CZECH TECHNICAL UNIVERSITY

BACHELOR THESIS

Intelligent Tree Data Management Component

Author:
Jakub LEČBYCH

Supervisor:
Ing. Petr KŘEMEN, Ph.D.

A thesis submitted in fulfillment of the requirements for the degree of Bachlor of Software engineering and technology

in the

Knowledge Based Software Systems Group
Department of Cybernetics

May 24, 2018

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Lečbych** Jméno: **Jakub** Osobní číslo: **457806**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Inteligentní komponenta pro správu stromových dat

Název bakalářské práce anglicky:

Intelligent Tree Data Management Component

Pokyny pro vypracování:

1.Proveďte
a)rešerši existujících JavaScript komponent pro výběr ze stromových dat, např. RC-tree a
b)existujících REST rozhraní pro poskytování stromových dat, včetně propojených dat.
2.Formulujte funkční a nefunkční požadavky na inteligentní komponentu pro výběr stromových dat a jejich tvorbu, která umožní přístup k propojeným datům od různých poskytovatelů.
3.Navrhněte komponentu a svůj návrh formulujte v jazyce UML.
4.Implementujte komponentu ve vhodném JavaScript frameworku a vhodně otestujte na několika datových sadách se stromovou strukturou a zhodnoťte její výhody ve vztahu k technologiím z bodu 1a.
5.Proveďte uživatelské testy na vhodných scénařích a otestujte škálovatelnost vzhledem k velikosti stromových dat a jejich struktury.

Seznam doporučené literatury:

[1] https://github.com/react-component/tree
[2] Tom Heath and Christian Bizer (2011) Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool.
[3] http://www.uml.org/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Petr Křemen, Ph.D., Skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **09.02.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce: **30.09.2019**

Ing. Petr Křemen, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

Declaration of Authorship

I, Jakub LEČBYCH, declare that this thesis titled, "Intelligent Tree Data Management Component" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Abstract

Intelligent Tree Data Management Component

Current UI components for working with structured tree data have some limitations. These limitations are especially performance issues when working with large datasets or support of Linked Data. After comparing and analyzing existing solutions and after the research of relevant web APIs, the design of the component is proposed, which complements these missing functions. Furthermore, this thesis describes the structure of the proposed component, its function and documents its API. Then it shows component usability and scalability in test scenarios. In conclusion, there is a summary of the results achieved and a proposal for successive steps describing the issues that need to be addressed.

Inteligentní komponenta pro správu stromových dat

Současná řešení UI komponent pro práci se strukturovanými stromovými daty mají určitá omezení. Těmito omezeními jsou zejména výkonostní problémy při práci s velkými datovými objekty nebo podpora propojených dat. Po srovnání a analýze existujících řešení a následnému výzkumu relevantních webových API je navrhnuta komponenta, která doplňuje tyto chybějící funkce. Dále tato práce popisuje strukturu navrhnuté komponenty, její funkce a dokumentuje její API. Poté je ukázána její použitelnost a škálovatelnost na testovacích scénářích. Závěrem je shrnutí dosažených výsledků a návrh následných kroků, popisující problémy, které je zapotřebí vyřešit.

Key words

Component, Filter, Linked Data, Multi-select, Select, Tree Data, Tree, React, Redux

Klíčová slova

Komponenta, Třídění, Propojená data, Více násobný výběr, Výběr, Stromová data, Strom, React, Redux

Acknowledgements

I would like to thanks to my supervisor Ing. Petr KŘEMEN, Ph.D. for his patience, advice, and feedback during my work on this project. Also I thanks for providing materials like example data and useful web sources to this topic.

Contents

List of Figures

List of Tables

List of Abbreviations

HTML	**H**yper **T**ext **M**arkup **L**anguage
API	**A**pplication **P**rogram **I**interface
RDF	**R**esource **D**escription **F**ramework
IRI	**I**nternationalized **R**esource **I**dentifier
URI	**U**niform **R**esource **I**dentifier
JSON	**J**ava**S**cript **O**bject **N**otation

Chapter 1

Introduction

1.1 Overview

Data, pieces of information that are measured, collected, analyzed and used for different kinds of things. Such data are distinguished based on their structure - unstructured (e.g., a list of strings) or structured (e.g., a graph), on their type - specific or general, and so on. Structured data are often represented as a graph. Human intuition allows us to understand these graph connections with no problem, but computers do not have that intuition. So the data context must be defined to enable the computer to understand these connections between the data and the meaning of that data. This problem is one of the main reasons why people created a concept of the Web Ontology Language (Group, 2009). However, more about that will be down below.

Data are in many cases provided through the web APIs. The most straightforward definition of the web APIs is following. Web API is the interface through which an application can communicate with that server. Then fragmentation of the web APIs is so significant that many existing solutions support only one type and does not allow to get responses from multiple APIs simultaneously. More detailed information about web API is in Chapter 3.

With all that data provided by the web APIs, there is a question - "How to render that data as options efficiently?". Because rendering large lists, is operation can be time-consuming and can affect the performance of an application significantly. Performance is one of the most critical aspects of the modern applications. Based on the study (Nah, 2011) the delay of 2 seconds is the limit where a response to simple commands becomes unacceptable to users.

1.2 Current situation

UI developers often need to use a tree select input in their applications. The Problem is that default HTML select input is not usable in many cases. Also, most current solutions cannot handle complex data (e.g., graph data) or cannot render large data lists efficiently. Moreover, if the developers finally find a solution with functionality that they need, the solution does not fit by design into their application. So in many cases, they have to develop their component for that specific problem.

1.3 Thesis goal

The primary goal of this bachelor thesis was to create a UI component that will support all features of the previous component such as rendering data as a tree, support of linked data, and support of graph type data structure. Also, the component

should provide several new features. These features are:

- selecting or multi-selecting options
- simplicity of use and integration with other applications
- flexibility (e.g. customization, different types of datasets)
- re-usability
- good performance with large datasets
- creating new options

1.4 Outline

This thesis is divided into two main parts - research part and implementation part. In the research part, the primary focus will be on the technology and work related to this problem. The second part will describe the implementation of the component, logical structure, and behavior. So the second chapter will be dedicated to the used technology, and to the related problems. Especially the Linked Data and the concept of the semantic web. Then we will have a look at the APIs that provide these data, their construction and how they behave. In the next part, we will look at the technical aspects of the component such. Then the focus will be on the component itself. A chapter about implementation will describe individual parts or sub-components and at the end of that chapter algorithms for filtering and rendering will be described. In the end, there will be a comparison of other solutions and benchmark. The last chapter will summarise the results and describe future steps.

1.5 Term definition

In this theses will be often used a several terms. To avoid any misunderstanding this table shows description of each term.

TABLE 1.1: Definition of terms used in this thesis

Term	Description
select input	HTML element represented as <input type='select'/>
dropdown menu	HTML elements displayed under select input. This element contains individual options
option	Part of the dropdown men. Option represent one data object. Example of data object: {value: '123', label: 'option label'} .
option provider	Web API providing data.
search	Process of filtering and rendering results of filter subprocess in form of dropdown menu with options.

Chapter 2

Background

One of the requirements of this thesis was the ability to work with linked data. As well as the React framework, because of high usage of that framework in other projects that are developed by Knowledge Based Software Systems Group. So now in this chapter, introduce the technologies that are related to this thesis.

2.1 Resource Description Framework

Resource Description Framework (V., D., and B., 2014) is a general description framework for describing informations provided by web sources. This framework creates a basis for the semantic web. Representation of the RDF can be as a graph or dataset.

In graph representation, data are a set of RDF triplets. Each triplet consists of three components - subject, predicate, and object. Subject and object are nodes and predicate is an edge of the graph. Triplet in official terminology express some facts about the source. A claim consists of three pieces that together create a sentence: subject –> predicate –> object. Within this statement, the source is a subject identified by URI (or IRI), the property is a predicate (what we say about the source), and value is an object. Predicates that we used for describing a source come from so-called schemas – that are vocabularies or ontologies. Examples can be *Dublin Core* or *Friend of a Friend* metadata standards. RDF syntax has various type of formats that are called serialization formats. Among these formats are for example Turtle, N-Quads, N-Triplets, or JSON-LD.(Sporny et al., 2014)

An RDF dataset is a set of RDF graphs. This set can consist of:

- default graph - exactly one RDF graph that does not have a name and may be empty
- named graphs - a pair of RDF graph and IRI or blank node

2.2 Linked Data

To understand what is Linked Data, you do not need any experience with web programming, just a basic knowledge of HTTP, URI, and IRI. The concept of Linked Data (Bizare, Heath, and Berners-Lee, 2009; Heath and Bizer, 2011) is following. Firstly, lest start what data are. Data are sets of values consisting of pieces of information. Problem with these data is that they do not carry any context about the information they represent. Imagine this example, two different data objects both objects have a property called 'name' with some value. That property represents some name, but without any further knowledge, that name representation can have a different meaning in each object.

On the other hand, Linked Data help to solve the problem by describing and interconnecting data by shared vocabularies. By packaging a data in the way that they express what kind of data they represent, you will receive a Linked Data. With all of this, there are still two main problems. First is what format use for Linked Data. There are lots of formats, for example, JSON, RDFa, XML, CSV, HTML, etc. The second problem is how we can link the pieces of data together. Most common and easiest way to express data is on the key-value pair.

2.3 JSON-LD

JSON-LD is a format based on JSON format. JSON format is easy to use in a web application because it is readily convertible to JavaScript object, which represents data in a web application

JSON-LD is an RDF syntax for describing linked data using JSON format. (Sporny et al., 2014) JSON-LD is both JSON document and RDF document (Klyne, Carroll, and B., 2014), but it has some differences with RDF. First, JSON-LD properties can be URIs (or IRIs) or blank nodes whereas in RDF properties must be URIs (or IRIs). That means that JSON-LD can serialize RDF data-sets. The other direction is not possible. Second, JSON-LD object lists are part of data model whereas RDF objects are part of the vocabulary. The last one, RDF values are either literals or language-tagged strings whereas JSON-LD also supports JavaScript native types, which are numbers, booleans, and strings.

EXAMPLE 2: Sample JSON-LD document using full IRIs instead of terms

```
{
  "http://schema.org/name": "Manu Sporny",
  "http://schema.org/url": { "@id": "http://manu.sporny.org/" },  ← The '@id' keyword means 'This value is an identifier that is an IRI'
  "http://schema.org/image": { "@id": "http://manu.sporny.org/images/manu.png" }
}
```

FIGURE 2.1: Example of JSON-LD document

2.4 React, Redux

React (*React*) is a JavaScript UI framework developed and maintained by Facebook. Its one of the most used framework because it's easy to learn and because of its performance efficiency. If you are familiar with the MVC model("Model view controller"), React is only the view part. Redux (*Redux*) is completely independent on React. Redux is a state container for JavaScript. This means, that Redux is just a framework for managing the state of your web applications. It evolves from Flux (*Flux*) framework but does not take Flux complexity.

2.5 React virtualized select and React select

React-virtualized-select and *React-Select* as the name suggests, are both React components. Both components are almost same. As the Brian Vaughn says: "react-select-virtualized works just like react-select". The only difference between them is that the first component is able to render large list more efficiently. This was achieved by a special way of rendering options. In a nutshell, the drop-down menu is rendered

only with a minimum required amount of options that are in focus (visible in the drop-down menu). A detailed description of this process will be described later on.

Chapter 3

Related work

In this chapter, I will compare three existing solutions. I choose these three solutions because they are the solutions with the highest number of downloads (more than 5000 per month. In case of rc-tree select 270 thousand per month). Also, all three solutions have most of their functionality similar or same to mine. These similar features are:

- selecting/ multi-selecting options
- displaying data as a tree or as a list
- toggle/ expand function for each option with children options
- filtering/ highlighting results

So I will not compare these functionalities, only the benefits, and disadvantages of each solution.

3.1 React dropdown tree select

One of the alternatives to my work is React dropdown tree select (*react-dropdown-tree-select*).

Pros	Cons
Design can be overridden to match Bootstrap (*Bootstrap*) or Material design (*Material design*) frameworks.	All nodes are collapsed by default and cannot be expanded altogether.
Search debouncing - The tree debounces key presses to avoid costly search calculations. The default duration is 100ms.	Cannot collapse/ uncollapse tree nodes during a search.
	Component does not support highlighting of matched text during a search.
	Component does not support asynchronous data loading.
	Data must have label and value properties.

Continued on next page

Table 3.1 – *Continued from previous page*

Pros	Cons
	Data cannot be represented as an array of graph nodes. Parent node must have child property to display children nodes and so on. This main problem with this approach is that for manipulating with some node, you need whole tree branch. Also, it is hard to calculate the length (total number of options) in this data structure.
	Renders all relevant options at once, this can slow performance of the component.

3.2 RC tree select

First competitive solution is a React tree select component (*rc-tree*). However, as I found out this solution is just a React solution of Ant design component. Moreover, nowadays Ant design components work with React as well so I do not see a point to use this component.

Pros	Cons
	Does not support highlighting of matched text during search.
	Selected items can be removed only via checkboxes. Other solutions have small '×' button that can remove corresponding option. But in the case of this solution you need at least two steps to remove a selected option. (1. find that option 2. uncheck it)
	Renders all relevant options at once which can slow performance of the component.

3.3 Ant design tree select

Last one is a tree select from Ant design (*TreeSelect*).

Pros	Cons
Complete ecosystem with other Ant design components.	Render all relevant options at once this can slow performance of the component.
Maintained and updated on a regular basis by the Ant UED team.	Does not cashed previous search result. For every new character (searched string) they filter through original data set.
Solutions for other UI frameworks like Angular, Vue, Ember and ClojureScript	

3.4 Summary of the related works

Each of these solutions provides expected functionality from this type of component. Also, these component are popular to use, that supports a fact of the high download rate. On the other hand, none of these components is useful for rendering a large number of options. Moreover, none provide the functionality of Intelligent tree select component such as support of multiple data providers.

Chapter 4

Design

4.1 Requirements

Requirements are distinguished into the two categories, functional, and non-functional. Each category has requirements specified for the user, and for a developer. A user is a person that use or communicate with the component in the form of actions like searching or filtering. Requirements are defined based on the assignment as well as based on the functionalities of the current solutions.

4.1.1 Functional

Functional requirements related to the user are following:

1. Component must support multiple options providers
2. Dropdown with results will be displayed on focus
3. Each option should show its state (additional info such as - comment, providers, or its state)
4. User must be able to filter among these options
5. Selection and multi-selection must be provided
6. The component must provide a way to create new options
7. The component should accept and visualize tree-structured data.

Other functional requirements for developers

8. The component must be able to work with linked data sources
9. The component should support multiple input formats – JSON (because of the easy interpretation on the client side) and XML (because of the wide market-place)

4.1.2 Non-Functional

Non-functional requirements

1. Operations like search and render must be scalable without negative impact on the user experience
2. Processing data, filtering and rendering should be real-time (not take longer than acceptable)

Other non-functional requirements for developers

3. Component must be easily integrable with React applications
4. Component must be flexible – custom styling, own filter method and render method

4.2 Use-cases

Based on the requirements, I identified the following use-cases. There are two actors, a developer and a user of the component. The focus of the use cases for the user is mainly on the interaction between user and data displayed by the component in the form of options. On the other hand for the developer use-cases are focused on the needs to integrate the component into their applications.

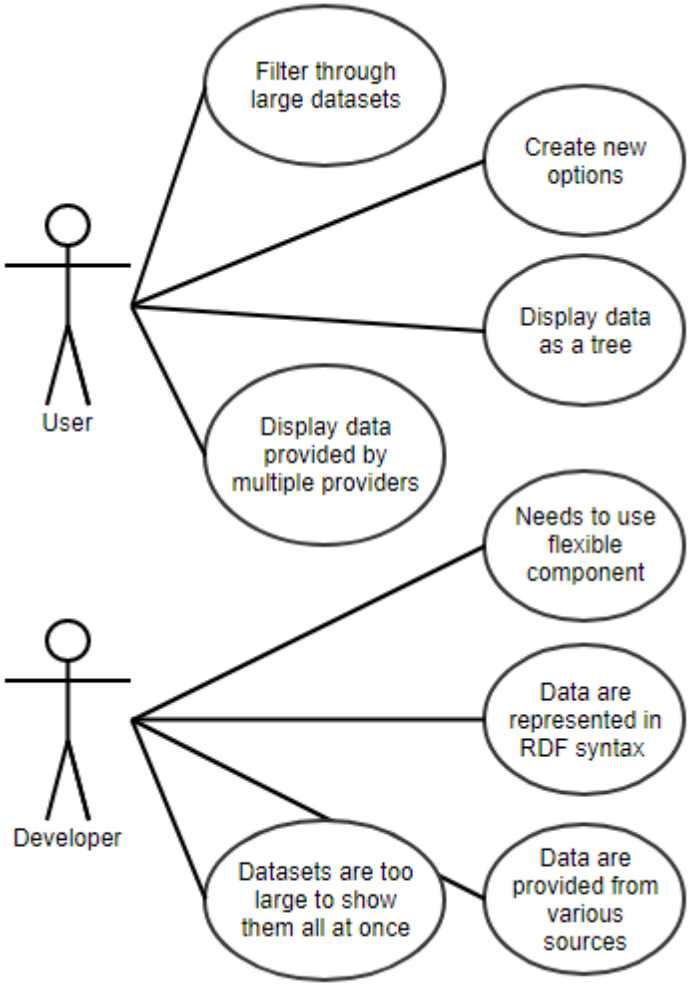

FIGURE 4.1: Use Case diagram

4.3 Providers APIs research

Firstly, lets quickly introduce what Application Programming Interface (API) is. API stands for Application Programming Interface. It is a program layer that is responsible for interacting with users, giving them responses based on their request. We will be focusing only on Web API. The simplest way to describe Web API is that Web API is set of dedicated web URLs, somewhere on the internet that return some response (usually in text format) to the requestor. There are several types of Web APIs. From historical SOAP (Simple Object Access Protocol) (D. et al., 2000) and SOA to more modern REST (Representational State Transfer) ("REST"). Because nowadays usage of the REST API is highest, let's focus only on this one type. I will be using Spotify API (*Spotify API*) as an example, because of their excellent documentation and variety of their endpoints. All Web APIs have three parts. As you can see below, first is their root address, in this case, it is https://api.spotify.com then following a version, but this is an optional part. Next is an endpoint /artists/id/albums; notice a id parameter in the URL, this is one way how to send some data to the API. Last part is query parameter. Query parameters are at the end of the URL behind question mark and contain key=value pairs connected with an ampersand. Usually, all query parameters are optional because they have a default value specified on the API side.

```
https://api.spotify.com/v1/artists/1vCWHaC5f2uS3yhpwWbIA6/albums?
market=ES&album_type=single&limit=2
```

Each HTTP (group, 2018) request must also have a header part, where are specified some other information. E.g. Accepted-language or encoding, but most important are Referer and Host.

FIGURE 4.2: Example of request header

Last part of each request is body part. This is the second way how you can send or get some data. But the data format must be in a format that is supported by the server side. The most commonly-used data format is JSON or XML. Often the service supports multiple formats, and the client can request one or the other by including 'json' or 'xml' in the header field 'Accept'.

The functionality of the Web API is much more complicated. The main problem is the complexity; each API can have different behavior based on what data format provides. As I mentioned earlier, most common data formats are JSON or XML. These are both text file formats, but you can also get images, web pages, music file and much more.

There are some examples of the Web APIs that returns different data types.

- returning json file: https://api.spotify.com/v1/artists/6sFIWsNpZYqfjUpaCgueju

- returning web page: https://open.spotify.com/artist/0OdUWJ0sBjDrqHygGUXeCF
- returning image: https://u.scdn.co/images/pl/default/438f9b65ac4eb48681351593142daeb070986293
- returning music file: https://p.scdn.co/mp3-preview/3eb16018c2a700240e9dfb8817b6f2d041f15eb1?cid=774b29d4f13844c495f206cafdad9c86

Some APIs supports filtering in the server side so you can create a requests to get a specific data you want. For example, this API returns 10 artists whose name contains 'tania'

- `https://api.spotify.com/v1/search?q=tania%20bowra&type=artist&limit=10`

Most of the time API returns only simplified data object, and to get the full information you must request it with for that specific ID.

- `https://api.spotify.com/v1/artists/1vCWHaC5f2uS3yhpwWbIA6/albums`

This request return an object with an array of simplified album objects. To get the full detail of the album object another request must be made.

- `https://api.spotify.com/v1/albums/43977e0YlJeMXG77uCCSMX`

This request returns a full album detail.

In conclusion, there is a lot of different Web API types. You can distinguish them based on their response format (JSON, XML, img, CSV, etc.), type of their response – some supports filtering results on their side, some return an only limited amount of data, while another may return only header information (e.g., IDs), so you must make another request for each ID to get detailed information. So in my component, I will provide an interface through which user can define how to handle communication with specific option provider (Web API)

4.4 Component architecture

Firstly let's have a look at the high-level view. Structure of the Intelligent Tree Data Management component consists of the three main parts or sub-components. The core sub-component is called Virtualized-tree-select and represent the input field with a drop-down menu. The second one is component representing modal form for creating new options. And the last one exposes some settings to the UI so the user can change the behavior of the component. Both Settings and Modal form sub-components, together with the main component communicate with Redux store, where all necessary data are stored. The third sub-component is independent on the Redux store because all necessary data are passed down as props from the parent component. More about each sub-component will be described in next chapter.

In a low-level point of view, let's look at the UML (*Unified modeling language*) class diagram. As you can see in the image below, main class have three important methods. First method 'onInputChange' check if current input is in history if not, then it call 'getData' for each provider, and then call 'addNewOptions'. The second method fetches data from providers and returns an array of results. The last method takes the results of the previous call, merge them based on priorities and save it to

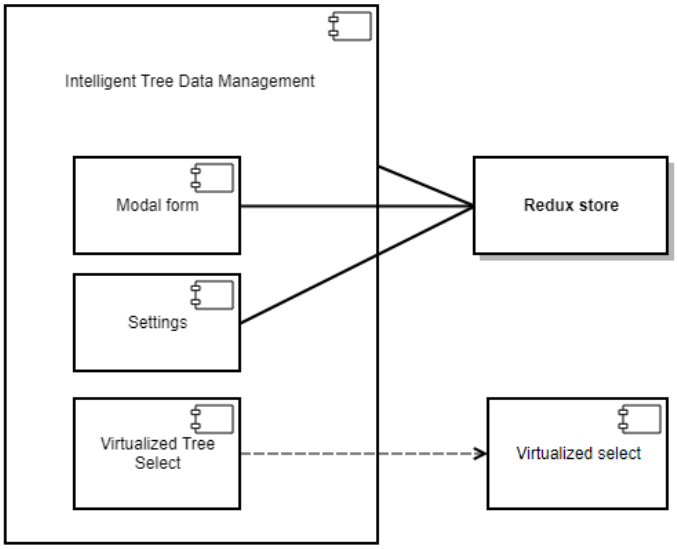

FIGURE 4.3: Component architecture

the already saved options. Other three classes are described in detail in chapter 5 section 5.2 Component parts.

The last one is not quite a class, but rather an object of a specific shape. Each option has one or more providers. Provider defines the option characteristics and provides some functions. First function 'response' should return result from the provider. Second one 'toJsonArr' format that response to the array of JSON objects if necessary, and the last one is called only if the label is not a string but array of objects. (e.g. label: [{'lang': 'en', 'value': 'This is label'}, {'lang':'de', 'value': 'Das ist Label' }]). Definition of all properties is in the appendix A.

The association between Provider and Option is 1..N: 1..M because each provider can provide the infinite number of options, but at least it must provide one option. From the other side, it is the same. Each option must have at least one provider. For options that are provided locally the provider called 'local provider' is assigned. Next, there is a reflexive association, where each option cant has more than one parent node and N children nodes. Because if you have an option with more than one parent node, then the data cannot be represented as a tree graph. Last association is between enum class called Type. This class represents the state of the option. Each option can have only one state. However, the state can be contained in N options.

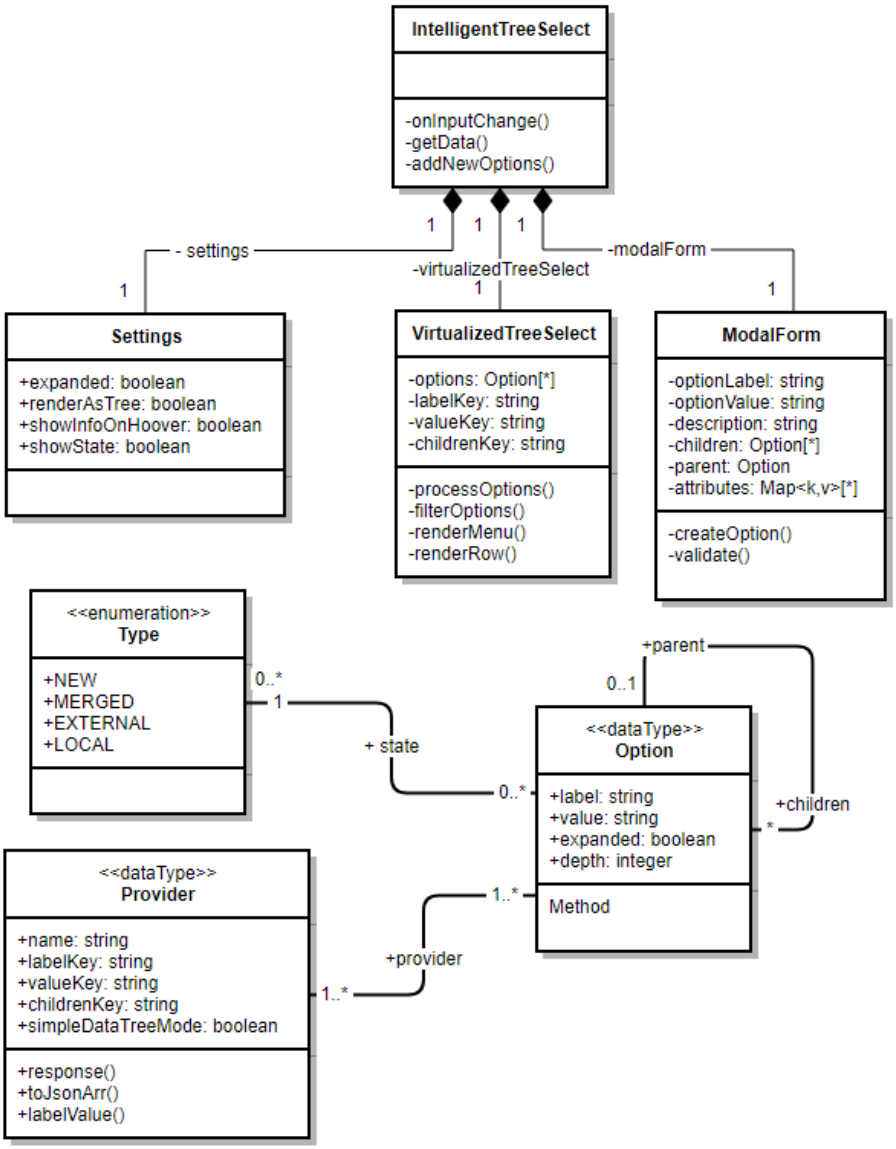

FIGURE 4.4: Component model structure visualized as a class diagram

4.5 Component life cycle and search cycle

I will not describe how React handle classes life cycle, because it is not a part of this theses. Also, I will not describe step-by-step of component initialization because it is an out-of-the scope of this thesis. Official React (*React*) documentation provides all the necessary information needed. So shortly, as you can see in this image below, the initialization process is divided into three parts. In the first part, the Redux store is created, and all variables (props) are saved. Then the actual HTML elements are rendered and in the last part. If local options are set then they are passed to the virtualized-tree-select sub-component, where they are processed, sorted and the tree graph is created.

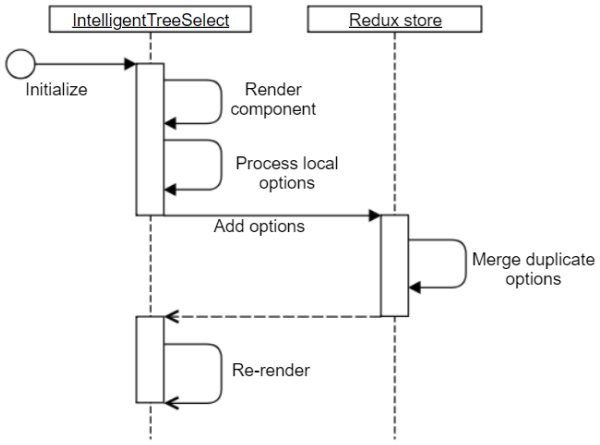

FIGURE 4.5: Component Initialization process

Search cycle is a little bit complicated because some actions are not handled by me, but by the *React-Select* and *React-virtualized-select* components. However, I can describe this process from my side. So, when user types into the input field the filter method is triggered by three arguments, current input, already selected options and all local (cashed) options. After that, the results are rendered. Then 'onInputChange' method is executed.The process of that method was already mentioned in the previoud section (4.4 Component architecture) in the second paragraph. This method fetches data from all providers, pre-process them (e.g., convert them to the same format if necessary). Then all new options are saved into the Redux store. Then the 'filterOptions' method is executed again but with new options and finally, new results are displayed. During all of this process, the loading indicator is displayed to the user, so the user is informed that some process is running in the background.

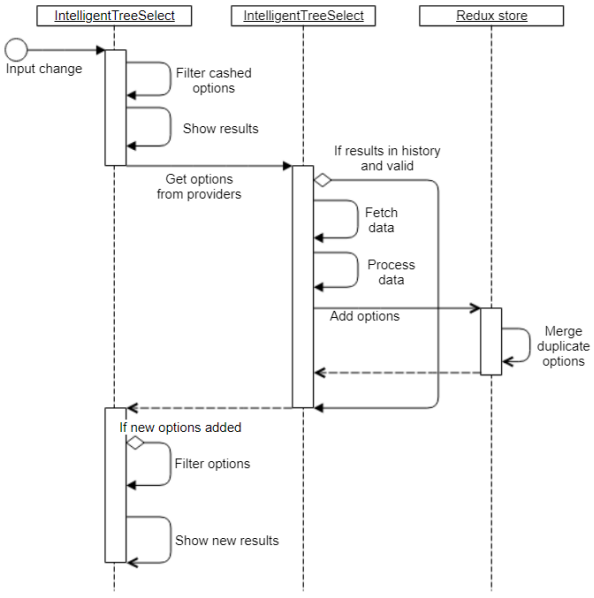

FIGURE 4.6: Search process

4.6 Creating and adding options

The process of creating and adding new option is simple. When a user clicks on button 'New option' the modal dialog with Redux form is displayed. Structure of this form is described in chapter 5 section 5.2.3 Modal form part. After filling up the form and submitting it, the validation function triggers. If the validation finishes with no errors (all required fields are not empty and contains at least 3 characters), the form is submitted. After that, the new option is created and added to all cashed options.

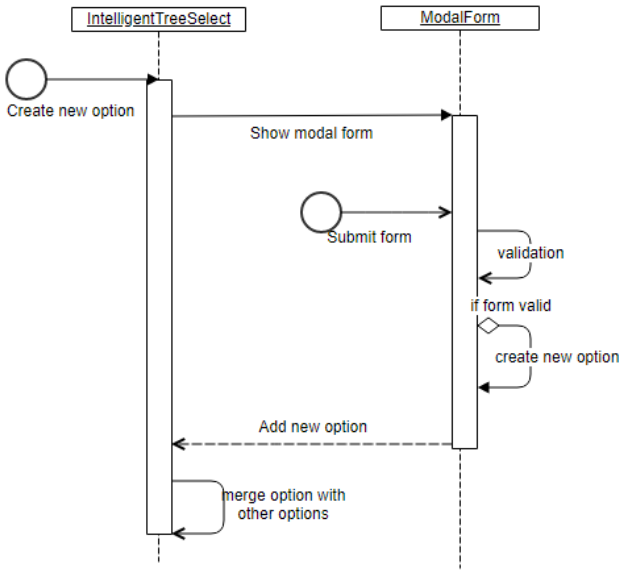

FIGURE 4.7: Process of creating new option

Chapter 5

Implementation

5.1 Inputs and outputs

As any other React component you can simply import the main class into your current project and that's pretty much all. Then you can customize the component, provide it some data and start using it. Detailed API is in the appendix, there you can find all properties that you can use.

The most important property is 'options', this property represents actual options that will be rendered. These options can be represented in two ways. As a graph (simplified version) where each node is represented as one object or as a list of objects (non-simplified) where children are parts of the parent. You can see an example of the options structure in the figure 5.3 down below.

Another important property is 'onInputChange', this is a function that will be called when a user selects an option. So this is the way how you can get all selected options. The option that you get back will not be same as you provide. The structure of the new option is following:

TABLE 5.1: Structure of the returned option

Key name	Type	Description
provider	Object	object representing provider of that option. See appendix for more detail structure of this object
state	Object	Object representing state of the option. E.g. {label: 'Merged', color: 'warning', message: '...'},
expanded	Boolean	Represent if option is expanded (children should be rendered or not)
'childrenKey'	List	List of the children IDs. Key name stay same as childrenKey value
...		All other properties that was in the original object

5.2 Component parts

5.2.1 Virtualized-tree-select part

Main part virtualized-tree-select component is custom component build on react-virtualized-select and react-select. This component retains the same API as both components, in addition it provide several new configurations, that configurations can be seen in appendix. So as React-Select, this component generates hidden text

input field that contains the value of the selected option, so it could be submitted as part of the standard form. When the option is selected, 'onChange' event is fired and this event return selected option. All the changes to the select input must be handled by the user; the user must pass that event value to the 'value' attribute of the select component.

5.2.2 Settings part

This is just a collapsible form with several checkboxes that provide some changes to the Virtualized-tree-select component, like expand or collapse all. Multi-select, this option, if it is checked then the component will provide multi-selection otherwise only one option will be selectable. Render as a tree, as the name suggests, this option renders all nodes as a tree also it slightly change filtering because by default if this option is checked the filtering will also show the whole path in the tree, meaning all parents until root parent will be displayed as well. Display info on hover, this option enables to show additional information for that node on hover. For example, description.

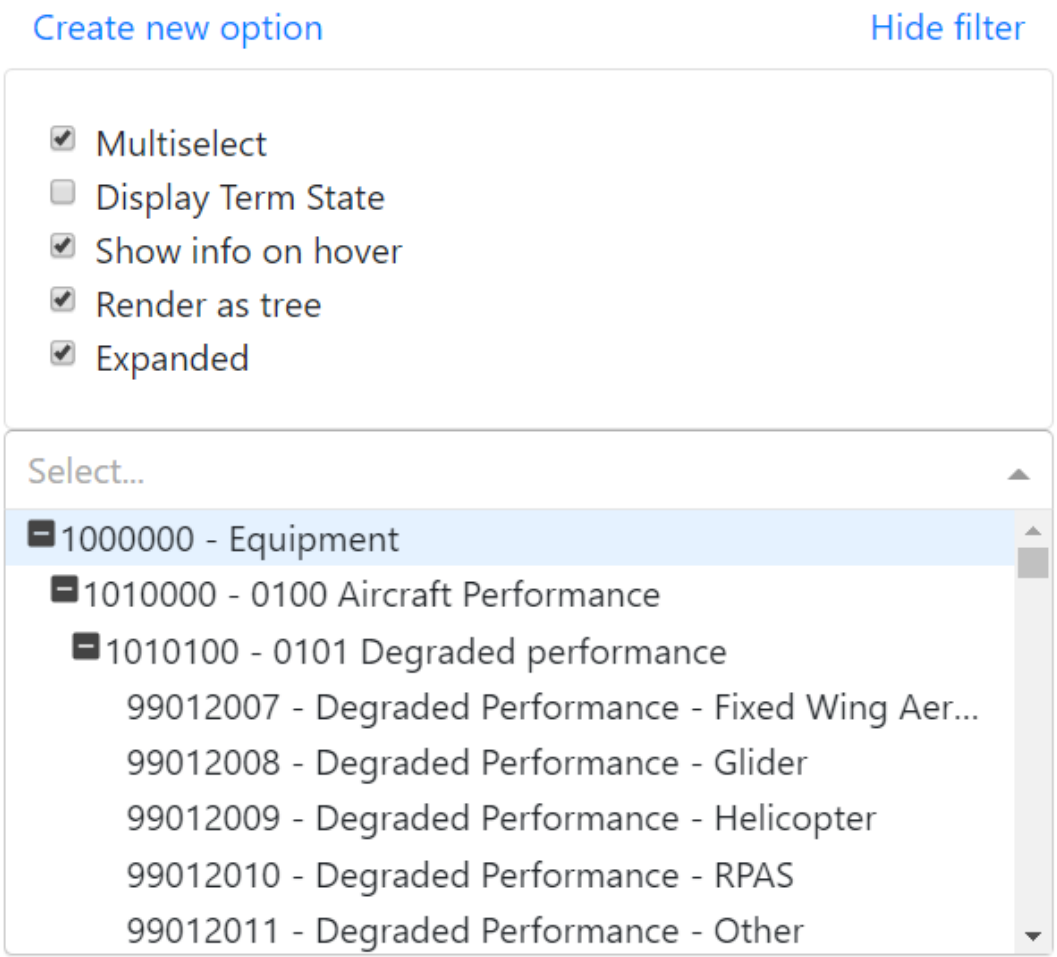

FIGURE 5.1: Virtualized tree select component with expanded settings

5.2.3 Modal form part

This component part consists of two dependent react classes. The first one renders empty modal dialog that contains an only header and close button. The second one renders the actual redux-form in modal body and actions buttons for submitting or canceling in the modal footer. As I mentioned earlier, this redux-form is used for creating new Nodes. It has several form fields:

- Option label (required) – representing a value that is visible in a drop-down box
- Option value (required) – representing the unique ID
- Option description
- Children – a multi-select box containing labels of all other options
- Parent – select box containing labels of all other options

In the advanced section, there is a button that adds a new pair of form fields. The first one represents object key, second one object value.

After each key press validation is triggered, so the user is informed about invalid inputs before submitting that form. Also, the form is submitted only in the case when all fields are valid. After that new node is created and its added to current tree graph and event 'onNewOptionCreation' will be fired.

5.3 Performance

Usability of the application may be affected by the performance of the used components. So the performance of my component was one of the critical points during the development. Even that the performance of the current version is acceptable, I still think that there is a space for improvements so that I will continue with the research in this field. As you may know for the flexibility, you have to pay something. Several algorithms have an impact on the overall performance. These algorithms are:

1. Filter algorithm
2. Algorithm responsible for rendering drop-down menu with options
3. Algorithm that merge all duplicate options
4. Sort algorithm
5. Algorithm for processing data

5.4 Render algorithm

These, who have good experience with HTML, may know that rendering process is a lengthy operation, compared to the JavaScript computation. So I decided to include *React-virtualized-select* that can perfectly handle the problem with rendering a large number of options. The first method is menu rendered and the second one is row renderer. Row renderer is responsible for rendering individual options. Menu renderer renders drop-down menu. This method takes several parameters, most important are - filtered options, currently focused option, option height, drop-down

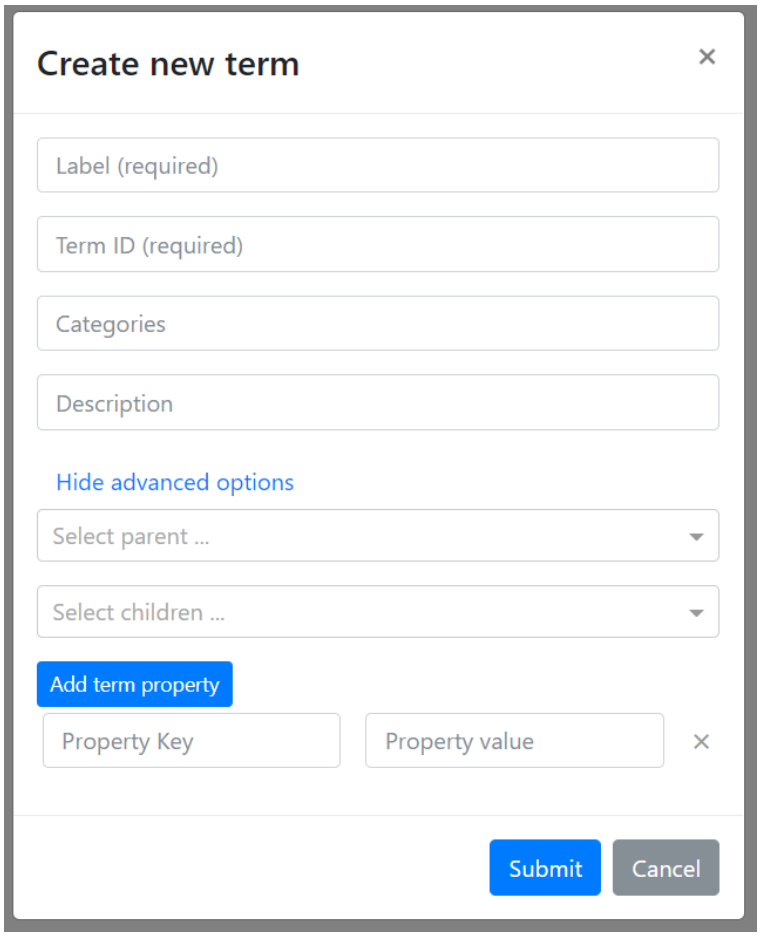

FIGURE 5.2: Redux form for adding new terms

menu height. From these parameters, it can compute how many and what options should be rendered and call row renderer for each of these options.

The best way is to provide an example. Let's say that filtered options is an array of 20 options. Currently focused option has an index equal to 10. Option height is 20px (pixels) and dropdown height is 100px. So the algorithm renders a container representing dropdown menu with 100px height, then the empty container with a height equal to 400px (20 options times 20px each option) into that previous container. Then algorithm renders individual options with the specified offset, so the method that renders each option is called for the ninth to sixteenth options.

5.4.1 Filter algorithm

The second algorithm that affects performance is an algorithm that filters all options. This algorithm can be divided into three parts. First, all matching options are filtered out. Second, for each filtered option find all parent options (this part will be conditional in future). Last part, all options that are not expanded (their parent have expanded set to false) are also filtered out.

In detail, the method for filtering options takes 3 arguments. All options, match string and selected options. First of all, options that contain a match string are filtered out. Then the parent options are found for each filtered option. Finding a parent option is done a constant time because the internal data structure of options

is same as the structure that is used by Firebase real-time database (), which is JSON tree structure. I choose this structure because data are represented as one big JSON object where keys correspond the option value (unique ID of that option) and values of the JSON object are options itself. This structure enables to access the options in constant time. And finally, the algorithm for every option of that previous subprocess (finding parents), check if an option has a property called 'expanded' equal to 'true', if not then the algorithm removes all children options of that option.

5.4.2 Merge algorithm

Next algorithm merges all duplicate options based on the provider's priority. This algorithm is executed only when new options are added. So if you decided to use only VirtualizedTreeSelect component where you include all options directly this algorithm is never executed. Anyway, this algorithm for each new option checks if that option is already present in the array of all cashed options. If no, then simply adds it and if yes then updates it.

5.4.3 Sort algorithm

Sort algorithm is included in the method that processes all input data. Like the previous algorithm, this algorithm is executed when new options are added. This process takes the option and adds some keys that are used, e.g., parent ID, expanded, state and depth. This algorithm has the same approach like a depth-first search algorithm ("Introduction to Algorithms"). So the first item in that sorted array is a root node, next item is left children node of that node and so on, and the last item in last right leaf node.

5.4.4 Performance results

Now let's look at some numbers and data. In this table below you can see how long does is to take for each process to finish. For the test purpose, I have one large dataset of 2617 options represented as a graph (simplified) 5.3a and one smaller dataset of 480 options represented as an array of objects with children as their properties (non-simplified) 5.3b.

These datasets represented types of aviation accidents and were provided by the Knowledge Based Software Systems Group(KBSS). Usage of these datasets is in the INBAS (https://www.inbas.cz/reporting-tool) project. Actual test data will be provided together with the source code.

This table represents the tests results. Each test was done 10 times on two different test machines and two different web browsers. Data in the table are average values.

TABLE 5.2: Time performance based on data structure and data size

Method	Data type	∼ Time to finish
Add new options	simplified	90 - 200 ms
Process options	simplified	4-7 ms
Filter options	simplified	1-7 ms
Add new options	non-simplified	3-5 ms
Process options	non-simplified	0-1 ms
Filter options	non-simplified	0-1 ms
Convert dataset	non-simplified	3-4 ms

```
[{
    "label": 1,
    "value": "some value",
    "children": "2",
},
{
    "label": 2,
    "value": "some value 2",
    "children": ["3", "4"]
},
{
    "label": 4,
    "value": "some value 4",
    "children": "",
},
{
    "label": 3,
    "value": "some value 3",
    "children": []
}]
```

(A) Example of simplified data structure

```
[{
    "label": 1,
    "value": "some value",
    "children": [
        {
            "label": 2,
            "value": "some value 2",
            "children": [
                {
                    "label": 4,
                    "value": "some value 4",
                    "children": [],
                }
            ]

        },
        {
            "label": 3,
            "value": "some value 3",
            "children": [],
        },
    ]
}]
```

(B) Example of non-simplified data structure

FIGURE 5.3: Example of data structure

Tests were done on two PCs and on the different browsers. However, the differences are negligible (0-2 ms).

The first test machine specification: Intel i7 series 2.8 Ghz, 16GB ram, win10. Second test machine specification: AMD FX series 3.5Ghz, 8GB ram, win10. Both tests were done on Chrome v66, Firefox v55, and Edge v40 web browsers.

As you can see, even with large datasets (2617 options) all together does not take more than 250 milliseconds. Maybe you think that non-simple datasets are processed faster. That's not really true, because even non-simplified datasets are converted into simplified datasets. That difference between them is because of these data was not so complex. The maximum depth of the child nodes was only 3 compared to the 8 in the other (larger) dataset. So the most significant impact has the complexity of the data, not the type you provide.

Chapter 6

Conclusion

6.1 Future work

After all, there is still much work that can be done. The current version of the component is in beta version because there are still some bugs that need to be solved. E.g., problems with destroying react child components in a drop-down menu or hovering effect. These bugs do not affect the functionality nor visual site, but they are not handled properly and result in errors in console.

Next thing I am planning to look at, as I already mentioned earlier, is a better function for filtering and processing data. Current methods are the first thing that comes to my mind and definitely can be optimized.

Also, I could add more interface features that will make the component even more flexible.

6.2 Summary

In conclusion, Intelligent tree select component is based on already great components such as *React-Select* and *React-virtualized-select*. This component is providing functionality for displaying options as a tree or creating a new one. As well it can be used with linked data, support two data formats, multiple providers, and much more. A structure is divided into three parts that are independent of themselves. Performance is not perfect in current beta version, but it can perfectly handle a large number of options in acceptable time. Overall compared to other solutions this component is same in core functionality but better in other aspects.

Appendix A

Intelligent Tree Select component API

All available select props are described here: https://github.com/JedWatson/react-select#select-props and here: https://github.com/bvaughn/react-virtualized-select/#react-virtualized-select-props. Additional parameters used by VirtualizedTreeSelect component are described in this table:

TABLE A.1: Intelligent tree data management component API

Property	Type	Default value	Description
childrenKey	PropTypes.string	'children'	Attribute of option that contains the values (ID) of the children options
valueKey	PropTypes.string	'value'	Attribute of option that contains the values of the option
labelKey	PropTypes.string	'label'	Attribute of option that contains option label
labelValue	PropTypes.func	null	Function that is called only if option[labelKey] is an object not an string. This function get option[labelKey] as an parameter and must return a string value. This is useful e.g. if your data are multilanguage
simpleTreeData	PropTypes.bool	true	Dataset is in simplified format
expanded	PropTypes.bool	true	Attribute if all options are expanded by default or not
renderAsTree	PropTypes.bool	true	Attribute if options should be rendered as a tree. If false options are rendered normally as for default select
displayInfoOnHover	PropTypes.bool	false	Display tool-tip with additional information
displayState	PropTypes.bool	false	Should display state of the option (local, external, new, merged)
optionRenderer	PropTypes.func	null	Custom way to render options (see below)
filterOptions	PropTypes.func	null	Custom way to filter options (see below)

Continued on next page

Table A.1 – *Continued from previous page*

Property	Type	Default value	Description
onOptionCreate	PropTypes.func	null	Callback on creating a new option
options	PropTypes.array	[]	Array of default options
providers	PropTypes.array	[]	Array of provider objects

Provider object structure

TABLE A.2: API of the provider object

Property	Type	Default value	Description
name	PropTypes.string	(required)	Unique identification of each provider
response	PropTypes.func	(required)	Function that return data. This function get one string parameter that is equal to current input
toJsonArr	PropTypes.func	null	Function that is called to convert providers response to JSON array. This function is called only when response is not an JSON object
childrenKey	PropTypes.string	'children'	Attribute of option that contains the values (ID) of the children options
valueKey	PropTypes.string	'value'	Attribute of option that contains the values of the option
labelKey	PropTypes.string	'label'	Attribute of option that contains option label
labelValue	PropTypes.func	null	Function that is called only if option[labelKey] is an object not an string. This function get option[labelKey] as an parameter and must return a string value. This is useful e.g. if your data are multi-language
simpleTreeData	PropTypes.bool	true	Dataset is in simplified format

Custom option renderer

TABLE A.3: Option renderer method API

Property	Type	Description
focusedOption	PropTypes.object	The option currently-focused in the drop-down. Use this property to determine if your rendered option should be highlighted or styled differently.
focusedOptionIndex	PropTypes.number	Index of the currently-focused option.
focusOption	PropTypes.func	Callback to update the focused option; for example, you may want to call this function on mouse-over.
labelKey	PropTypes.string	The attribute of option that contains the display text.
option	PropTypes.object	The option to be rendered.
options	PropTypes.arrayOf (PropTypes.object)	Array of options (objects) contained in the select menu.
selectValue	PropTypes.func	Callback to update the selected values; for example, you may want to call this function on click.
style	PropTypes.object	Styles that must be passed to the rendered option. These styles are specifying the position of each option (required for correct option displaying in the drop-down).
valueArray	PropTypes.arrayOf (PropTypes.object)	An array of the currently-selected options. Use this property to determine if your rendered option should be highlighted or styled differently.
valueKey	PropTypes.string	Attribute of option that contains the value.
onToggleClick	PropTypes.func	Callback to event for clicking on expand button
childrenKey	PropTypes.string	Attribute of option that contains the values of children options

Custom filter options

By default, a component uses a custom function for filtering the options. I don't recommend overriding this method unless you know what you are doing. For more details, you can look at https://github.com/JedWatson/react-select#advanced-filters

Bibliography

.

Bizare, C., T. Heath, and T. Berners-Lee (2009). "Linked Data - The Story So Far". In: URL: http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf.

Bootstrap. URL: https://getbootstrap.com.

codecademy. "Model view controller". URL: https://www.codecademy.com/articles/mvc.

— "REST". URL: https://www.codecademy.com/articles/what-is-rest.

Cormen, Thomas H. et al. "Introduction to Algorithms". In: ed. by edition. MIT Press and McGraw-Hill. Chap. 22.3: Depth-first search, pp. 540–549.

D., Box et al. (2000). "Simple Object Access Protocol (SOAP) 1.1". URL: https://www.w3.org/TR/soap/.

Dublin Core. URL: http://dublincore.org.

Financial, Ant. *TreeSelect*. URL: https://ant.design/components/tree-select/.

Flux. URL: https://facebook.github.io/flux/.

Friend of a Friend. URL: http://www.foaf-project.org.

group, MDN (2018). "HTTP". URL: https://developer.mozilla.org/en-US/docs/Web/HTTP.

Group, W3C OWL Working (2009). "OWL 2 Web Ontology Language". In: URL: https://www.w3.org/TR/owl2-overview/.

Heath, Tom and Christian Bizer (2011). *Linked Data: Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology*. 1st edition. Morgan and Claypool.

Jones, Dow. *react-dropdown-tree-select*. URL: https://github.com/dowjones/react-dropdown-tree-select.

Klyne, G., J. J. Carroll, and McBride B. (2014). "RDF 1.1 Concepts and Abstract Syntax". In: 1.8 RDF Documents and Syntaxes. URL: https://www.w3.org/TR/rdf11-concepts/#rdf-documents.

Material design. URL: https://material.io.

Nah, Fiona Fui-Hoon (2011). *A study on tolerable waiting time: how long are Web users willing to wait?* URL: http://sighci.org/uploads/published_papers/bit04/BIT_Nah.pdf.

rc-tree. URL: https://github.com/react-component/tree.

React. URL: https://reactjs.org.

Redux. URL: https://redux.js.org.

Sporny, M. et al. (2014). "JSON-LD 1.0." In: 16. URL: https://www.w3.org/TR/json-ld/.

Spotify API. URL: https://beta.developer.spotify.com/documentation/web-api/.

Unified modeling language. URL: http://www.uml.org/.

V., Guha R., Brickey D., and McBride B. (2014). "RDF Schema 1.1". In: URL: https://www.w3.org/TR/rdf-schema/.

Vaughn, Brian. *React-virtualized-select*. URL: https://github.com/bvaughn/react-virtualized-select/.
Watson, Jed. *React-Select*. URL: https://github.com/JedWatson/react-select.