the Data Stewardship Portal project, analyze its requirements and requested functionality.

I did a review of state-of-the-art Haskell based solution for web frontend development, considered their advantages and disadvantage and chose the most suitable one for the Data Stewardship Portal.

I designed and implemented the architecture of the web frontend for the DSP using the selected solution. Then I designed and implemented a web UI and functionality of the Knowledge Model Editor and Migration Module within the portal.

I elaborated a technical documentation to make it easy for other developers to collaborate on the project and prepared generated API doc.

I created tests and test scenarios for the solution and evaluated the quality of the UI. I described the results and outlined what the future of the project should be.

All the goals and assigned tasks were fulfilled and the project is ready for future development.

Appendix A

Acronyms

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CI Continuous Integration

CSS Cascading Style Sheets

DOM Document Object Model

DSP Data Stewardship Portal

FFI Foreign Function Interface

GHC Glasgow Haskell Compiler

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

JSON JavaScript Object Notation

REPL Read-Eval-Print Loop

TEA The Elm Architecture

UI User Interface

UML Unified Modeling Language

UUID Universally Unique Identifier

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `dsp-client` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `assignment.pdf` ..... the assignment in PDF format
  - `thesis.pdf` ..... the thesis text in PDF format

Appendix C

Test Scenarios

C.1 Login

C.1.1 Login and menu items

User wants to login and should see the correct menu items based on his role.

Precondition

User is not logged in the portal.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to the login page
2	User	User fills in his email and password and clicks the Login button
3	System	System logs the user in and redirect him to the Homepage

Scenario Extensions

Step	Condition	Action Description
3a	User filled in incorrect combination of username and password	User is not logged in and an error message is shown instead

Success End Condition

User is logged in and see the correct menu items based on his role.

- **Admin**
  - Organization
  - User Management

  - Knowledge Models
  - Package Management
  - Wizards
  - Data Management Plans

- **Data Steward**
  - Knowledge Models
  - Package Management
  - Wizards
  - Data Management Plans

- **Researcher**
  - Wizards
  - Data Management Plans

C.2 Organization Module

C.2.1 Edit Organization

User wants to edit organization details.

Precondition

User is logged in and has Admin role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to the Organization
2	User	User fills in Organization name and Organization Group ID
3	User	User clicks the Save button
4	System	System updates the Organization detail and shows the success message

Scenario Extensions

Step	Condition	Action Description
2a	User leaves some fields empty	Error message below the field is shown and the form cannot be submitted
2b	User fills in invalid Group ID	Error message below the field is shown and the form cannot be submitted

Success End Condition

The organization detail is updated and the success message is shown.

C.3 User Management

C.3.1 Create User

User wants to create a new user profile.

Precondition

User is logged in and has Admin role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to the User Management and clicks Create User
2	User	User fills in the following fields: Email, Name, Surname, Password; and selects a Role for the new user
3	User	User clicks the Save button
4	System	System creates a new user profile

Scenario Extensions

Step	Condition	Action Description
2a	User leaves some fields empty or fills them with invalid value	Error message below the field is shown and the form cannot be submitted
3a	User clicks the Cancel button	No new user is created and user is redirected back to the User Management
4a	Email is already used by another user	New user profile is not created and an error message is shown

Success End Condition

New user profile is created, can be seen in the user list and can log in.

C.3.2 Delete User

User wants to delete an existing user profile.

Precondition

User is logged in and has Admin role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to the User Management where he can see the list of user profiles
2	User	User clicks the Delete button in the row of the user profile he wants to delete
3	System	System shows a modal window with the confirmation of deleting the profile
4	User	User clicks the Delete button in the modal window
5	System	System deletes the user profile and shows the success message

Scenario Extensions

Step	Condition	Action Description
4a	User clicks the Cancel button	Confirmation modal window is closed and no user profile is deleted

Success End Condition

User profile is deleted, it is no longer visible in the user list and cannot be used to log in the portal.

C.3.3 Edit User Profile

User wants to change an existing user profile.

Precondition

User is logged in and has Admin role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to the User Management where he can see the list of user profiles
2	User	User clicks the Edit button in the row of the user profile he wants to edit
3	System	System shows a form with user profile data
4	User	User updates Email, Name, Surname and/or Role of the user profile
5	User	User clicks the Save button
6	System	System updates the user profile and shows a success message

Scenario Extensions

Step	Condition	Action Description
4a	User leaves some fields empty or filled with invalid value	Error message below the field is shown and the form cannot be submitted
6a	Email is already used by another user	Profile is not updated and an error message is shown

Success End Condition

User profile is updated and all changes can be seen in the user list.

C.3.4 Edit User Password

User wants to change the password of an existing user profile.

Precondition

User is logged in and has Admin role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to the User Management where he can see the list of user profiles
2	User	User clicks the Edit button in the row of the user profile he wants to edit
3	System	System shows user profile with tabular navigation between editing user profile and user password
4	User	User selects the Password tab
5	System	System shows a form for editing user password
6	User	User fills in New password and New password again fields
7	User	User clicks the Save button
8	System	System updates the password of the user profile and show success message

Scenario Extensions

Step	Condition	Action Description
6a	Passwords in the fields are different	An error message is shown and the form cannot be submitted

Success End Condition

Password is changed and the edited user profile can log in with the new password.

C.3.5 Edit Own Profile

User wants to edit his own profile.

Precondition

User is logged in.

Main Success Scenario

Step	Actor	Action Description
1	User	User clicks his name in the menu
2	System	System shows a form with user profile data
3	User	User updates his Email, Name and Surname
4	User	User clicks the Save button
5	System	System updates the user profile and shows a success message

Scenario Extensions

Step	Condition	Action Description
3a	User leaves some fields empty or filled with invalid value	Error message below the field is shown and the form cannot be submitted
5a	Email is already used by another user	Profile is not updated and an error message is shown

Success End Condition

User profile is updated.

C.3.6 Edit Own Password

User wants to change his password.

Precondition

User is logged in.

Main Success Scenario

Step	Actor	Action Description
1	User	User clicks his name in the menu
2	System	System shows user profile with tabular navigation between editing user profile and user password
3	User	User selects the Password tab
4	System	System shows a form for editing user password
5	User	User fills in New password and New password again fields
6	User	User clicks the Save button
7	System	System updates the password of the user profile and show success message

Scenario Extensions

Step	Condition	Action Description
5a	Passwords in the fields are different	An error message is shown and the form cannot be submitted

Success End Condition

Password is changed and user can log in with the new password.

C.4 Knowledge Models

C.4.1 Create Knowledge Model

User wants to create a new knowledge model.

C.4.2 Precondition

User is logged in and has Admin or Data Steward role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Knowledge Models
2	User	User clicks Create KM button
3	System	System shows a form for new Knowledge Model
4	User	User fills in Name and Artifact ID for the Knowledge Model and can select Parent Package
5	User	User clicks the Save button
6	System	System creates new Knowledge Model and shows the Editor
7	User	User can use the editor and clicks the Save button
8	System	System updates the Knowledge Model according to the changes in the Editor

Scenario Extensions

Step	Condition	Action Description
4a	User leaves some fields empty or fill in an invalid value	Error message below the field is shown and the form cannot be submitted
7a	User clicks the Cancel button	No new changes to the Knowledge Model are saved

Success End Condition

New Knowledge Model is created and is visible in the Knowledge Models list.

C.4.3 Edit Knowledge Model

User wants to edit an existing Knowledge Model.

C.4.4 Precondition

User is logged in and has Admin or Data Steward role. Knowledge Model is in one of the following states: Default, Edited, Outdated.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Knowledge Models
2	System	System shows a list of Knowledge Models
3	User	User clicks Edit button for the Knowledge Model he wants to edit
4	System	System shows the Knowledge Model Editor
5	User	User makes changes in the Editor and clicks the Save button
6	System	System updates the Knowledge Model according to the changes in the Editor

Scenario Extensions

Step	Condition	Action Description
5a	User clicks the Cancel button	No new changes to the Knowledge Model are saved

Success End Condition

Knowledge Model is updated according to the changes made in the Editor and is now in the Edited state.

C.4.5 Upgrade Knowledge Model

User wants to upgrade the Knowledge Model to a newer version of parent.

Precondition

User is logged in and has Admin or Data Steward role. Knowledge Model is in the Outdated state.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Knowledge Models
2	System	System shows a list of Knowledge Models
3	User	User clicks the Upgrade button for the Knowledge Model he wants to upgrade
4	System	System shows a modal window with new parent versions available
5	User	User selects one of the new parent versions
6	User	User clicks Create button
7	System	System creates a new migration and redirects the user to the migration
8	System	System shows one change after another during the migration
9	User	User selects if he wants to Apply or Reject the change to his Knowledge Model
10	System	System shows that the migration is completed once the last change is processed

Scenario Extensions

Step	Condition	Action Description
6a	User clicks Cancel button	The modal window is closed and no migration is created
9a	User leaves the page in the middle of migration	Knowledge Model stays in the Migrating state and user can continue the migration later

Success End Condition

Knowledge Model is parent is upgraded and all the changes selected during the migration are applied to the Knowledge Model. It is no longer in the Outdated state.

C.4.6 Delete Knowledge Model

User wants to delete an existing Knowledge Model.

C.4.7 Precondition

User is logged in and has Admin or Data Steward role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Knowledge Models
2	System	System shows a list of Knowledge Models
3	User	User clicks Delete button for the Knowledge Model he wants to delete
4	System	System shows the confirmation modal window
5	User	User clicks Delete button
6	System	System deletes the Knowledge model from the portal

Scenario Extensions

Step	Condition	Action Description
3a	User clicks the Cancel button	The modal window is closed and Knowledge Model is not deleted

Success End Condition

Knowledge Model is deleted from the portal and is no longer visible in the Knowledge Models list.

C.4.8 Publish Knowledge Model Version

User wants to publish a new version of the Knowledge Model.

Precondition

User is logged in and has Admin or Data Steward role. Knowledge Model is in one of the following states: Edited, Migrated.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Knowledge Models
2	System	System shows a list of Knowledge Models
3	User	User clicks Publish button for the Knowledge Model he wants to publish
4	System	System shows a form for the new version
5	User	User fills in the new version number and the description
6	User	User clicks the Publish button
7	System	System creates a new version for the Knowledge Model Package

Scenario Extensions

Step	Condition	Action Description
5a	User fills in invalid values or leaves the fields empty	An error message is shown below the field and the form cannot be submitted
6a	User clicks the Cancel button	User is returned back to the Knowledge Model list and nothing is created

Success End Condition

New Knowledge Model Package version is available in the Package Detail. Knowledge Model is in the Default state.

C.5 Package Management

C.5.1 Import

User wants to import package version to the system.

Precondition

User is logged in and has Admin or Data Steward role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Package Management and clicks the Import button
2	System	System shows the Import package screen
3	User	User clicks Choose file button and select the appropriate file
4	System	System shows the file details so that user can confirm the import
5	User	User clicks Upload button
6	System	System imports the version from a file to the system

Scenario Extensions

Step	Condition	Action Description
3a	User uses drag and drop to drop a file from the filesystem to the drop area	Scenario continues the same way to step 4
5a	User clicks Cancel button	Scenario returns back to step 2
6a	User selected invalid file	Nothing is imported and an error message is shown, scenario gets back to step 4
6b	User selected version that is already present in the portal	Nothing is importend and an error message is shown, scenario gets back to step 4

Success End Condition

New package version is imported to the portal. If the whole package was not yet present in the portal it can now be seen in the package list. The version is visible in the package detail.

C.5.2 Delete Version

User wants to delete a specific version from the portal.

Precondition

User is logged in and has Admin or Data Steward role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Package Management
2	System	System shows a list of packages
3	User	User selects the package whose version he wants to delete
4	System	System shows the package detail view with the list of versions
5	User	User clicks the Delete button for the version he wants to delete
6	System	System shows the confirmation modal
7	User	User clicks the Delete button
8	System	System deletes the version

Scenario Extensions

Step	Condition	Action Description
7a	User clicks the Cancel button	The confirmation modal is closed and nothing is deleted

Success End Condition

User is at the package detail and the deleted version is no longer there.

C.5.3 Delete Package

User wants to delete the whole package from the portal.

Precondition

User is logged in and has Admin or Data Steward role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Package Management
2	System	System shows a list of packages
3	User	User selects the package he wants to delete
4	System	System shows the package detail
5	User	User clicks the Delete package button
6	System	System shows the confirmation modal
7	User	User clicks the Delete button
8	System	System deletes the package and redirects the user to the package list

Scenario Extensions

Step	Condition	Action Description
7a	User clicks the Cancel button	The confirmation modal is closed and nothing is deleted

Success End Condition

User package and all its versions are deleted and are no longer in the package list.

C.5.4 Export Package Version

Precondition

User is logged in and has Admin or Data Steward role.

Main Success Scenario

Step	Actor	Action Description
1	User	User navigates to Package Management
2	System	System shows a list of packages
3	User	User selects the package whose version he wants to export
4	System	System shows the package detail
5	User	User clicks Export button for the version he wants to export
6	System	System returns the file that is downloaded by the browser

Success End Condition

The package file is downloaded.

Bibliography

[1] ELIXIR. *What we do* [online]. [cit. 2018-01-03]. Available from: https://www.elixir-europe.org/about-us/what-we-do

[2] ELIXIR CZ. *ELIXIR CZ Interoperability Platform*. [cit. 2018-01-03]. Available from: https://github.com/DataStewardshipPortal

[3] LIEBERMAN, Benjamin. *UML Activity Diagrams: Detailing User Interface Navigation* [online]. 2004-04-29 [cit. 2018-01-05]. Available from: https://www.ibm.com/developerworks/rational/library/4697.html

[4] HASKELL COMMUNITY. *Haskell* [software]. [cit. 2017-12-04]. Available from: https://www.haskell.org

[5] FACEBOOK, Inc. *React* [software]. [cit. 2017-12-04]. Available from: https://reactjs.org

[6] THE GLASGOW HASKELL TEAM. *The Glasgow Haskell Compiler* [software]. [cit. 2017-12-04]. Available from: https://www.haskell.org/ghc/

[7] HASKELL COMMUNITY. *Hackage* [software]. [cit. 2017-12-04]. Available from: https://hackage.haskell.org

[8] CABAL DEVELOPMENT TEAM. *Cabal* [software]. [cit. 2017-12-04]. Available from: https://www.haskell.org/cabal/

[9] NPM, Inc. *npm* [software]. [cit. 2017-12-04]. Available from: https://www.npmjs.com

[10] BOWER TEAM. *Bower* [software]. [cit. 2017-12-04]. Available from: https://bower.io

[11] WEBPACK CONTRIBUTORS. *webpack* [software]. [cit. 2017-12-04]. Available from: https://webpack.github.io

[12] GOOGLE, Inc. *Google Closure Compiler* [software]. [cit. 2017-12-04]. Available from: https://github.com/google/closure-compiler

[13] EKBLAD, Anton. *Haste* [software]. [cit. 2017-12-19]. Available from: https://haste-lang.org

[14] FAYLANG. *Fay programming language* [software]. [cit. 2017-12-19]. Available from: https://github.com/faylang/fay

[15] FAYLANG. *Fay programming language – Quick Start* [online]. [cit. 2017-12-19]. Available from: https://github.com/faylang/fay/wiki

[16] GHCJS CONTRIBUTORS. *GHCJS* [software]. [cit. 2017-12-20]. Available from: https://github.com/ghcjs/ghcjs

[17] TRINKLE, Ryan. *Reflex* [software]. [cit. 2017-12-20]. Available from: https://github.com/reflex-frp/reflex

[18] TRINKLE, Ryan. *Reflex-DOM* [software]. [cit. 2017-12-20]. Available from: https://github.com/reflex-frp/reflex-dom

[19] TRINKLE, Ryan. *Reflex Platform* [software]. [cit. 2017-12-20]. Available from: https://github.com/reflex-frp/reflex-platform

[20] GHCJS CONTRIBUTORS. *GHCJS Examples* [online]. [cit. 2017-12-20]. Available from: https://github.com/ghcjs/ghcjs-examples

[21] YAMADA, Pedro Tacla. *ghcjs-loader* [software]. [cit. 2017-12-20]. Available from: https://github.com/beijaflor-io/ghcjs-commonjs/tree/master/ghcjs-loader

[22] PURESCRIPT. *PureScript* [software]. [cit. 2017-12-09]. Available from: http://www.purescript.org

[23] PURESCRIPT CONTRIBUTORS. *Differences from Haskell* [online]. [cit. 2017-12-09]. Available from: https://github.com/purescript/documentation/blob/master/language/Differences-from-Haskell.md

[24] PURESCRIPT. *Pursuit* [software]. [cit. 2017-12-09]. Available from: https://pursuit.purescript.org

[25] FREEMAN, Phil. *purescript-thermite* [software]. [cit. 2017-12-09]. Available from: https://github.com/paf31/purescript-thermite

[26] PURESCRIPT CONTRIB. *purescript-react* [software]. [cit. 2017-12-09]. Available from: https://github.com/purescript-contrib/purescript-react

[27] FREEMAN, Phil. *Thermite Documentation* [online]. [cit. 2017-12-09]. Available from: https://github.com/paf31/purescript-thermite/blob/master/generated-docs/Thermite.md

[28] SLAMDATA, Inc. *purescript-halogen* [software]. [cit. 2017-12-09]. Available from: https://github.com/slamdata/purescript-halogen

[29] SLAMDATA, Inc. *The Halogen guide* [online]. [cit. 2017-12-09]. Available from: https://github.com/slamdata/purescript-halogen/tree/master/docs

[30] SLAMDATA, Inc. *Halogen Examples* [online]. [cit. 2017-12-09]. Available from: https://github.com/slamdata/purescript-halogen/tree/master/examples

[31] MINGOIA, Alex. *PUX* [software]. [cit. 2017-12-09]. Available from: https://github.com/alexmingoia/purescript-pux

[32] MINGOIA, Alex. *Build type-safe web applications with PureScript* [online]. [cit. 2017-12-09]. Available from: http://purescript-pux.org

[33] MINGOIA, Alex. *PUX Examples* [online]. [cit. 2017-12-09]. Available from: https://github.com/alexmingoia/purescript-pux/tree/master/examples/

[34] CZAPLICKI, Evan. *Elm* [software]. [cit. 2017-12-10]. Available from: http://elm-lang.org

[35] GOOGLE, Inc. *Angular* [software]. [cit. 2017-12-10]. Available from: https://angular.io

[36] CZAPLICKI, Evan. *The Elm Architecture* [online]. [cit. 2017-12-10]. Available from: https://guide.elm-lang.org/architecture/

[37] CZAPLICKI, Evan. *Blazing Fast HTML* [online]. 2016-08-30 [cit. 2017-12-10]. Available from: http://elm-lang.org/blog/blazing-fast-html-round-two

[38] ELM ORGANIZATION. *Elm Packages* [online]. [cit. 2017-12-10]. Available from: http://package.elm-lang.org

[39] CZAPLICKI, Evan. *An Introduction to Elm* [online]. [cit. 2017-12-10]. Available from: https://guide.elm-lang.org

[40] *Elm Companies* [online]. [cit 2017-12-10]. Available from: https://github.com/lpil/elm-companies

[41] HOOFT, Rob. *Ideas behind the data model* [online]. [cit. 2018-01-05]. Available from: https://github.com/DataStewardshipPortal/ds-km/tree/master/datamodel

[42] MONS, Barend. *Data Stewardship for Open Science: Implementing FAIR Principles*. ISBN 9780815348184.

[43] CZAPLICKI, Evan. *Elm Core Libraries – Basics* [online]. [cit. 2017-12-13]. Available from: http://package.elm-lang.org/packages/elm-lang/core/latest/Basics

[44] CZAPLICKI, Evan. *Elm Core Libraries – Maybe* [online]. [cit. 2017-12-13]. Available from: http://package.elm-lang.org/packages/elm-lang/core/latest/Maybe

[45] CZAPLICKI, Evan. *Elm Core Libraries – Result* [online]. [cit. 2017-12-13]. Available from: http://package.elm-lang.org/packages/elm-lang/core/latest/Result

[46] CZAPLICKI, Evan. *The Elm Architecture + Effects* [online]. [cit. 2017-12-15]. Available from: https://guide.elm-lang.org/architecture/effects/

[47] ELM COMMUNITY. *Elm loader* [software]. [cit. 2017-12-15]. Available from: https://github.com/elm-community/elm-webpack-loader

[48] WEBPACK CONTRIBUTORS. *webpack-dev-server* [software]. [cit. 2017-12-15]. Available from: https://github.com/webpack/webpack-dev-server

[49] ELM ORGANIZATION. *elm-make* [software]. [cit. 2017-12-15]. Available from: https://github.com/elm-lang/elm-make

[50] ELM ORGANIZATION. *elm-package* [software]. [cit. 2017-12-15]. Available from: https://github.com/elm-lang/elm-package

[51] ELM COMMUNITY. *elm-test* [software]. [cit. 2017-12-27]. Available from: https://github.com/elm-community/elm-test

[52] NIELSEN, Jakob. *10 Usability Heuristics for User Interface Design* [online]. 1995-01-01 [cit. 2017-12-28]. Available from: https://www.nngroup.com/articles/ten-usability-heuristics/

[53] DOCKER Inc. *What is Docker* [online]. [cit. 2017-12-29]. Available from: https://www.docker.com/what-docker

[54] THE KUBERNETES AUTHORS. *Kubernetes* [software]. [cit. 2017-12-29]. Available from: https://kubernetes.io

[55] GOOGLE, Inc. *Google Cloud Platform* [software]. [cit. 2017-12-29]. Available from: https://cloud.google.com

[56] NGINX, Inc. *nginx* [software]. [cit. 2017-12-29]. Available from: http://nginx.org

[57] INTERNET SECURITY RESEARCH GROUP (ISRG). *Let's Encrypt* [software]. [cit. 2017-12-29]. Available from: https://letsencrypt.org

[58] GIT. *git* [software]. [cit. 2017-12-29]. Available from: https://git-scm.com

[59] GITHUB, Inc. *Github* [software]. [cit. 2017-12-29]. Available from: https://github.com

[60] TRAVIS CI, GmbH. *Travis CI* [software]. [cit. 2017-12-29]. Available from: https://travis-ci.com