BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

MODERN TREE VIEW COMPONENT FOR THE WEB

MODERNÍ TREEVIEW KOMPONENTA PRO WEB

BACHELOR’S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **LEVENTE BERKY**
AUTOR PRÁCE

SUPERVISOR **Ing. LENKA TUROŇOVÁ**
VEDOUCÍ PRÁCE

BRNO 2018

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems Academic year 2017/2018

Bachelor's Thesis Specification

For:	**Berky Levente**
Branch of study:	Information Technology
Title:	**Modern TreeView Component for the Web**
Category:	Algorithms and Data Structures

Instructions for project work:

1. Research the already existing web-based tree view implementation. Specify the requirements of a modern tree view web component.
2. Compare these implementations and analyze their performance. Study the fundamentals of modern JavaScript and Web Components.
3. Create implementations of the specification using modern JS frameworks. Design benchmarks to compare the performance of the implementations.
4. Evaluate the results and release the best implementation as a Node package manager package.

Basic references:

- Ivan Bozhanov. jstree. https://www.jstree.com/, 2017. [Online; accessed 2017-10-14].
- Jonathan Miles. Bootstrap tree view. https://github.com/jonmiles/bootstrap-treeview, 2008. [Online; accessed 2017-10-14].
- Open Source. Patternfly bootstrap tree view. https://github.com/patternfly/patternfly-bootstrap-treeview, 2017. [Online; accessed 2017-10-14].
- Martin Wendt. Dynatree. http://wwwendt.de/tech/dynatree/index.html, 2013.[Online; accessed 2017-10-14].
- Martin Wendt. Fancytree. https://github.com/mar10/fancytree, 2017. [Online; accessed 2017-10-14].

Requirements for the first semester:
Items 1, 2 and 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Turoňová Lenka, Ing.**, DITS FIT BUT
Beginning of work:	November 1, 2017
Date of delivery:	May 16, 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

Petr Hanáček
Associate Professor and Head of Department

Abstract

This thesis focuses on creating a web component to display hierarchical structure in a form of tree. The thesis explores the current state of web and web components, researches similar tree view components and describes the fundamentals of used technologies. The practical part describes the requirements for the component and describes the development and implementation of it. Furthermore, the thesis describes the process of testing and benchmarking and their results.

Abstrakt

Tato práce se zabývá tvorbou webové komponenty k zobrazení hierarchické struktury ve tvaru stromu. Práce zkoumá aktuální situace webu a webových komponent, vyšetřuje podobné "tree view" komponenty a popisuje základy v nich použitých technologií. Praktická část popisuje požadavky na navrhovanou komponentu a zabývá se procesem vývoje a samotné implementace. Dále je popsán proces testování a měření výkonu a jejich výsledky.

Keywords

web, web components, web applications, data structures, javascript frameworks, data visualization

Klíčová slova

web, web komponenty, web applikace, data struktury, javascript frameworky, data vizualizácia

Reference

BERKY, Levente. *Modern Tree View Component for the Web.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lenka Turoňová

Rozšířený abstrakt

Tato práce se zabývá tvorbou webové komponenty pro zobrazení hierarchické struktury ve tvaru stromu. Tento UX element se nazývá “tree view” a je již dlouhodobě využíván na desktopech. Nyní je potřeba ho vytvořit i pro web, jelikož webové aplikace se v poslední době rychle rozvíjely a dosáhly funkcionality desktopových aplikací. Tato práce se zabývá vývojem hlavně pro ManageIQ, který je vyvíjen společností RedHat.

Nejprve bylo zapotřebí stanovit technologie, které budou využity při vývoji. Existují dvě možnosti pro vytvoření komponent. Jednou z možností je knihovna a druhou komponenta pro nějaký framework. V JavaScriptu je manipulace s dokumentem dost obtížná a při tvorbě knihoven se většinou využívá nějaká jiná knihovna., proto jsou preferované frameworky. Sice poskytují jiné rozhraní ke kódování, ale mají spoustu výhod jak pro práci s daty, tak i pro vykreslování a optimalizaci při změně dokumentu.

V úvahu přichází tři největší frameworky: AngularJS, Vue.js a React. Po porovnání všech fremeworků byl zvolen jako nejvhodnější vhledem k požadavkům React. React má waterfall data flow, což sice nezlehčuje implementaci komponent, ale zajišťuje lepší strukturu kódu a je lépe udržovatelný. AngularJS je příliš komplikovaný a jeho komponenty se nedají používat v jiných aplikacích, které nejsou založeny na AngularJS. Nevýhoda Vue.js je obtížná spolupráce s TypeScriptem a to, že za ním stojí jenom malý tým.

Dále byl pro vývoj zvolen TypeScript, což je JavaScriptová nadstavba a kompiluje se v JavaScriptu. Výhodou je, že zavádí typy do JavaScriptu a upozorní na typovou nesprávnost při kompilaci. Zvyšuje kvalitu kódu a usnadňuje debugování.

Před vývojem byly ještě zkoumány ostatní implementace tree view. Tři nejlepší na webu jsou Dynatree, Patternfly Bootstrap Tree View a Fancytree. Zde jsou zajímavé tři aspekty: jaké funkce implementují, jak se pracují s daty a jak rychle se vykreslí stromovou strukturu.

Výkonnostní testy byly prováděny pomocí Google Chrome zabudovaného výkonnostního měření. Ukázalo se, že Dynatree je nejpomalejší při vykreslování velkého stromu a Fancytree je nejrychlejší. Funkcionalita Fancytree byla nejslabší. Patternyfly Bootrsap Tree View splnilo všechny požadavky na funkcionalitu a ve vykreslování dopadlo průměrně. S daty všechny knihovny pracovaly stejně. Data jsou v tree view modifikovány pomocí metod, které implementují.

Po zvolení vhodných technologií a prozkoumání existujících implementací můžeme přistoupit k samotné implementaci. Je potřeba se zaměřit na funkcionalitu i design. Obojí hraje důležitou roli při integraci do větších aplikací. Vyžadovanou funkcionalitou jsou základní prvky tree view, možnost expandovat a odebírat jednotlivé uzly, vybrat je a zobrazit checkbox. Dále jsou vyžadovány nadstandardní funkcionality jako hierarchické chování checkboxů, zakázání zrušení výběru a povolení vybírání položek. Další požadavky se vztahují na jednotlivé položky. V některých případech je potřeba skrýt jen několik checkboxů nebo zakázat interacke s položkou. Líné načítání (lazy loading) je jedním z nejdůležitějších požadavků a většina knihoven ho nemá implementované.

Požadavky na desing jsou jednodušší. Většina úprav vizuální stránky je provedena za pomocí tříd přiřazených k položkám. Při změně, výběru nebo zvolení položek je přiřazena CSS třída, a tímto způsobem může uživatel přizpůsobit svoji aplikaci. Dále se dají specifikovat ikony, obrázky namísto ikon, ikony pro výběr a pro checkboxy, výměna ikony s checkboxem a ke každé položce se dá přiřadit vlastní CSS třída.

Celá komponenta se skládá ze čtyř podkomponent. Na nejnižší úrovni jsou takzvané hloupé komponenty. První z nich slouží k vykreslení tlačítka pro expandování a odebrání položky a druhý má slouží k manipulaci s checkboxy. Nad nimi je další komponenta Node.

Tato komponenta slouží k vykreslování celku a rozhoduje co, kde a jak bude vykresleno. Má také částečně na starosti kontrolu vstupů od uživatele. Komponenta na nejvyšší úrovni se nazývá Tree. Tree má na starosti celé ovládání stromů, zpracování dat, změny dat, kontrolu vstupů od uživatele a poskytování rozhraní ke komponentám. Dále komponenta Tree zahrnuje několik pomocných statických metod pro lepší práci s daty stromu a snadnější použití komponenty.

Během vývoje se mnohokrát změnilo místo uložení dat. Nakonec jsou data uložena mimo strom v rodičovské komponentě. Tento způsob ukládání dat je doporučen v Reactě. Znamená to, že všechny změny, které jsou provedeny v komponentě, musí být delegovány na rodiče. Ten pak musí na jednotlivé delegované události reagovat správně, aby strom fungoval. Výhodou tohoto přístupu je, že se každá akce dostane k rodiči a ten pak může provádět další akce nebo implementovat svoji logiku. Komponenta se tímto způsobem stává mnohem více flexibilnějším.

Výkon byl jednou z nejdůležitějších částí projektu, proto musely být v některých případech provedeny kompromisy. Jedním z těchto kompromisů bylo přepsání cyklů do for cyklů, jelikož je tento způsob rychlejší než ostatní varianty. Rekurze byla optimalizována pomocí tail-call.

Ve vykreslování byl kladen důraz na to, aby se bylo nutné vykreslit, co nejméně elementů a tímto způsobem urychlit proces. Pro různé konflikty byly použity automatizované benchmarky, které ukázaly, jaký způsob je nejlepší pro danou problematiku. První benchmark zkoumal, jak zanořit elementy ve stromu. Existovala možnost využít zanořené listy, přidané <span> elementy před jednotlivé uzly a CSS třídy. Benchmark ukázal, že nejvýhodnější cestou jsou CSS třídy. Druhý benchmark zkoumal čas inicializace dat. Nakonec byla data částečně inicializována na začátku a tam, kde neexistovala, byly použity výchozí hodnoty. Tento způsob pozitivně ovlivnil využití paměti.

Unit testy byly prováděny pomocí knihovny Jest. Jest poskytoval jednoduché rozhraní na testování a byl integrován do vývojového prostředí. Testy snapshot testing byly drženy na minimu, protože veškeré změny ve vykreslování stromu by způsobily aktualizaci všech uzlů. Pomocí vlastních metod se kontrolovaly jen prvky z dokumentu důležité pro test-case. Na konci testování dosáhlo pokrytí testů 100% i na různé větve a řádky.

Jakmile byla komponenta otestována a obsahovala všechny požadavky, byl vytvořen NPM package pomocí utilit, které kompilovaly TypeScript a minimalizovaly kód. Tento minimalizovaný kód byl publikován a dá se snadno použít bez jakéhokoliv nastavení. Vývoj bude pokračovat i nadále a v budoucnu budou přidány nové optimalizace a funkce, které nebyly součástí této práce.

Modern Tree View Component for the Web

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the academic supervision of Ing. Lenka Turoňová and the technical supervision of Ing. David Halász. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Levente Berky
2018-05-16

Acknowledgements

I wish to express my thanks to my supervisors Ing. Lenka Turuňova and Ing. Dávid Halász who helped to create a more professional work. I also want to express my gratitude to Ing. Karel Hala, the ManageIQ team and Red Hat for their professional technical help and suggestions while programming the component.

Contents

Chapter 1

Introduction

The World Wide Web has remarkably quickly become a huge repository of information that use users all around the world. As browsers have become more sophisticated, users have become dependent on them. In other words, users can hardly work without them. Cloud services are all around us and the synchronization is a basic aspect of an application. Web is used for creating documents, printing, playing games, shopping and of course administrating all these services. Moreover, new technologies are still being developed, such as cloud services where synchronization plays a key role.

The layout of the web page carries a lot of information. The amount and the variability of the information compete with the desktop applications. However, not all the UX patterns have been implemented in the web applications. An example of such pattern is the tree view. It can represent hierarchical data and it helps the users to navigate, select, move, add or delete objects. The most common usage of the tree view for an average user is to select files to upload. Tree navigation provides a helpful interface to display the data stored in a tree structure. However, it can become problematic in case of large trees containing a huge amount of nodes which can cause slow loading times.

The work will focus on the performance and other library dependences as they make available a wider range of use. Since the work is a collaboration with Red Hat, the main goal is to implement those features that are required by their ManageIQ system.

ManageIQ is a management and automation platform that gives the administrators the possibility to simplify the control of their virtual, private and hybrid cloud infrastructures. Many cloud based projects, like Openstack, Amazon EC2, Google Compute Engine or Microsoft Azure are supported by ManageIQ and that allows the administrators to manage the diverse environments by one single tool [1],

The work is divided into five parts. Before the first part there is a Chapter 2 which introduces the reader to the web technologies. Then in the Chapter 5 the tree view is described in more detail. Moreover the other existing implementations of tree view components are analyzed and compared. The Chapter 3 contains information about technologies which will be used and why. The selected technologies are described in Chapter 4. The two final parts describe the process of development, testing and the final product. In the Chapter 6 all implementation details are described from the used tools to the publishing. The Chapter 7 describes the testing and the benchmark.

Chapter 2

Theory Basis

In year 1991, the Hypertext Transfer Protocol (HTTP), foundation of the World Wide Web (WWW) was introduced. Not a long after the first version of the Hypertext Markup Protocol (HTML) was introduced. HTML elements are the building blocks of a structured document. The elements can be links, images, quotes and other items, and they can be nested. This protocol allowed the creation of the first static pages. With the introduction of the Cascading Style Sheets (CSS) the ability to style the web pages had been extended. The need of the 'cascading' style sheets came from the fact that they can be loaded from multiple sources. Back then, the user had limited interaction with the web page. It served mostly for some representing purposes.

Later in year 1995, Netscape, the dominating browser at that time, defined and introduced a client side programming language. Nowadays, it is known as JavaScript. JavaScript is a prototype based, object oriented, interpreted scripting language. These three technologies together: HTML, CSS and JavaScript are the core technologies of World Wide Web content engineering.

As mentioned above, in the early days, the web pages were delivered as static documents without JavaScript. Any interaction with this document had to make a round trip to the server. Later Flash, the vector animation, was introduced. However, we can talk about web application after the introduction of JavaScript 2.2. JavaScript handled the client side programming and thus allows the web application to handle user interactions without refreshing the document from the server.

In year 1999, the Asynchronous JavaScript [9] was introduced which gave the ability to send and retrieve data from server asynchronously. Thanks to this feature web applications were able to change the content dynamically. At this time, HTML 4.0 and CSS 2.0 had already been introduced and web applications started to receive more spotlight. Powerful JavaScript libraries, e.g. jQuery, were developed to simplify the development of the client-side's JavaScript. In the near past, JavaScript received another helpful feature. It started to support Promises natively. Promises help to execute asynchronous code more easily. It had been available in a form of libraries before but they had their limitations. The new Promise has a callback function and the promise can be rejected or fulfilled in the body of the function.

To access and manipulate the web resources the REST (Representational State Transfer) was introduced. It is the last piece of the technologies which is needed to create a complete modern applications. Those web services which conform to the REST architectural style are called RESTful applications. They provide interoperability between the systems on the internet.

In the present time, there are lots of technologies which help in the web development. The current major HTML version is 5.0 and the newest CSS is the CSS3. JavaScript is able to draw pictures pixel by pixel (Canvas) and lots of frameworks are dependent on it. The most known frameworks are Angular.js [2], Vue.js [14] and React [6]. The frameworks depend not only on JavaScript but also on AJAX. A framework defines the entire application design. Some of them follow the model-controller-view paradigm to improve the code quality. The frameworks often define some kind of methods and functions which are not included in JavaScript. To keep the compatibility, they simply compile to JavaScript.

As the World Wide Web has grown larger and larger, the need of the reusable parts has grown too. To satisfy this need a new concept was introduced: the so-called components. The most portable components are the Web Components. New tags can be used in HTML by including the Web Components. These tags are processed by the component which most likely creates HTML elements as result. This concept is widely used in the frameworks too, but they implement their own way to deal with the components. The biggest library of the reusable JavaScript parts is the npm library. There are hundred thousands of packages including frameworks and their components.

To further improve code quality and maintainability the JavaScript language has to be developed continually. The current version is the ES6 which stands for ECMAScript6 (this is the official name of JavaScript). In addition, Microsoft created a superset language to the JavaScript and named it Typescript. Code written in Typescript [8] is compiled to JavaScript [9] and has compile time type checking. This improvement helps the developers to avoid common mistakes. Furthermore it provides more strict and more readable code.

To understand this project it is required to understand the current state of the web applications and the technologies they use. This chapter's goal was to provide enough information on the subject and make the next chapters easier to understand.

2.1 Glossary

In this section some of the words and abbreviations will be explained. These can help to understand the upcoming text.

- HTML DOM - (further as DOM) is a standard object model and programming interface for HTML. It defines:
  - The HTML elements as objects
  - The properties of all HTML elements
  - The methods to access all HTML elements
  - The events for all HTML elements
- JavaScript - is a client side scripting language which is able to manipulate the DOM.
- CSS - describes how HTML elements are to be displayed on screen, paper, or in other media. They consist of rules. These rules can be stored in separate CSS files.
- JSON file - stores data in JavaScript object notation.
- Lazy loading - is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed.

- Lazy rendering - is a design pattern to defer the rendering of an object until the point at which it is displayed.
- MIT License - is a permissive free software license originating at the Massachusetts Institute of Technology. As a permissive license, it puts only very limited restriction on reuse and has, therefore, an excellent license compatibility.

Chapter 3

Used Tools and Development Setup

This chapter contains the description of the problems and proposes the right tools to solve them. This means the current section studies of the advantages and disadvantages of the available technologies and select the most fitting ones for the work.

3.1 Development Setup

As is stated in the Chapter 5 summary, the libraries implementing tree view have no modern solution. They are all dependent on jQuery library which is considered as a negative attribute in this project. Another problem was the data binding. All of the libraries used methods for manipulating data. All data changes made on the originally passed data set would not change the tree view itself. Other operations have to be made to achieve the change.

Some libraries or frameworks could solve the above described problems. The frameworks are preferred because if a library would be used then a dependency has to be loaded to a project using a specific framework. Therefore frameworks which allow data binding and support components are in the scope. When looking for framework, not only the features are counted but the performance too.

When looking at maintainability a good code structure is needed. JavaScript [9] is a typeless language. Therefore the developer has to understand the code well. Most of the libraries are well documented but their code is poorly commented. The code should be well commented and structured in this project to be easily maintainable later on.

To solve the maintainability problems there are superset languages to the JavaScript. The superset language means that the language compiles to JavaScript with additional features. One of the most popular of these languages is TypeScript [8] from Microsoft. TypeScript provides type checking, interfaces and much more. On compile it can check the code structure and create warnings if it is not the expected one. For example, the class member order can be defined or warnings about logging and non-typed variables. The comments can be made as obligatory as needed. It is true that defining the comment section does not mean a well commented code, but at least inspires developers to do it. As a result anyone working with the code has to follow these rules and they stay consistent and readable for the other developers.

3.2 Available technologies

Nowadays there are lots of JavaScript frameworks out on the internet and their number is constantly growing. Three of the currently most popular frameworks are Angular (version 2 or above) [2], React [6] and Vue.js [14].

Angular 2 was introduced in 2016. It has been created and is maintained by Google. The current major version is the Angular 4. React is described as "a JavaScript library for building user interfaces". Initially it was released and developed in 2013. Nowadays it is maintained by Facebook. Vue was first released in 2014 by a former Google employee. Now it is maintained by a small team of 16 people. Currently it is the fastest growing JavaScript framework on the market. Let me have a look on these frameworks more closely.

All of the chosen frameworks are component-based. The components are easily reusable in multiple applications or in other components. However, all the tree frameworks are component-based React and Vue handled excellently dumb components (small stateless functions). The frameworks support data binding as well. Two way data binding is available in Angular while the one way data binding is implemented in React. Vue supports both of them. Angular's two way data binding changes the UI element on the side of the user input. React's one way binding goes only in one way. First the model is updated then it renders the element. Because of that Angular code is cleaner and easier to implement. The React approach results in a better data overview and makes debugging easier.

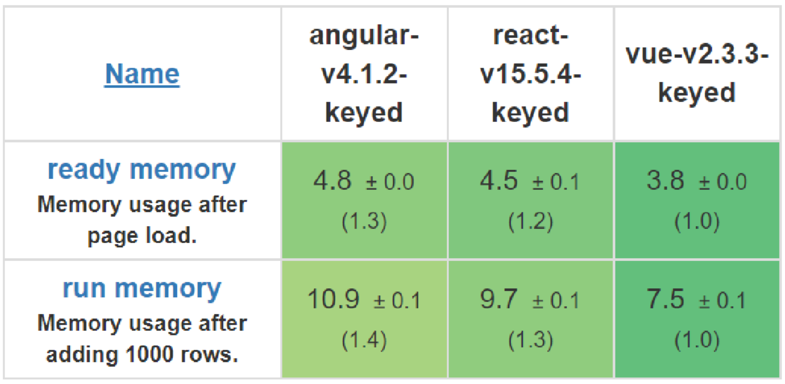
At this point all, the frameworks in question are suitable. Let's see how well they work with TypeScript, our favored JavaScript superset language. Angular relies on TypeScript and cannot be used without it. When working with React, TypeScript does not come with it by default. However they can work together well since TypeScript supports the React's JSX (what is JSX is explained in Section 4.1.1), so it can be type checked too. Unfortunately, Vue's TypeScript support is a work in progress.

It is worth to mention the complexity of the frameworks. Angular is more complex than the other two, but it offers more functions. This however means that the setup of an Angular project takes more time than the setup of the other two frameworks. React and Vue are rather libraries than frameworks. It is enough to include them in the project to use components from both of them and they work in any application.

When it comes to the performance then there are lots of benchmarks for JavaScript frameworks. One of the most needed benchmark is the rendering time. The result of a trustworthy [7] benchmark can be viewed in the Figure 3.2. The memory benchmark comes from the same source for the same frameworks. The results Figure 3.1 confirm that Vue has great performance and memory allocation, but all these frameworks are really close to each other when compared to particularly slow or fast frameworks.

3.3 Summary

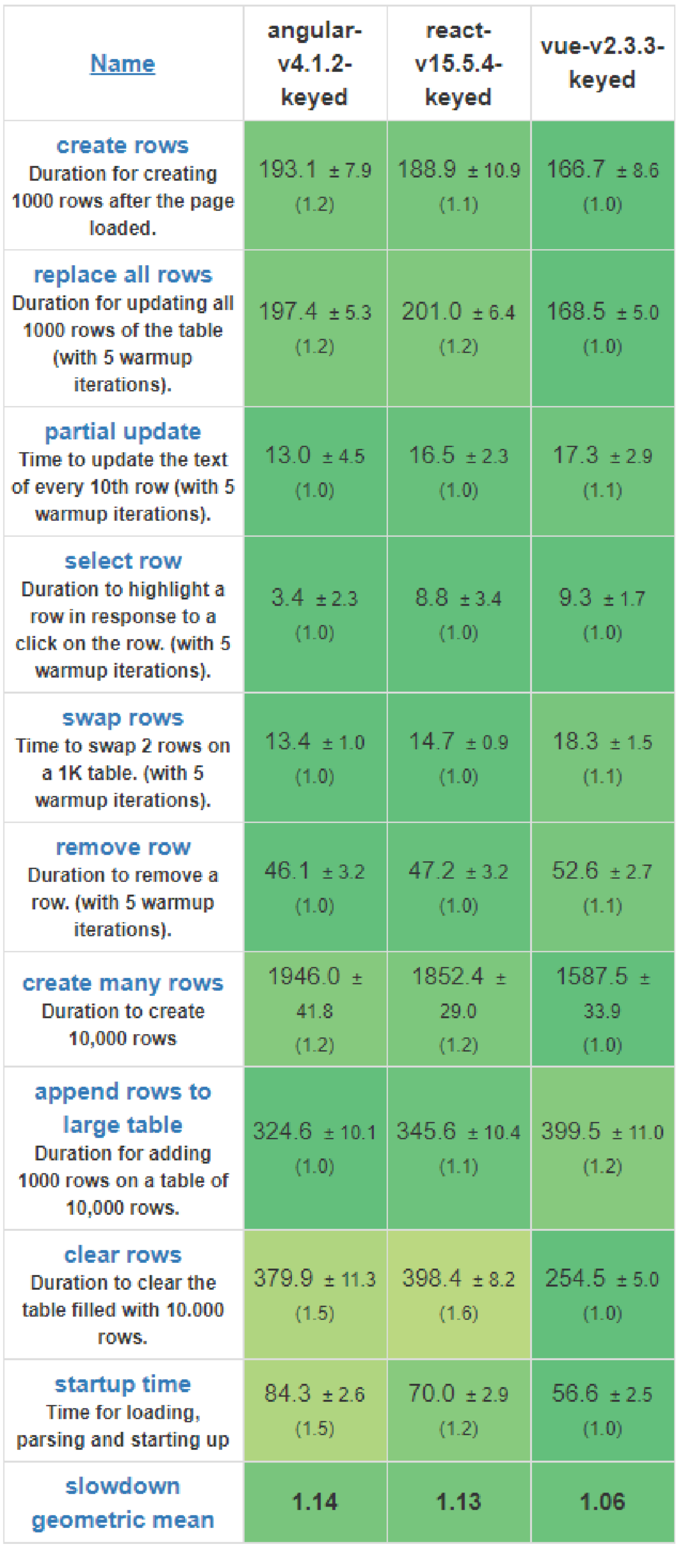
All the selected frameworks, Angular, React and Vue support components and data binding. However, Vue has the best performance. On the other hand it does not support TypeScript and therefore it is less suitable for the work. When developing a component it is good to know how long it can be used. The length of the usage can depend on the framework too. Angular and React is backed up by a larger company and it is less likely that they will drop its development. Vue has only a small team and that is the last reason why it will not be used. It means only the Angular and the React figure as possible choices.



Name	angular-v4.1.2-keyed	react-v15.5.4-keyed	vue-v2.3.3-keyed
ready memory Memory usage after page load.	4.8 ± 0.0 (1.3)	4.5 ± 0.1 (1.2)	3.8 ± 0.0 (1.0)
run memory Memory usage after adding 1000 rows.	10.9 ± 0.1 (1.4)	9.7 ± 0.1 (1.3)	7.5 ± 0.1 (1.0)

Figure 3.1: Angular, React and Vue memory benchmark results

Angular as the more complex system is more difficult to learn. React is simpler to set up but because of one way data flow, the approach to structure the code is different and has to be learned too. The last argument was the usability. Angular components are hard to use outside an Angular app however React components can be used on any site as long as the react is included. This raises React as the most fitting framework to write the component in.



Name	angular-v4.1.2-keyed	react-v15.5.4-keyed	vue-v2.3.3-keyed
create rows Duration for creating 1000 rows after the page loaded.	193.1 ± 7.9 (1.2)	188.9 ± 10.9 (1.1)	166.7 ± 8.6 (1.0)
replace all rows Duration for updating all 1000 rows of the table (with 5 warmup iterations).	197.4 ± 5.3 (1.2)	201.0 ± 6.4 (1.2)	168.5 ± 5.0 (1.0)
partial update Time to update the text of every 10th row (with 5 warmup iterations).	13.0 ± 4.5 (1.0)	16.5 ± 2.3 (1.0)	17.3 ± 2.9 (1.1)
select row Duration to highlight a row in response to a click on the row. (with 5 warmup iterations).	3.4 ± 2.3 (1.0)	8.8 ± 3.4 (1.0)	9.3 ± 1.7 (1.0)
swap rows Time to swap 2 rows on a 1K table. (with 5 warmup iterations).	13.4 ± 1.0 (1.0)	14.7 ± 0.9 (1.0)	18.3 ± 1.5 (1.1)
remove row Duration to remove a row. (with 5 warmup iterations).	46.1 ± 3.2 (1.0)	47.2 ± 3.2 (1.0)	52.6 ± 2.7 (1.1)
create many rows Duration to create 10,000 rows	1946.0 ± 41.8 (1.2)	1852.4 ± 29.0 (1.2)	1587.5 ± 33.9 (1.0)
append rows to large table Duration for adding 1000 rows on a table of 10,000 rows.	324.6 ± 10.1 (1.0)	345.6 ± 10.4 (1.1)	399.5 ± 11.0 (1.2)
clear rows Duration to clear the table filled with 10.000 rows.	379.9 ± 11.3 (1.5)	398.4 ± 8.2 (1.6)	254.5 ± 5.0 (1.0)
startup time Time for loading, parsing and starting up	84.3 ± 2.6 (1.5)	70.0 ± 2.9 (1.2)	56.6 ± 2.5 (1.0)
slowdown geometric mean	**1.14**	**1.13**	**1.06**

Figure 3.2: Angular, React and Vue benchmark results

Chapter 4

TypeScript and React

Now that all the required technologies are selected the next step is to study them in more detail. It is crucial to understand the technologies well as later on they will be used to create the component. The selected framework was React which works well with TypeScript.

4.1 Fundamentals of React

This section provides some basic understanding about React [6]. It describes its basics and some of those features which are needed to write the tree view component. React is a declarative, efficient, and flexible JavaScript library for building user interfaces.

4.1.1 JSX

JSX is a syntax extension for JavaScript and looks like HTML but accepts JavaScript code. JSX elements are compiled to JavaScript expressions and objects. It is cheaper than create DOM elements, therefore in React, all modifications are made on JSX elements and React takes care about the synchronization with DOM.

4.1.2 Rendering

It is necessary to specify at least one root element to render React content. The element rendering is done with a render function in which it is necessary to specify the JSX element (what to render) and the DOM element (where to render the defined element). When updating an element React updates in DOM only what is necessary. Conditional rendering is done with the help of JavaScript conditions, in if - then - else blocks. It can be decided which elements to render and which not. To render react elements outside of the React's DOM, portals can be used.

4.1.3 Components

There are two types of components: functional and class components. The functional components are often called as dumb components and they have only props which are data passed from its parent. They are only displaying the data. Class components can have states and various functions too alongside the props. They mostly contain the logic to control themselves and their child components.

4.1.4 State and Events

The state variable contains the components data. The difference between the props and the states is that the props are passed to the component and they should not change it. Unlike the state variable which represents the state of the component, and the component can change the own state, depending on user's input or other events. As data follow a waterfall model to synchronize the data between the components, the closest ancestor has to be found. This ancestor should store the data in question and it becomes the „only source of truth".

Event handlers help to modify the data stored in the children components in this one way binding model. Event handler is passed to the children component. When the child's data is changed the passed event handler is called which should modify the state which is passed to the child as prop in the parent component.

4.1.5 List and Keys

React have to identify all elements in its DOM, therefore when rendering a list each sibling needs its own key. Creators encourage using the element's identification when possible. Indexes should be used only when no other options are available. Keys have to be unique only in the current list, so the same keys can be used in the different lists. When rendering a list of children in a component, React fragments can be used to group them without adding an extra node to the HTML DOM.

4.2 Fundamentals of Typescript

This chapter describes the fundamentals of the TypeScript which is a superset of JavaScript language. It provides static typing, classes and interfaces. These properties allow writing more reusable, readable and editable JavaScript code. This chapter expects basic knowledge about the JavaScript language.

4.2.1 Basic Types and Variable Declaration

The Typescript [8] introduces type checking to the JavaScript. Basic types as string, Boolean and number are available. The compiler checks variable types when compiling to JavaScript. There is an *any* type which stands for any type as the reader might have guessed. This is the default for JavaScript without TypeScript.

The variable is declared in the same way as in JavaScript with *let* or *var* keywords. The difference between the two declarations is that the *let* declared variables are block scoped while the *var* is function scoped. The function scoped variables could be used, but sometimes they are the source of bugs, mainly when the programmer used other languages.

Another difference is that redeclaring the *var* is allowed while redeclaring *let* ends with error. TypeScript prevents redeclaring *let* with *var*. Shadowing variables or function parameters are done by redeclaring them inside another scope. This behavior helps when using the same variable name in inner function or in nested loops. However TypeScript gives a warning about the shadowed variable in an inner function.

The constant variables are declared with a keyword *const* and they cannot be changed after the first initialization. That does not mean that they are immutable!

TypeScript allows deconstructing arrays, objects, even in the function declaration itself. Swapping two variables in an array can be solved easily with deconstruction for example.

Spreading an array or an object is not the opposite meaning of the deconstruction. When spreading an array the same fields are rewritten with the last values. When spreading an object, however, there is a limitation: own methods cannot be spread.

4.2.2 Interfaces

TypeScript interfaces [8] work similar to the known interfaces of C++ or Java languages. The interface can describe required and optional parameters too. If it is needed the function parameters can be described by one or multiple interfaces.

Interface property with *readonly* option means that this property cannot be changed. It is the same as the constant variables except that they are used as properties. If the variable has more properties than it is specified in the interface and there is a need to pass it to this interface, then there are multiple options: it can be converted, said explicitly that it implements that interface or just modify the interface with *any* field at the end. All these solutions will allow passing a variable which has more properties, but implements the defined to an interface.

The functions can be declared with types like in Java or C++, too. The syntax is more like Pascal. If it is defined then the compiler checks if the passed variable's type is the same as it is defined in the function header.

Of course, instances can be extended similarly to classes. An interface can extend multiple interfaces. In that case, the extending interface gets all the properties from the other interfaces. When extending classes, the properties remain but the same methods stay without implementation.

4.2.3 Classes

Typescript [8] allows us to use object oriented approach to classes where classes are inheriting functionally and objects are built up from classes. The inheritance is done with *extends* keyword and calling the parent method is done with *super* like in Java.

Classes have private, protected and public instances. When nothing is explicitly defined instances are public by default. Of course the *readonly* option works here too. When using *readonly* the variables have to be set in the constructor.

TypeScript supports getters and setters. The assigning and retrieving values schematics are the same with these functions, only there is a function where some advanced rules can be defined. These functions are created with the keywords *get* and *set.* When extending an abstract class the child class must declare those abstract functions which do not have definitions.

4.2.4 Functions

In JavaScript same as in TypeScript[8] the functions can be named or declared anonymously. In TypeScript both must have defined parameter types and return type. If function does not have any return type then the returning type is *void.* Functions also can capture variables defined outside, of course, if it is not shadowed by another variable.

When defining a function parameter it can be set to optional or optional with a default value. Another helpful feature is that if the last parameter is *...rest* in the function declaration then the variable rest will contain an array of parameters passed after the last parameter. This parameter is handled in a same way as an array. Of course, the type of the rest can be specified in the declaration.

In JavaScript, *this* is a variable that is set when a function is called. Because it is set automatically the user always has to know about the context where the current function is executed. In TypeScript arrow functions allow us to bind the *this* variable to the place where it was declared not to the context where it is used. Further, the *this* type can be set in functions. This way the compiler can check if the variable has the correct type and warn about it. Setting the *this* variable to *void* in functions will make it unusable.

Overwriting the function is done with multiple declarations. The compiler selects the first matching functions so declarations should go from specific to more generic.

4.2.5 Generics

Generics [8] allow to write a function not only with one data type but with variety of data types. For this there is a 'type variable' which works on types rather than variables. Once set, the type variable can be used as another data type. The type in function argument can be set both explicitly and implicitly. Type variables can hold every kind of type which is available for normal variables.

The type variable works similarly as in functions in the generic interface. When creating an interface the type is passed. In classes type variable cannot be used to create static members, otherwise it works like an interface.

If there is a need to constraint a type variable, we can extend an interface or class. This ensures that properties defined in the interface will be available in the type and it will be checked on compilation. Further constraints can be placed between two type variables. As example the 'K' type variable has to be the 'key of T' type variable. This ensures that whatever the 'T' will be, the 'K' will contain a key from it. There is only one restriction that generics are not available for enumerations and namespaces.

4.2.6 Enumerations

Enumerations [8] can be numeric, string or heterogeneous but the last one is not recommended. The numeric value of an element is initialized with 0 when it comes first and it increments by one if it comes after an initialized value. They can be set to be computed members but after a computed member a constant value has to be set. The string enumerations have to be explicitly initialized.

TypeScript provides reverse mapping to each enumeration but if it is not desired then a constant enumeration can be used. Another property of constant enumeration is that it cannot have computed member. In ambient enumerations, a special case, the members are always considered as computed but we do not have to define them at initialization just later.

4.2.7 Type Interface and Compatibility

TypeScript [8] tries to calculate the best common type of an array if supplied variables with multiple types or the type can be specified explicitly. This is useful when compiler will not be able to decide it from supplied variable types. In that case, the compiler sets the type to a union type calculated from all represented types.

On the difference to other languages like C or Java when comparing interfaces, classes or arrays, to pass it is enough to have the same structure. So, assigning class to an interface with similar structures will not generate an error.

Functions are compatible if the source has at least those parameters as the target. The target must have a subset of return parameters of the source. The optional parameters can be left out and the rest parameter is threated as an infinite number of optional parameters.

Classes are compared only with instances not with static members, but the private and protected members have to match as well to pass the comparison. The generics are compared after the type substitution. Enumerations are compatible with numbers but not with different enumerations.

4.2.8 Symbols and Iterators

Symbol in TypeScript [8] is a primitive data type. It is created with constructor and have an optional key parameter. They are always unique (even with the same key). There are many predefined symbols.

The *Symbol.iterator* is used to determine if the object is iterable. Every object which implements an iterator symbol is considered iterable. Array, string, map and set have a predefined *Symbol.iterator*. There are two types of iteration. One is done with *for..of* keywords and returns rather values while the *for..in* returns a list of keys.

4.2.9 Modules

TypeScript [8] supports modules similarly to JavaScript too. Any file containing top-level import or export statement is considered as a module. Modules are executed in their own scope and only the exported variables are visible outside.

While exporting and importing objects they can be renamed. One default export per module is allowed. That allows importing from module without defining the concrete object.

When importing other JavaScript libraries without interface then the defining can be supplied in special files with '.d.ts' extension. This file should contain the definitions for the imported module. When using shorthand definition in case there is no '.d.ts' file then all the types will be *any*. Importing different types of files can be done with wildcards. For example, importing text file is done with „*!text“ or with „text!*“ wildcard.

4.2.10 Namespaces

Namespaces [8] are very basic objects and they allow better code organization. Declaring an object in a namespace rather than in the global namespace makes easier to name them. If using namespace object outside the namespace then they must be exported.

Namespaces can be split into multiple files. When extending or using the namespace from multiple files a reference must be put at the beginning of the current file. If an namespace is spanned across three files and to use something from each all three of them must be referenced. It is advised and more common practice to use and structure code with modules rather than with namespaces.

Chapter 5

Tree View

The web has been around for more than 20 years, giving us sample time to discover user experience (UX) problems. Some solutions for UX problems worked so well that designers began to use them repeatedly everywhere. And because of familiarity, users came to expect them. This was the beginning of web design patterns: a common visual language that both the designers and the end users understand. One of these UX patterns is the tree view. Tree views are used to present hierarchical view of information. The items in the tree view are called nodes. The most common visualization is an indented list. They were used before the internet as well. The most common usage of tree views is in file managers where they help to navigate in the file system. They are often used as an outline for a document. The most known implementation is in the Window's file manager which is shown in the picture below.

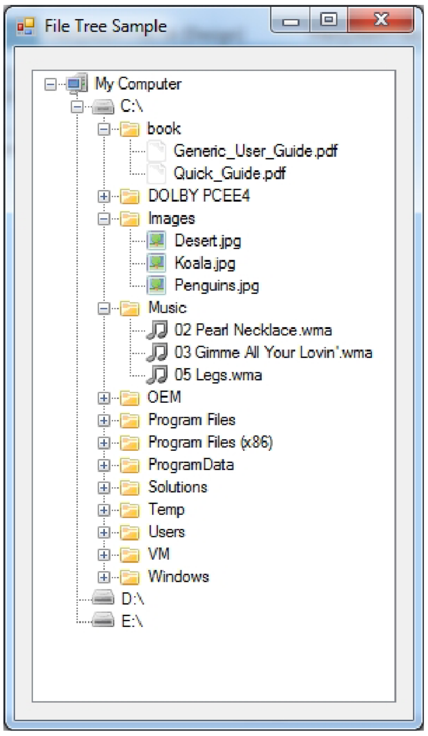


Figure 5.1: Example of a Tree View in Windows 8

The web applications started using them later on. Google Documents use it for the outline. For uploading files to web there is most likely a tree view to represent our file system and they can be used to navigate around in an application as well. To be able to implement the behavior of the tree views most of the code has to be on the client's side. This means web implementations depend on JavaScript. Before there were any frameworks the first implementations were libraries. When the libraries were used on a web page, they had to include them. The libraries mostly rewrote a part of the HTML document. To make this process simpler these implementations often use some of the available libraries. One of the often used libraries is the jQuery. It is well documented and has plenty of methods to manipulate with the document object model (DOM).

However, these tree view libraries can be used with frameworks too, it is simpler and resource-wiser to use tree view component in the framework. Some of the frameworks already has some tree views implemented, but most of them are not a viable replacements of the older libraries.

5.1 Existing Implementations

The aim of this section is to research and compare some of the already existing tree view libraries. The tree views will be inspected from two points of view: features and performance. When comparing the performance the main aspect will be the rendering time. Moreover, the dependencies will be compared. The optimal situation would be a library without a dependency.

The rendering time is measured with Google Chrome's built in performance test. The process consists of three steps. Firstly, a tree is generated containing only the title field. This is generated programmatically for each component. The JavaScript code is demonstrated in the Listing 5.1. The generated tree is inserted into the source file as a string, therefore the generating time is excluded from the tests. At last Google Chrome's Performance test is executed. The interval for tests is between the start of the reload and the page ready event. At this time the page generates only the tree.

Two types of the performance tests are executed. Only the top level nodes are shown in the first one. In the second one, the nodes are expanded. The total time of the performance test is divided into five different parts: loading time, scripting time, rendering time, painting time and other. The most important from these is the rendering time and the total time (for the present work).

```
tree = []
for (var i = 0; i < 500; i++)
{
    tree[i] = { title: "title" + i, children: [] }
    if ( i < 200 ) {
        for (var k = 0; k < 200; k++)
            tree[i].children[k] = { title: "subtitle" + i + "." + k}
    }
}
```

Listing 5.1: Tree Generating function

The features have to be compared in a more complex way. In order to do this, the requirements of the ManageIQ [10] system had to be considered here. The basic required features are the node expansion and the checkboxes. Connected to the checkbox, the option of the hierarchical check is needed. This means, if a node is checked, then all children nodes will be checked. Then the parents depending on their children's state can pick up one of the next states: checked, unchecked or partially checked. Another helpful and needed feature is the node selection. When a node is selected, there should be a way to the parent application to catch this event. This feature needs at least two options. The first one is the ability to prevent deselect. This option should guarantee that there is at least one selected node all the time. The second one is related to the first one. When both prevent reselect and allow reselect are enabled, when node is clicked when it is already selected the select event should be fired again. The multi-select is not a must have feature.

When manipulating the data, the overall requirement is to have as much control over it as possible. In addition, a common feature is required, the lazy loading. Lazy loading means that the node's children are loaded from the server when the node is expanded. It is done by the help of AJAX methods. When talking about the data it is worth to mention that some of the features should have the option to be disabled or enabled per node basis, namely the checkboxes and the ability to select the icon.

The last requirement is the styling. Most of the applications need the tree in their own design, so it is important how much and how the tree styling is done. When comparing the implementations it is hard to say which method is better in customization. For now the preferred method will be the class approach. The class approach manipulates with CSS classes. The optimal would be if for each item, operation and change there would be appended class. Then the user can style the elements by class name which is common in web development.

After the research on the web, I found three feature rich libraries. Two of them are from the same author: Dynatree and the Fancytree. The third one is currently used in the ManageIQ, the Patternfly Bootstrap Tree View. It is a customized version of the Bootstrap tree view. As for the frameworks there are some tree views for them but they do not meet the expectations. For example, the Angular Tree Control does not have checkboxes, the React Tree View lacks checkboxes too. Moreover it does not have the most of the required functionality.

5.1.1 Dynatree

Dynatree [12] is an older tree view plugin, but still a viable option for displaying hierarchical data. It is feature rich and has a really good documentation with examples. First we look at its features, what it can do and how it is done. It is important to state at the beginning that this library heavily depends on the jQuery library. If the user wants to use more of its features, he would definitely need to include not only the main jQuery library but the jQuery-UI and the jQuery-contextMenu.

As it was expected, this tree view can expand and collapse nodes. The node's expanded state can be set on initialization or per node basis. Both the checkboxes and the select boxes can be enabled. The checkbox has all the abilities that are listed above, meaning it has an option for hierarchical select and can be disabled for each node if it is needed. On the other hand the select box is not what is expected. If it is enabled, there is another circular box which can be checked or unchecked. and while supporting multi select, the behavior cannot be matched to the expectations in any way. To catch the selected event there is only a workaround. Mostly there are only operations with the checked and the selected nodes. The other two requirements, preventing deselect and allowing reselect, are not available in this library. The images Figure 5.2 show how the checkbox and the select work in the Dynatree.

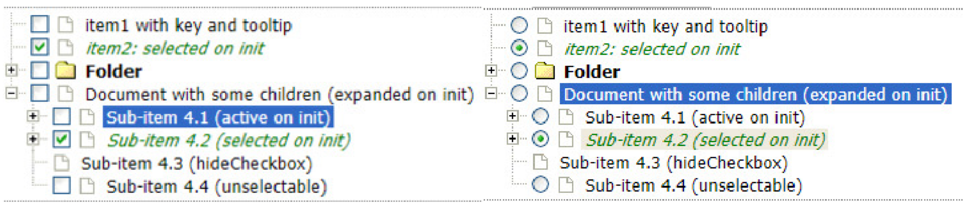

item1 with key and tooltip
item2: selected on init
Folder
Document with some children (expanded on init)
Sub-item 4.1 (active on init)
Sub-item 4.2 (selected on init)
Sub-item 4.3 (hideCheckbox)
Sub-item 4.4 (unselectable)
item1 with key and tooltip
item2: selected on init
Folder
Document with some children (expanded on init)
Sub-item 4.1 (active on init)
Sub-item 4.2 (selected on init)
Sub-item 4.3 (hideCheckbox)
Sub-item 4.4 (unselectable)

Figure 5.2: Example of Dynatree checkboxes (left) and selects (right)

When talking about styling there are not many options. The icons can be set for nodes with and without children and also the expanded node icon can be set. To each node a class can be added. It is not impossible to theme the whole tree but it would raise difficulties. Fortunately Dynatree comes in two themes, the standard and the Vista theme.

The analysis of the tree data structure and the methods to change it revealed that there is only one way to change the data in the tree after initialization. It can be done only with the help of the tree methods. There are methods for adding, removing and modifying nodes. However, the tree will not update itself if the data is changed outside of it. This library implements the lazy loading functionality.

There are much more features which are not relevant for this project. These features include HTML node text, context menu, in form options, and all sort of methods to expand, collapse or change checkbox for all nodes.

When it came to test of the performance, Dynatree performed variously on the two tests. The total time on the first test was acceptable with its 1062.1ms. However, the rendering time was 79.3ms and there was a 16ms painting time too. The chart and the more detailed result can be viewed in the chart Figure 5.3. The higher rendering time means that the performance may drop on bigger trees.

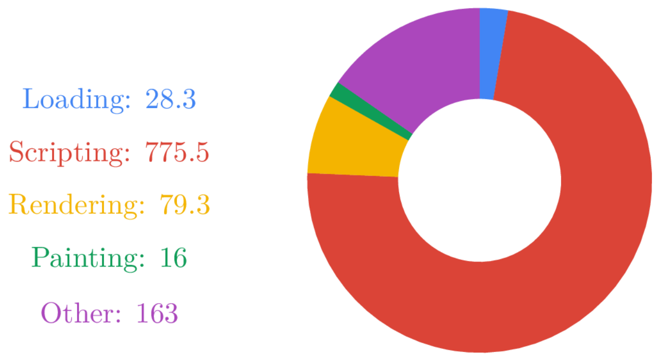


Figure 5.3: Dynatree first performance chart

The performance dropped a lot during the second test. The total time increased to 14160.7ms and out of that that time 4774ms was the rendering time. Compared to the first test it increased too much. This is because the number of the displayed nodes was much higher. The graph with the details are shown in Figure 5.4.

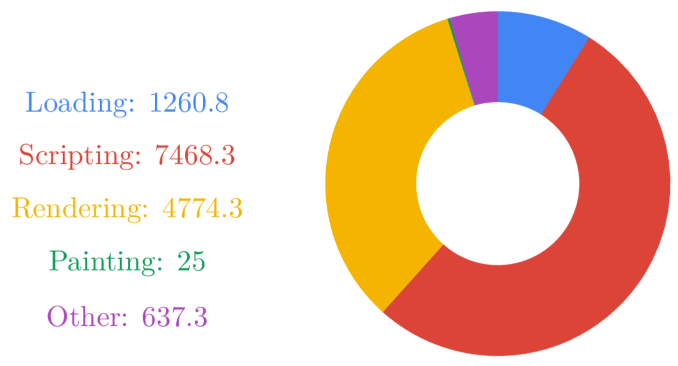


Figure 5.4: Dynatree second performance chart

5.1.2 Patternfly Bootstrap Tree View

Patternfly Bootstrap Tree View [11] is the currently used tree view library in ManageIQ. It was developed from the Bootstrap Tree View and it is customized to fit the needs. Therefore it is expected that it will meet the requirements the most splendidly. At first look, it is obvious that it depends on the bootstrap library for the styling and on jQuery for manipulating the DOM. Let me start with the features.

The library provides the exact same functionality for checkboxes and selection as it is required. Hierarchical select can be enabled and disabled. For each node the checkbox can be set to hidden additionally. The select works perfectly too. Two switches, the preventUnselect and the allowReselect, are implemented to add this functionality into the tree. For catching all the check and select events there are methods. These methods are helpful when action should be fired on interactions with the tree.

The styling is more extensive as it was for the Dynatree. All of the icons can be changed including checked, unchecked, selected or changed icons. The default node icon can icon or image. There are lots of options to set colors such as text color, background color or highlight color. These options are available on node basis too. Additionally, each node accepts additional classes. The default view with checkboxes and with a selected node is shown in the image Figure 5.5.

The data passed to this tree can be accessed again with methods. There is a wrapper around the tree. To change the data this wrapper has to be used. There are plenty of methods to update, add or remove data in the tree. It also supplies methods to retrieve the nodes or part of them, but the changes made to them outside the tree are not reflected back. So in case ofdoing a custom node update first the node has to be retrieved then modified. At the end it has to be pushed back to the tree. The lazy loading works well. The node has to be marked as lazy loadable and the callback function is called when it is expanded.

This tree also has a number of methods and features which were not mentioned as they are obsolete for this project. The ‘events’ are worth of meantioning out of these features.

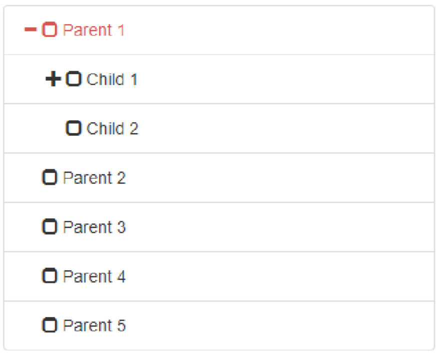


Figure 5.5: Example of the Patternfly Tree View checkboxes and select

There are always generated events on user interactions. If it is needed, the fired events can execute custom functions.

The performance test showed that this library is focused on cutting down the rendering time at the cost of more scripting time. As a result the tree rendered with similar speed on both tests. On the first test the rendering finished at 3675.7ms and the rendering time was only 8.9ms. For more details see the chart Figure 5.6.

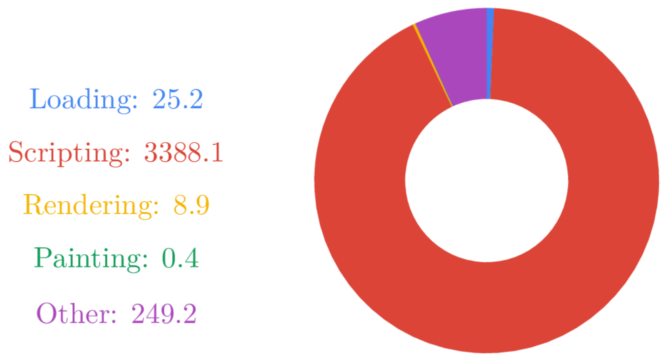


Figure 5.6: Patternfly first performance test

However, during the second test the displayed node number increased shockingly, still the library finished the rendering at 4104.8ms. The rendiring time was almost identical to the first one (9ms). Only the scripting time increased a little. Most probably this library would perform well in case of rendering bigger trees. For more details see the chart Figure 5.7.

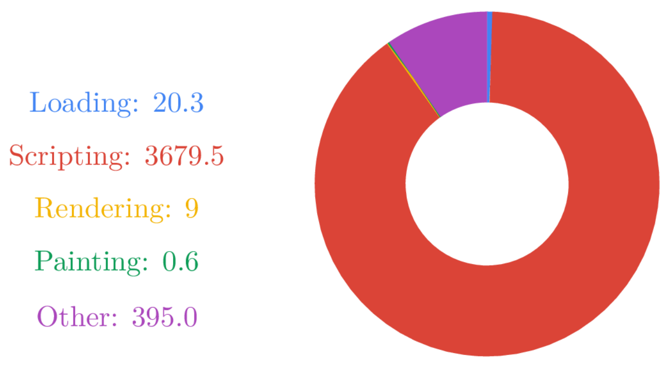


Figure 5.7: Patternfly second performance test

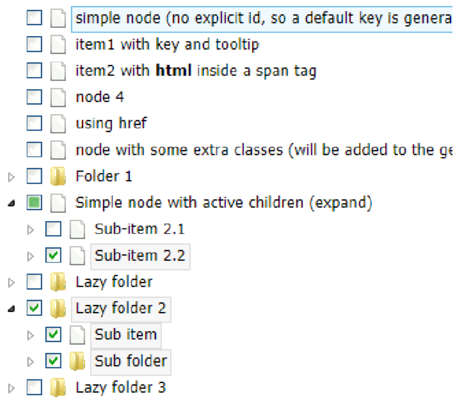
5.1.3 Fancytree

Fancytree [13] is from the same author as the Dynatree. Lots of the features of Dynatree has been implemented in this tree view too. On the downside, the used libraries stayed the same. FancyTree uses jQuery similarly to the other two trees. For all the features, including custom themes, another dependency has to be loaded. It is called fancytree-alldeps. The documentation for this tree is extensive and contains lots of demo setups to see how certain things are done.

The library implements the checkboxes, however, the hierarchical check does not works as it is expected in this project. When a node is checked but it is unfolded then their children are not checked. The selection functions differently as well. There can be multiple selected nodes, each one at different level in the tree. To be fair there is a state, called active node which behaves like our definition of the selection. This library, similarly to Dynatree, does not implement the prevent deselect and allow reselect functionalities. The selected and checked nodes can be retrieved from the tree as an array of nodes with methods.

This tree view has themes and much more than Dynatree. For creating custom themes there is a guide. Despite this fact, when initializing the tree there are only a few options which can affect the look of it. Node and folder icons are customizable and the custom node class is available. but that is all the user can change. The tree with the default theming, enable checkboxes and with multiple selected nodes is presented in the image Figure 5.8.

The Fancytree was the best with its 747.4ms total time on the first performance test. Even thoughthe rendering time was higher, it can still be considered low with its 49.5ms. The complete test result is shown in the chart Figure 5.9.



simple node (no explicit id, so a default key is genera
item1 with key and tooltip
item2 with **html** inside a span tag
node 4
using href
node with some extra classes (will be added to the ge
Folder 1
Simple node with active children (expand)
Sub-item 2.1
Sub-item 2.2
Lazy folder
Lazy folder 2
Sub item
Sub folder
Lazy folder 3

Figure 5.8: Example of the Fancytree with default theme, checkboxes and select

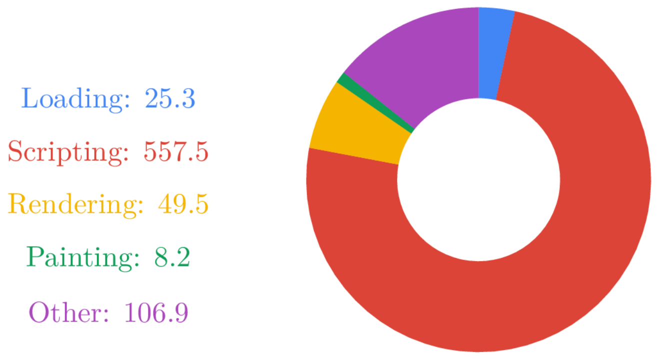


Figure 5.9: Fancytree first performance test

In spite of the fact that this library won the second performance test as well, the difference was not remarkable. The total time was 3616.8ms and the rendering time stayed on the same level. This indicates that most probably it would handle bigger trees well. The complete result for this test is also displayed as a chart, see Figure 5.10.

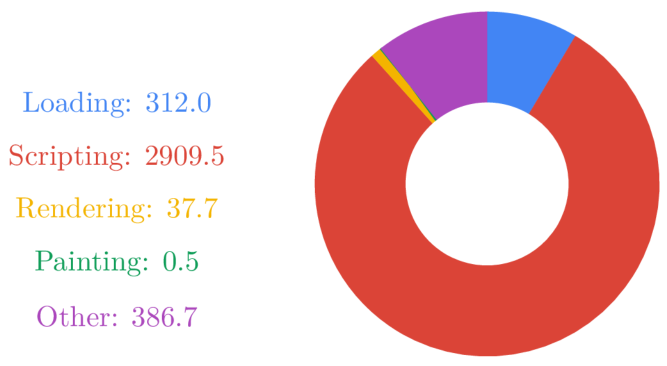


Figure 5.10: Fancytree second performance test

5.1.4 Summary

The Table 5.1 shows the comparison of the three analyzed libraries. The rows in the table are the categories explained at the start of the section. As it was predictable only the Patternfly Bootstrap Tree View has all the required features. The results of the data manipulation show that there is one common problem with all of the tree views. They use methods to manipulate the data. When the passed data is changed outside the tree, none of them reflects it. The Dynatree performance results were miserable but both Patternfly Bootstrap Tree View and Fancytree done well, especially with the small rendering time. The next drawback of these libraries is that all of them are jQuery dependent.

The performance test may show that Dynatree is not recommended to use when rendering a bigger tree. The other two libraries performed well on the test and most probably they can handle bigger datasets. Using other modern technologies and by excluding the jQuery dependency, probably the performance could have been improved.

All of these libraries are at least a few years old and there are no existing modern components to replace them. The major problem, the data change reflection, can be solved by using one or two way data binding. Most of the modern frameworks allows it, however choosing a framework limits the component's usability to the framework. By using frameworks the jQuery dependency also could be removed to reduce the required code size.

Features			
	Dynatree	Patternfly	Fancytree
Expand/Collapse	true	true	true
Checkbox	true	true	true
Hierarchical check	true	true	true
Node select	true	true	true
Prevent reselect	false	true	false
Allow reselect	false	true	false
Per node option	true	true	true
Styling	icons	icons, colors, classes	classes, theming
Data manipulation			
	Dynatree	Patternfly	Fancytree
Reflects data change	false	false	false
Changes given data	true	true	true
Lazy loading	true	true	true
Depends on	jQuery, jQuery-UI	jQuery, Bootstrap	jQuery, jQuery-UI
Performance tests			
Collapsed tree	Dynatree	Patternfly	Fancytree
Total time	1062.1ms	3675.7ms	747.4ms
Rendering time	79.3ms	8.9ms	49.5ms
Expanded tree			
Total time	14160.7ms	4104.8ms	3616.8ms
Rendering time	4774.3ms	9ms	37ms

Table 5.1: Summary of the researched tree views

Chapter 6

Implementation

This chapter provides a step by step description on how the tree view component was developed. The final implementation is able to render the tree, display checkboxes, expand nodes and it also supports hierarchical check and multi-select as well.

The process of the implementation can be divided into three main parts according to the problems to be solved. The first of these is the storing and manipulating the data. In this part, the React's top-down data flow and the 'single source of truth' had to be taken into account. The second main part is the rendering. The main requirement for the component is to render as fast as possible, therefore the faster rendering of the elements was the priority. The last, but maybe the most important part is the testing. This part included unit tests and it was done with the help of Jest library [5] which works well with React. The testing is described in the Chapter 7 in detail.

6.1 Used Tools and Development Setup

Based on the comparisons in the previous chapters, the React framework and the TypeScript language have been selected for the implementation. TypeScript allows typing in JavaScript therefore the code becomes more secure and readable. This does not mean that the applications should use TypeScript by using the component. The code is translated to plain JavaScript and the typings can be included separately when needed.

A server and a compiler were needed for better development. For this purpose, the 'create react app' [4] was used. It provides environment for running a React app on the development server and an option to compile both the component and the React code to the production version. Furthermore it has already set up the Jest testing library. The component is ready to be developed right after setting up the App file.

When working with the 'create react app' all the required tools are set up, including TypeScript. However to ship the component as a package, it is needed to compile it separately. To do this another library was needed, called 'webpack'. When it is properly set up, the webpack can compile TypeScript code into JavaScript. To optimize the size of the final package, the webpack can minify the code. This results in a mostly unreadable code as it removes all whitespace from JavaScript and CSS files. Two files are created after compilation. One is the component's JavaScript code the other is the CSS file with the CSS rules necessary for the component. These two files can be delivered as a package allowing other applications to use the component.

6.2 Features

The component has plenty of options which can be set when rendering the component. These options can change certain functionality or change the look of the tree. First the functionality changing options will be described. Then the options for the styling follow.

6.2.1 Functionality

The functionality can be divided into three groups: the checkbox, the node selection and the lazy loading. Let me start with the checkbox. There are three options which affect the checkbox: the 'showCheckbox', the 'hierarchicalCheck' and the 'checkboxFirst'. The last one rather belongs to the styling than to the functionality as it ensures the order of the node icon and the checkbox icon.

The 'showCheckbox' option enables the checkboxes. This option can be overridden on node basis. If the node has the option 'hideCheckbox' set to true then on the specified node it will not show the checkbox. If the 'hierarchicalCheck' option is set to true, it ensures that in case of changing the checkbox value of a node all its children copy this value recursively. Therefore, all their parents are set to the appropriate value. The parent values can be *checked*, *unchecked* and *partially checked*. The node is partially checked if either it has at least one partially checked child or when its children have values both checked and unchecked.

On node basis there is more option than the ability to show or hide the checkbox. The node can be set to ignore the checkbox changes in two ways. One of the options is to disable the entire node. This option is passed to the state of the node. However, this option disables all interaction with the node. The other solution is to set the 'checkable' option to false which is set to true by default. This option only prevents the checkbox change. None of the above mentioned options prevent the node checkbox change when it is changed because of the hierarchical check. This means, if the hierarchical check is enabled then changing the parent node will affect even the child nodes which are disabled or non-checkable.

The selecting has multiple options which can be controlled. The options connected to the select are the 'multiSelect', the 'preventDeselect' and the 'allowReselect'. Individual node selection can be disabled only with the disabled state. If the user wishes to disable only the selection, then it can be done by ignoring all the callbacks on data change for select.

The most obvious option is the multi select. This option is set to false by default. This causes that only one node can be selected at a time. This is ensured in two steps. First when initializing the tree and there are multiple nodes set to be selected, then only the first one remains selected. The second step ensures the deselection of the previous node when a new one is selected. If the multi select is set to true then the previously mentioned functions are left out. This allows the user to select more than one node.

The 'prevent deselect' option is used when the user wants to ensure that there is always a selected node. When this option is set to true then clicking on the selected node will not set the node to the deselected state. If the allow reselect is turned off, nothing happens if the selected node is clicked. However, if it is active then clicking on the currently selected node fires the select event again.

The last discussed functionality of the tree is the lazy loading. This functionality makes possible to load the data for the tree node when it is needed. The node must meet two requirements for lazy loading. The node has to be defined as a lazy loadable node. This is done by setting the node's 'lazyLoad' option to true. This option also ensures that an expand icon will be displayed next to the node. Next to the tree the 'lazyLoad' callback function has to be passed. This function should contain one parameter, a node, and should return a promise about a list of nodes. When expanding the node this function is called and if the promise resolves then the result is displayed as a subtree in the expanded node.

6.2.2 Styling

In contrast with other solutions this tree does not offer a lot of properties for styling. The main concept of the styling is that the user does the styling through CSS rules in this component. The tree has two options, the 'changedCheckboxClass' and the 'selectedClass', in which the user can define which class to append to the node if the checkbox has changed or it has become selected. Then it can be styled in a stylesheet by creating rules for the class.

However, not all the styling would be comfortable with this style. Therefore the icons can be specified one by one. Each icon used in the component can be changed. In default the tree uses font awesome icons and therefore it needs the font awesome stylesheet as a dependency. Though if all icons are changed then this dependency could turn to obsolete. The icons which can be changed are the following: default icon for the nodes, checked checkbox, unchecked checkbox, partially checked checkbox, collapse and expand icons, loading and error at loading icons and the selected icon. On node basis, the user can always define custom node icon or image to replace the node icon or the selected icon. Also there is an option to add custom classes to the node.

As addition the ability to hide node icon is available through the 'showIcon' property on the tree. Furthermore, the checkbox and the node icon can be swapped with the 'checkboxFirst' option. The 'showImage' option gives the ability to select the preferred one between the node icon and node image. When set to false, even if the node has a custom image, the icon is shown.

6.3 Architecture

When creating the component the React's guidelines were considered. They encourage the developers to find the correct place for storing the data. This is in connection with the one way data flow. Another good advice from them is to use dumb and clever components. This means that if the developer builds the component, the top level component should contain the logic and the control of the child components. The other components should be dumb components. These dumb components should contain smaller parts of the objects and they should not contain any logic.

All of these guidelines were considered while the component structure was created. As the result, the next hierarchy of components was created: The main smart component: Tree. The Tree contains the Node component which does the rendering. The Node component uses two dumb components, the CheckboxButton and the ExpandButton. The Table 6.1 shows the list of components. The component right in a table belongs to its left neighbor.

Now it is time to describe the components. They will be described from down to up. The two dumb components will be the simplest ones. They should receive a few options like the current state and some callback function to let the parent component know whether the user clicked on them. These callback functions lead to their parent which is the Node component.

	Smart component	Main rendering	Dumb components
Component name	Tree	Node	CheckboxButton ExpandButton

Table 6.1: Tree View component hierarchy

The Node component should do more rendering than its children. The logic of the rendering should take place right here. This component should basically decide how and what to render. It should receive data about the node which will be rendered here. It also contains itself recursively and that way it keeps up the hierarchy in the rendered tree. The node component should also have callback functions from its parent, the Tree component. However this component should not edit the data of the node. Any user interaction should be forwarded to the Tree component. This does not mean that it should not do any logic. Some of the user input checking can take place in the component.

Now it is time to describe the smart component, the Tree. This component should process all data changes. After rendering the root node, all the user interactions should lead here. Then this component will decide what happens with the data. If the data should change, then this component changes it. The tree initialization should also be done here. This component is also the component which is used to create the tree, therefore it should receive all the options and data needed for the setup.

To sum up the component hierarchy it can be stated that sooner or later all changes end up in the Tree component. If this change was made on one of the dumb components, Node component can make additional logic before the change gets to the Tree component which processes the changes.

6.4 Code History

The right place and format to store the data was one of the most important parts of the component. In the first solution, the React's top-down data flow was not taken into account. This version of the code manipulated data inside the component as states and the props were used only as the default values. The problem of getting the data out and the fact that this solution used an anti-pattern ended the development of this solution rather quickly. The lesson was learned: the data should be stored in one big hierarchical object.

Now that was clear that the data should be stored in one place, its place had to be decided. The node component was out of the question. Up to this time there was another component with name List. The list was another wrapper around the nodes. In the end, the object was stored in the Tree component and as the List wrapper component became obsolete, it was deleted. The list rendering was moved to the Node. At this point, it became clear that the Tree component will be the 'clever' one and the Node component will be the 'dumb' component containing the button components.

Now the data was stored in a class variable in the Tree component. It also contained a state variable for it too. This variable was required to force React to update the tree. The data flowed in the following way: First the Tree component got the data through the props. It was saved from the props to the class variable. Any changes to be made on the data before rendering were done on it. Then the state variable was set from this class variable. If anything was changed in the Tree or the Node, then the Tree component data was modified in the class variable first. Then when all changes were done, the state variable got updated as well.

Using one object and passing it down to the tree added one more problem. The states used in the Node component were initialized from the pops. If no value was supplied, then it used a default value. All values had to be initialized in the Tree within one object. For this purpose a method was created which recursively initialized all the values in the object. This particular case however has another solution. This second solution only initialized a node if its value was needed. It had a price of one additional variable and checking its value at nearly every change. For these two solutions a benchmark was created that is mentioned in Section 7.2.2. Manipulating this data from outside would be allowed by methods implemented in the Tree component. This solution however did not give much freedom over the data and still had to update the state in the Tree if got new data in the props. So in the end this solution was also considered inappropriate.

6.5 Data Manipulation

Compared to the previous version a state lifting had to be done. That means the data is not stored anywhere in the Tree component. Instead it is stored somewhere in the parent. This trade-off resulted in two things. The positive outcome is that the user has full control over the data. This means that any changes made on the passed data outside the tree is reflected in the tree. On the other hand the user should initialize and then update the data even when it was changed from the tree.

The initialization is required to make sure that IDs of the nodes are correct and all required members are initialized. These members are the children nodes and the status variables: expanded, disabled, checked and selected. The other variables cannot be changed in the tree. Their default values are defined with the help of React's default values to the component. The IDs are important because they indicate the node's position in the tree. Every ID is constructed by the next pattern: numbers joined by '.' and every number indicates the index of the node in the parent's array of children. Meaning 1.2 would indicate that the owner of this ID is the second root node's third child. The IDs therefore are important as all searches are done with the help of them. This method ensures that all queries on the nodes would have linear time complexity. The searching algorithm is described in the Section 6.6. The user can initialize the tree by himself but it is much easier to use the Tree component's public static function, created exactly for this purpose. It can be used multiple times on a data set if required and on subtrees as well. In the case it is used for initializing subtrees, the parent ID has to be passed to it. For selecting the nodes from an initialized tree can be done by the static 'nodeSelector' public method.

The tree at every change calls the callback function, namely the 'onDataChange' which has to be passed to the component. This way the user can launch custom actions if something changes in the tree. However, the user must provide the appropriate data changes for some events. When the node status changes (expanded, checked, disabled, selected), then an appropriate function should set the values of the nodes from the callback function. For this purpose there are five helper functions which do just this. Four of them can be called when the status changes. The fifth should be called to update the changed node reference in the tree as all changes are immutable. Therefore we have to update the node references in the hierarchical object. If no or wrong responses are given to an event, the tree may show unexpected results but most likely it will not do anything.

Another feature of the Tree where the data is changed is the lazy loading. This feature allows passing data to the nodes after initialization. When the lazy loadable node is expanded, the supplied callback function will be called. This callback function should return an array of nodes - the children of the expanded node. This data should be initialized, as the data passed before if the tree is about to behave correctly. In matter of fact all changes are made outside the tree. If the user wishes he can implement its own lazy loading function which would do the same. This is not recommended.

Additionally to the data manipulation some trade-off were made. Lots of the algorithms crawled through the tree, therefore they had to be made as fast as possible to cut down scripting time. The most fruitful was the proper usage of the loops. For looping through array of objects JavaScript offers three methods. The most specific is the *map*. This method is the most readable and easy to use. The second one is the *foreach* loop, a common function looping trough items in an array and quite readable and straightforward. The last method is the *for* loop which needs more typing than the others and sometimes it can confuse when accessing the right element in an array. The catch is that *foreach* is two times slower than the *for* loop and the *map* is nearly ten times slower. Knowing this all the loops in the components are converted to a *for* loop.

Another optimization was made with the help of tail-call when using recursion. Not all the methods can be converted to use tail-call. Functions using 'tail call' are faster and use up less memory. Therefore it is recommended to use it. In this case two functions were converted to tail call which are describe in detail in the Section 6.6.

6.6 Algorithms

This section contains the algorithms and their description. There are a few algorithms that are worth to mention. One is the node selecting algorithm, the second handles the node selection and the last one handles the checkbox change.

The node selecting algorithm is relying on the node IDs which are appended to the node at the tree initialization. These ids indicate the node's place in the tree. They are array index numbers divided by an '.' character. For example, a valid format of an id would be '1.5'. This indicates that the node is the sixth (starting from zero) node of the second root node. This algorithm practically divides the id into numbers and then sequentially dive into the nodes until it reaches the specified node. The algorithm is quite short and can be viewed in: Listing 6.1. With this solution it selects nodes on the same level in the tree with similar time. The time complexity is increasing with the deepness level of the searched node.

```
let path: number[] = id.split('.');  // Splitting the node ID

let node = tree[path[0]];     // The first root node is set.
for (let i = 1; i < path.length; i++) {
    node = node.nodes[path[i]];  // Diving into the tree.
}
return node;
```

Listing 6.1: Node selecting algorithm

The other two algorithms are more complex. To start with the simpler one, the node selection handler will be described. This algorithm has to decide when to change the selected state of a node. It gets the new state of the node (selected or deselected) and the node. Then it has to consider all the passed props and decide whenever should or should not change the state of the node. The flowchart Figure 6.1 describes how the algorithm works. There are only nested if-then-else code blocks. When the flowcharts say change the real function, it calls the specified callback function which is passed to the Tree component to change the data.

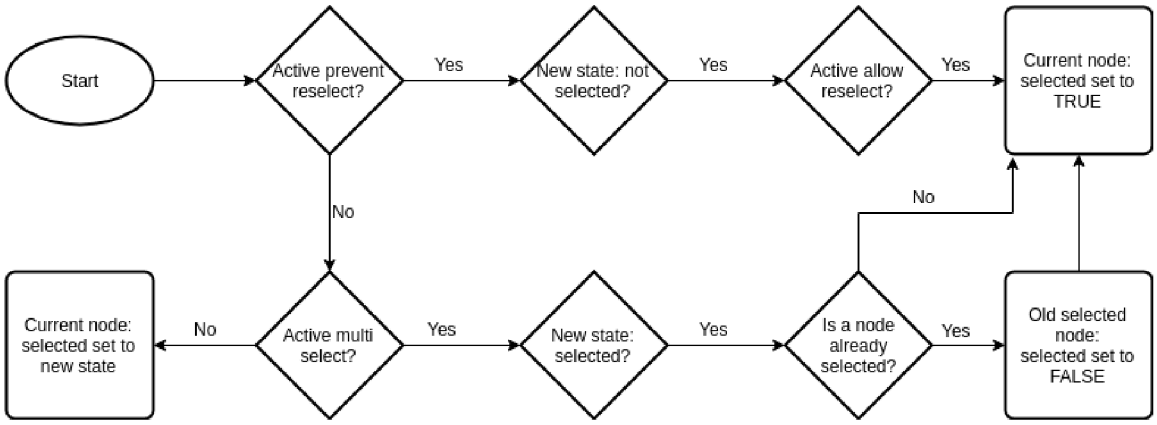

Start
Active prevent reselect?
Yes
New state: not selected?
Yes
Active allow reselect?
Yes
Current node: selected set to TRUE
No
No
Current node: selected set to new state
No
Active multi select?
Yes
New state: selected?
Yes
Is a node already selected?
Yes
Old selected node: selected set to FALSE

Figure 6.1: Flowchart diagram of handling select change algorithm

The last of the algorithms is the most complex one. It spans across two recursive functions. Both of them are using tail call. Tail call is not optimized in ECMAScript 6 (latest JavaScript version), but it is a feature yet to come. When the hierarchical check is turned off, then only the current node has to be set to checked or unchecked. The problem comes when it is turned on. In this case all the node's children should be changed. Also if it is not a root node, then all its parents up to the root node should be updated.

Let me start with the first part, updating the children nodes. On the input side there are three values needed: the changed node, the new state and the information whether the node was selected directly or it was called in the recursion. Then the flowchart looks like this: Figure 6.2. It shows that if the node is directly changed and the hierarchical check is enabled, then the node first makes the call to the other function to change its parents. If the node has children, the function calls itself for all of the children nodes, but with false 'changed directly' option. This ensures that the children nodes will not want to update their parents.

The second recursion updates the parent nodes. This operation costs a great deal. The parent can take on three states: checked, unchecked and partially checked. Node is partially checked if at least one of its children is checked or partially checked and also there is at least one node unchecked or partially checked. The Figure 6.3 flowcharts show how the decisions are made.

To start with the recursion ending condition is added. The recursion ends when it is called by a root node. Then it is checked if the child which called the function is partially checked. If it is so, then the parent node is partially checked and this information is stored. If it is not, then all its children are iterated over. The iteration ends when a child node is partially checked or there are no more children. After the iteration the parent node's new state is decided. If the counter equals to zero, then the parent is unselected. If it equals to the number of children nodes, then the parent is selected. Otherwise the parent is partially selected.

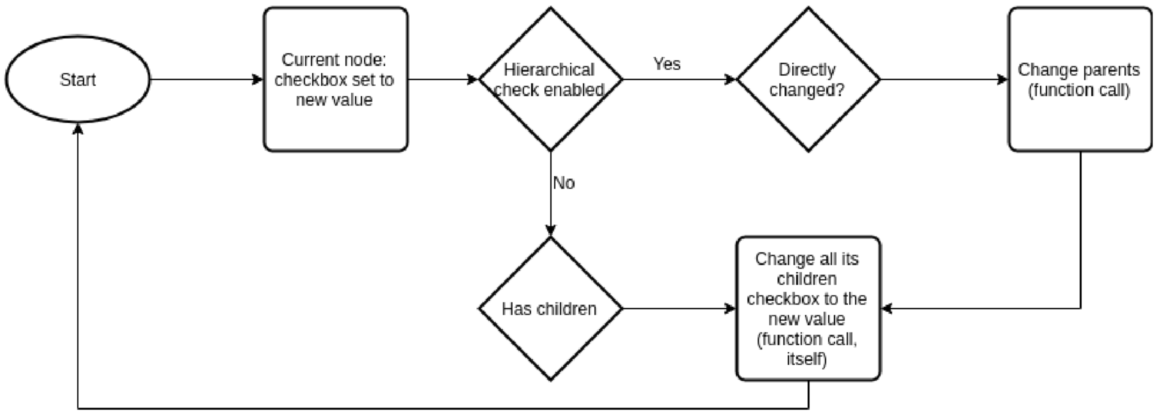

Start
Current node: checkbox set to new value
Hierarchical check enabled
Yes
No
Directly changed?
Change parents (function call)
Has children
Change all its children checkbox to the new value (function call, itself)

Figure 6.2: Flowchart diagram of changing node's and its children's checked state

At the end of the process, there is a check if the parent node has the same old checked value as the new one. If the condition is fulfilled, there is nothing to do. No new callback should be created to change the value. It would not have an effect and would slow down the process. Otherwise the data would be changed. At last the function calls itself for the next parent, this way creating the recursion.

6.7 Rendering

This section discusses another important part of the component. The main feature of this application had to be the rendering performance. React keeps a shadow DOM which consists of JavaScript objects. These objects are fast to update then React takes care about the synchronization of the real DOM and its own shadow DOM. However on the page load all elements are rendered which are visible. Therefore the main goal was to render as few elements as possible.

The first problem was indenting the list elements. The most obvious solution would be to use nested HTML DOM list elements. This would generate twice as many elements than without it. The second option and the more common one was to use span elements to indent the list items. Commonly span elements are rendered faster than list elements. This, again, did not seem to be very appealing. The problem with this solution is that if the most of the elements are very deep in the hierarchy, then hundreds and hundreds of these span elements could be generated. The last solution was to indent with CSS rules. This way there is no additional element rendered and it can cut on times. The created benchmark for this problem tested two version of the CSS indent. First was a CSS file filled up with some rules. The second one generated the CSS rules dynamically. The benchmark is described in Section 7.2.2.

The results were close to the expectations. The CSS indent won over the other solutions. The difference was so little between the two CSS solutions that using the predefined file over generating the rules was obsolete. Therefore the used solution was the generated CSS rules indent. It uses two helper functions. First is to determine the deepness of the tree. This is but a simple recursive function which returns the max depth of the tree as a number. The second function needs the depth of the tree and generates the minified CSS rules and returns them as a string. This string is injected at the end of the tree.

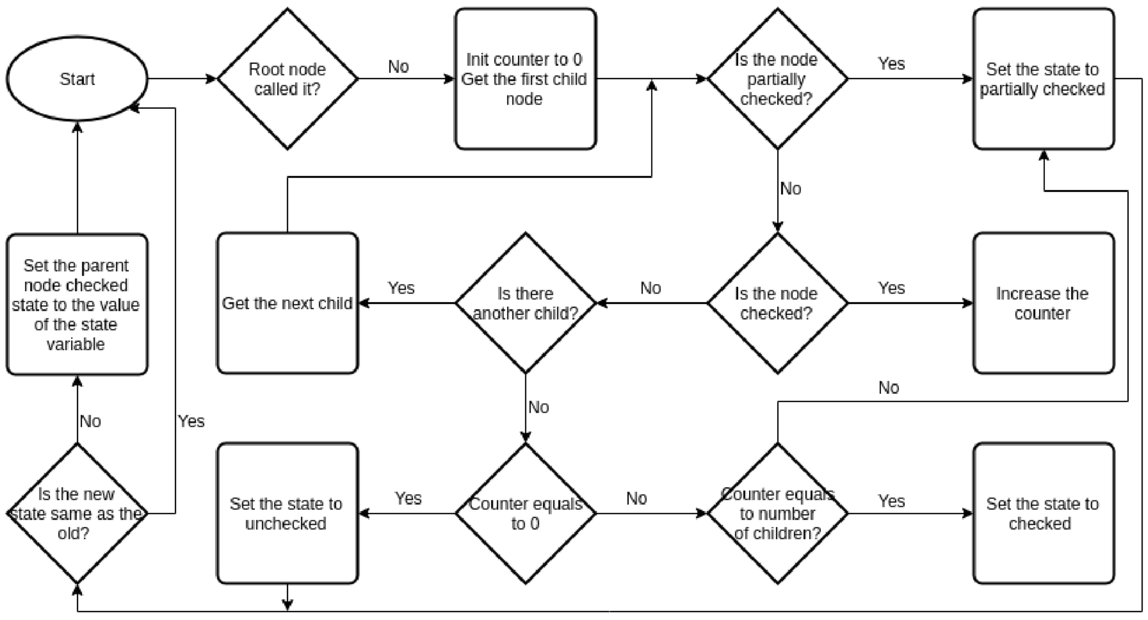

Start
Root node called it?
No
Init counter to 0 Get the first child node
Is the node partially checked?
Yes
Set the state to partially checked
No
Set the parent node checked state to the value of the state variable
Get the next child
Yes
Is there another child?
No
Is the node checked?
Yes
Increase the counter
No
No
Yes
No
Is the new state same as the old?
Set the state to unchecked
Yes
Counter equals to 0
No
Counter equals to number of children?
Yes
Set the state to checked

Figure 6.3: Flowchart diagram of changing parent nodes checked state

In React every component can render only one root element. Now that there was one list then the list items had to be rendered after each other. However rendering all elements in the Tree component would not be a wise choice and would bring complications. For situations like this there is a solution provided by the React: by using as root element the 'React.Fragment' more elements can be rendered in the component without rendering extra wrappers in the DOM therefore cutting rendering times.

Dealing with the expand button or more precisely the lack of the expand button yielded another difficulties. When the children nodes do not have children then the expand button should not be displayed. This results in smaller padding to the list item and the whole node gets lined up with the parent. There were two possible solutions. The first solution would insert a placeholder element when there is no expand button. The second one would maintain the space before the element by CSS rules. It is clear from the previously mentioned benchmark that CSS rules are faster to render. The second solution was preferred even though the 'padding-left' property was already used for maintaining the indent of the list items. Therefore a trade-off was made: using the last CSS property, the 'margin-left' property, to fill up the space which is created by the lack of the expand button. By this solution every option for further aligning the list items with CSS rules is taken away from the user.

The only solution which is no entirely optimal is the text in the list items. At first it was not wrapped into a 'span' element but later it had to be added. The reason for this change came with the node selection feature. When the on click event for selection was the list items property then it fired even when the expand or the checkbox button was clicked. This unfortunate situation had only two solutions: either Moving the buttons outside the list item or adding an extra element around the text and move the on click event to this wrapper element. The first solution was inappropriate, because icons hanging around with the list items generally considered as a wrong concept. Therefore, the other one had been implemented.

6.8 Packaging

The npm is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. The built component also depends on tree of the existing npm packages. In order to being able to use this component easily and without any setup, it had to be published as an npm package.

To publish to the npm a few tasks had to be done. The first of these things was the compilation. In order to be able to use the package without typescript it has to contain plain JavaScript code. To achieve that, the 'webpack' utility was used. With correct setup it compiles and minifies the typescript code into the JavaScript equivalent. This process and the package needed an entry point to the component. This was also added to the component.

The next step was required to protect the code. The code is published under an MIT license. The license file had to be added in order to apply and the appropriate field had to be set in the package setup file. The last step before publishing connects to the package setup file as well. Properties like author, entry point, version number had to be set.

After all these steps the package was ready to be published. It needed an account to publish under so after registering and signing in the package was published under the name 'react-wooden-tree'. The component is accessible to the public both on GitHub[1] and as an npm package[2]. The github reprository contains the demo app and the benchmarks too, however on npm there is only the compiled component and the necessary files. All instructions how to use the component are described in the readme file.

[1]https://github.com/brumik/react-wooden-tree
[2]https://www.npmjs.com/package/react-wooden-tree

Chapter 7

Testing

This chapter is about the unit testing, the automatic benchmark script and the tested and compared versions of the tree. During the developing process the testing is one of the most important parts because a properly tested code means less maintenance time. Automated tests mean lesser bugs on modification. The performance tests and benchmarks are helpful during the development if a decision has to be made. The present chapter describes the tests in more detail.

7.1 Unit Tests

The Jest testing library [5] was used for the testing purposes. This library was developed to test effectively and easily JavaScript code. A unit test can be created by the *test* or *it* functions. These two functions are equivalent to each other. The *it* keyword fits better and not only because it is shorter but because the test should be defined as *it („should do something“, function())*. This makes the tests more readable. The *expect()* function should appear at least once in the function to match values. If all the *expect()* functions are passed, then the whole test is passed, otherwise the test ends with a failure.

Jest has the ability to create and test against snapshots. This means that a snapshot is created at the first run. All the following runs are tested against this first (or later updated) snapshot. In this test suite, these snapshot tests were kept at a minimum because any change in the rendered tree can result in multiple test failure and requires a snapshot update. There are more ways to use Jest, like the mock functions but they are not used in this project.

The need of asynchronous testing rises when the lazy loading function had to be tested. Jest provided it in multiple ways. At the end the ’asynch’ and ’await’ keywords were chosen to wait while the promise became fulfilled and then tested if the tree was corresponding to the correct form.

The test suite was extensive and reached 100% coverage as for the line coverage as well as for the decision coverage. This does not mean that the code is bug free but lowers the chances of an error and excludes the chance of some of the common bugs.

The testing is divided into three parts and each part has its own ’describe’ wrapper function. The first part contains tests of the public functions of the Tree component. These are the static functions and they are responsible for changing the data in the tree and then return the new data. The second part contains test of the node rendering. They are in the Node.test file. These tests are responsible for the correct rendering. Only the

Node component renders the elements directly, therefore it was logical to test it here. The test suite includes the ExpandButton and CheckboxButton test as well. The test cases should be straightforward and readable.

The third and at the same time the most important part is to ensure the correctness of the events which are reactions to every change made internally. This was the most difficult task. Let me remind you, the number of snapshots should remain at the minimum. Enzyme is used with Jest most of the time. Enzyme can render components shallowly then it has a plenty of methods to make the element selection simple. However, it was not used in this project as there was no specific need for it. Despite this fact a few functions were created to help when selecting the elements from the JSON formatted rendered object. The following 3 functions were created: 'liSelector', 'childrenTypeSelector' and 'childrenClassSelector'. The 'liSelector' returns the n-th list item element from the given object. The other two functions are working with the objects returned from the 'liSelector'. One selects the element from the object which has the specified class; the other function selects the element which has the specified type. There is another function which simplifies and unites the interface to these functions, the 'childrenSelector'. By using this function it is quite easy to receive the reference to a button or to the text from a list item.

Three different trees are created during the setup phase. These are hard to change as they are used in multiple tests with multiple purposes. The reason of initialization as global variables is to type less and make tests more readable. The callback function was implemented for the events. It does all of the required changes. In addition, to be able to test the events, the callbacks modifies two global variables which are reseted in the setup. The first variable is a counter. This counter indicates how many times the callback function is called. In other words, how many events are fired. The second one contains the last callback values. It is enough to save these two pieces of information to determine whether the component behaves correctly.

Writing the tests becomes nearly trivial task with the above described helpers. However, reaching the 100% coverage was still not easy. During writing the tests, problems with undefined values were discovered. These were corrected and the uninitialized values were initialized. Despite the undefined values the component worked as expected. In addition, it would have been better and more precise if the testing had been done by someone else than the creator. Unfortunately, there was no opportunity for it.

7.2 Performance

This section describes how the benchmarking was set up and the benchmark tests which were run. First, it shows how the benchmark automatization was done and how the tests look like. This follows the specific tests and the results.

7.2.1 Automated Benchmark

Firstly, a project with the help of 'create react app [4] was created, where both the production and the development compiler is ready. The main features were the 'npm run build' for the benchmarking which generates the production code and the 'serve' command which is used to run the server on localhost.

Then the Google's Lighthouse [3] was added because it is capable to analyze the web page and then generate JSON file from the results. There were specifically two values which are helpful: first meaningful paint and the first interaction. Since the react starts to interact when the whole app is ready, it was meaningless to observe the complete interactive value. First Meaningful Paint is essentially the paint after which the biggest above-the-fold layout change has happened, and the web fonts have been loaded[1]. The First Interactive metric measures when a page is minimally interactive[2]:

- Most, but maybe not all, UI elements on the screen are interactive.
- The page responds, on average, to most user input in a reasonable amount of time.

And lastly a tool was needed to extract the data from JSON. Jq which is a lightweight json processor has been chosen. In order to ease the installation the npm version was used. As a result everything can be installed with ‘npm install‘ this way.

Logic and Folder Structure

To feed different trees to the application, the Generator has been created. It has a single function: to generate different sized trees which can be reused in different App files. These generators have become the test cases. All the benchmarks are tested with all of these generators. The following generators are available:

- All nodes expanded tree, 3 level deep
  - 27, 125, 1000 elements in total
- All nodes collapsed tree, 3 level deep
  - 27, 125, 1000 elements in total

Each version of the code has to contain the component folder and the App file. The App file is needed because the interfaces can differ from each other as the component gets more developed. The supplied App file can be carefully checked if the benchmarks are running on the same settings (like checkboxes and icons).

Before starting the benchmark script, the server has to be started with ‘npm run prod‘ or with the command ‘serve -s build‘. This command starts the server on localhost: 5000. Then the benchmark.sh can be started. It needs one parameter and that is the folder name of the benchmark. The benchmark script does the next steps: for each subfolder (in the folder which we specified as the parameter) it copies the component folder to the *src/* folder. Then it builds and runs the lighthouse with each generator. Then it gets the relevant data with the help of jq, the JSON processor from the generated lighthouse JSON and stores them in the file results.txt.

7.2.2 Benchmarks and Results

This section contains the actual benchmarks – the difference between the code versions, and the benchmark results for them. Each benchmark group contains a summary where the choice of implementation is stated and reasoned.

[1]https://developers.google.com/web/tools/lighthouse/audits/first-meaningful-paint
[2]https://developers.google.com/web/tools/lighthouse/audits/first-interactive

Nested List Indent

The first comparison was about how to implement the list item indent. There are five different approaches that can be taken into consideration. The most obvious one is to maintain a hierarchical list with the *<ul>* elements (nested lists). The others contain only one *<ul>* element and all the nodes are inside it. As a result the indent is solved by other means.

The second approach creates the indent by inserting extra elements before each <li> (span indent in the results). The node's depth was counted in the hierarchy and then '<span>' elements were inserted which indented the list item accordingly - however it carried the problem that deeply nested elements would generate too many DOM elements.

The third approach indents the item with the help of in-line styling. It calculates the padding size in pixels depending on the node's depth and then inserts it as in-line style for each node. In-line styling is when the CSS rules are inserted to the 'style' attribute of an HTML element.

The last two approaches used CSS classes which were assigned to the elements (as class indent in the results). The difference between these is the following: The first approach manipulated the predefined CSS rules while the rules are programmatically generated depending on the tree depth at the startup in the second one. This means that the rules are generated in a function and returned as a string. Then this string is injected into the HTML with the CSS rules. The problem with the first approach is that if the tree gets too deep then it can run out of the classes. On the other hand the second one's drawback lied in the fact that we have to generate the CSS rules and inject it. This may make it really sluggish.

Summary

As the results show in the Table 7.1, the two fastest were the class indents. The predefined class indent was slightly faster but not enough to be preferred over the generated one. Choosing the generated class indent provides a good performance and ensures that all kind of trees will be handled.

Tree Nodes Initialization

The default values and structures are generated differently in this benchmark. The first approach was to initialize recursively the whole tree before being rendered. The second one manipulated with the initialization on demand. Meaning, if we want to use the node's value (render or change) then we call a function which initializes only the nodes we need.

In this benchmark there is another optimization: testing whether the static methods or the functions are faster. This is tested against the on demand test. One has the static class methods and the other has these methods implemented as a function in the same file along the class and each function is exported one by one.

Summary

The results in the Table 7.2 show us that the on demand filling is faster than filling the structure at the start. Assumptions are that with more and more nodes the on demand filling would be faster and faster than the other one. Furthermore, the benchmark showed that importing and using functions are slightly slower than the static class methods counterparts. However, at the end the structure moved to the outside source and the initialization

For expanded cases				
Name	Type	Gen-27	Gen-125	Gen-1000
Class indent	first paint	1634.3	1687.9	2088
	first interactive	1634.3	2654.9	3032.1
Generated class	first paint	1648.4	1695.2	2092.7
	first interactive	1648.4	2662.3	3040.3
In-line style	first paint	1641.7	1694.1	2149.7
	first interactive	1641.7	2661.9	3087.1
Nested lists	first paint	1639.1	1705.6	2238.5
	first interactive	1639.1	2664.3	3126.7
Span indent	first paint	1639.8	1724.5	2224.9
	first interactive	1639.8	2685.8	3157.0
For collapsed cases				
Class indent	first paint	1624.5	1605.5	1616.7
	first interactive	1624.5	1605.5	1616.7
Generated class	first paint	1612.3	1616.3	1640.9
	first interactive	1612.3	1616.3	1640.9
In-line style	first paint	1624.2	1620.5	1658.6
	first interactive	1624.2	1620.5	1658.6
Nested lists	first paint	1611.3	1626.6	1641.2
	first interactive	1611.3	1626.6	1641.2
Span indent	first paint	1601.3	1636.2	1660.3
	first interactive	1601.3	1636.2	1660.3

Table 7.1: Results of nested tree indent benchmark

For expanded cases				
Name	Type	Gen-27	Gen-125	Gen-1000
At startup	first paint	1715.6	1705.8	2097.4
	first interactive	1715.6	2668.9	3056.6
On demand	first paint	1650.3	1683.2	2073
	first interactive	1650.3	2650	3008.5
On demand functions	first paint	1699.8	1697.7	2016.8
	first interactive	1699.8	2662.1	3015.5
For collapsed cases				
At startup	first paint	1640.6	1618.7	1652.7
	first interactive	1640.6	1618.7	1652.7
On demand	first paint	1645.6	1622	1618
	first interactive	1645.6	1622	1618
On demand functions	first paint	1623.4	1642.4	1624.8
	first interactive	1623.4	1642.4	1624.8

Table 7.2: Results of tree nodes initialization benchmark

is done before passing the data to the tree. This means it has to be initialized before the usage. However all the helper functions were implemented as static class methods to boost the performance.

Chapter 8

Conclusion

The goal of the bachelor's thesis was to design and implement a tree view component. After the research of already existing implementations and available technologies, the component was implemented in TypeScript with the help of the React framework. Most of the implemented features were demanded by ManageIQ, the system from Red Hat.

When developing the component not only the functionality but the rendering time has been taken into consideration too. During the process of developing the component multiple versions were created and discarded due to the problem of storing and manipulating the data. The last version lined up with the React guidelines and allowed full control of the component data.

The advanced schematic of the language and the ECMAScript6 was used such as arrow functions, class exports and interfaces. The implementation of the component depends only on the React for which it has been built.

The result of the work is an NPM package which contains the tree view component for the React library. The component's main features are the fast rendering, the full control over the data and the customization. Some solutions required some trade-offs. In case of multiple choices, benchmarks were made to help in the decision making. The benchmark process was automatized for smoother development. In case of further development, they can be used as well. The final product was tested with Jest testing framework. The unit tests coverage reached 100%.

As the component is required by ManageIQ, its maintenance will not end with this thesis. The current version is reachable at GitHub repository[1] and as an NPM package[2]. Some newly proposed features are already under development, hovewer, they are not covered by this thesis. For example, the performance could be significantly improved if only the visible nodes would be rendered. Also the data handling will be implemented using the Redux library.

[1]https://github.com/brumik/react-wooden-tree
[2]https://www.npmjs.com/package/react-wooden-tree

Bibliography

[1] Blanco, R.: *Dialog Editor in AngularJS for ManageIQ*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2016.

[2] Google: AngularJS. 2018.
Retrieved from: https://angularjs.org/

[3] Google: Lighthouse. 2018.
Retrieved from: https://developers.google.com/web/tools/lighthouse/

[4] Inc., F.: Facebook Create React App. 2018.
Retrieved from: https://github.com/facebook/create-react-app

[5] Inc., F.: Jest. 2018.
Retrieved from: https://facebook.github.io/jest/

[6] Inc., F.: React Guide. 2018.
Retrieved from: https://reactjs.org/docs

[7] Krause, S.: JS web frameworks benchmark – Round 6. 2017.
Retrieved from: http://www.stefankrause.net/wp/?p=431

[8] Microsoft: TypeScript Handbook. 2017.
Retrieved from:
https://www.typescriptlang.org/docs/handbook/basic-types.html

[9] Mozilla; individual contributors: JavaScript. 2018.
Retrieved from: https://developer.mozilla.org/bm/docs/Web/JavaScript

[10] RedHat: ManageIQ. 2018.
Retrieved from: http://manageiq.org/

[11] Source, O.: Patternfly Bootstrap Tree View. 2017.
Retrieved from:
https://github.com/patternfly/patternfly-bootstrap-treeview

[12] Wendt, M.: Dynatree. 2013.
Retrieved from: http://wwwendt.de/tech/dynatree/index.html

[13] Wendt, M.: Fancytree. 2017.
Retrieved from: https://github.com/mar10/fancytree

[14] You, E.: Vue.js. 2018.
Retrieved from: https://vuejs.org/