ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση και σύγκριση Front-end JavaScript Frameworks: React.js & Vue.js

Διπλωματική Εργασία

Χρήστος Τσισλιάνης

Επιβλέπων: Γιώργος Θάνος

Φεβρουάριος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάλυση και σύγκριση Front-end JavaScript Frameworks: React.js & Vue.js

Διπλωματική Εργασία

Χρήστος Τσισλιάνης

Επιβλέπων: Γιώργος Θάνος

Φεβρουάριος 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Analyzing and Comparing Front-end JavaScript Frameworks: React.js & Vue.js

Diploma Thesis

Christos Tsislianis

Supervisor: Giorgos Thanos

February 2022

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων/πουσα	**Γιώργος Θάνος** Μέλος ΕΔΙΠ, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Γιώργος Σταμούλης** Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Χαρίκλεια Τσαλαπάτα** Μέλος ΕΔΙΠ, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Διπλωματική Εργασία

Ανάλυση και σύγκριση Front-end JavaScript Frameworks: React.js & Vue.js

Χρήστος Τσισλιάνης

Περίληψη

Το οικοσύστημα της JavaScript (JS) έχει αναπτυχθεί ραγδαία τα τελευταία χρόνια με έναν ολοένα αυξανόμενο αριθμό πλαισίων (frameworks) JavaScript (JSfs) που οδηγούν την υιοθέτηση λύσεων Singe Page Applications (SPA) σε υψηλά ποσοστά. Λόγω του μεγάλου αριθμού πλαισίων και βιβλιοθηκών με παρόμοια χαρακτηριστικά, η επιλογή της υιοθέτησης του καταλληλότερου JSf για ανάπτυξη διαδικτυακών υπηρεσιών έχει γίνει αινιγματική και προβληματική. Αυτή η εργασία επιχειρεί την αντιμετώπιση του διλήμματος επιλογής ενός πλαισίου από την προαναφερθείσα πληθώρα πλαισίων, εστιάζοντας σε δύο από τα πιο δημοφιλή και ευρέως χρησιμοποιούμενα JSfs, τα React.js και Vue.js. Συγκρίνοντας και αντιπαραβάλλοντας την υλοποίηση των πιο σημαντικών λειτουργιών τους και αξιολογώντας τις πιο κοινές ποιοτικές μετρήσεις που πρέπει να λαμβάνονται υπόψη κατά την επιλογή ενός πλαισίου, επισημαίνονται οι διαφορές και οι ομοιότητές τους. Στη συνέχεια, τα ευρήματα αναλύονται σε μια προσπάθεια να ρίξουν φως στο αναφερόμενο πρόβλημα. Η παρούσα εργασία επισημαίνει και κάνει συστάσεις με βάση τα ευρήματα της ανάλυσης, η οποία πραγματοποιείται από την οπτική των προγραμματιστών.

Στη συνέχεια εμβαθύνει στην ανάλυση και προχωρά στη δοκιμή των ευρημάτων με την υλοποίηση δύο εφαρμογών SPA σε κάθε πλαίσιο, για να επιδείξει τις βασικές λειτουργίες και να προσομοιώσει πώς αυτές υλοποιούνται και συμπεριφέρονται στην πράξη

Τα αποτελέσματα δείχνουν ότι κάθε πλαίσιο/βιβλιοθήκη έχει πλεονεκτήματα και τα μειονεκτήματά, ανάλογα με την περίπτωση χρήσης. Εν τέλει, θα πρέπει κανείς να εξετάσει το προς υλοποίηση έργο, και ποιοι παράγοντες είναι πιο σημαντικοί, ώστε να κάνει την βέλτιστη επιλογή.

Λέξεις-κλειδιά:

JavaScript; Frameworks; React; Vue; Comparison; Web Applications; SPA

Diploma Thesis

Analyzing and Comparing Front-end JavaScript Frameworks: React.js & Vue.js

Christos Tsislianis

Abstract

The JavaScript (JS) ecosystem has grown rapidly in the recent years with an ever-increasing number of front-end JavaScript frameworks (JSfs) leading the Single Page Applications (SPA) adoption in high rates. Due to the overwhelming amount of frameworks and libraries with similar features, the choice of adopting the most suitable JSF for SPA development has become puzzling. This thesis attempts an abstraction of the "problem" of the aforementioned plethora of front-end frameworks by focusing on two of most popular and widely used JSfs: React.js and Vue.js. By comparing and contrasting the implementation of their most important functionalities and evaluating the most common qualitative metrics to be considered when choosing a framework, their differences and similarities are pointed out. The findings are then analyzed in an attempt to shed light on the stated problem. The focus of the thesis is to point out and make any recommendations based on the findings of the analysis, which is performed from - and based mainly on- the developers' perspective. It then continues -and deepens- the analysis and proceeds to put the findings to the test with the implementation of two SPA applications on each framework, to demonstrate the core functionalities and simulate how these are implemented and behave in practice, in real life development procedure.

The results indicate that each framework/library comes with their different advantages and disadvantages overall, as well as according to use case scenarios. In the end, one has to consider the project at hand and which factors matter the most overall to make the optimal choice.

Keywords:

JavaScript; Frameworks; React; Vue; Comparison; Web Applications; SPA;

Table of Contents

Chapter 1 Introduction

1.1 Background

As the web evolves and takes on an ever-growing part of people’s digital lives-at least in parts of the world where (fast) internet connection is available-which is also growing in fast rates, the technologies that enable and support it grow with it. The “world” is now connected and the demand for the tasks that can be performed online, skyrockets. At the same time as the usage of the web shifts further to mobile devices each year [1], all the while those devices become more powerful in technical terms, by getting significant memory, processor and graphics upgrades each year, the demand for what can be done on the browser rises. This fact has in turn increased the complexity of web applications powered by JavaScript, resulting to the development of libraries and frameworks as a means to deal with new scaling demands and streamline the development process by removing a lot of duplicate work through abstractions [2]. State management, routing, data processing, and manipulation are just a few of the advanced themes that libraries and frameworks help developers handle. Many of today’s frameworks are adopting single-page application principles [3] which enhances the browser experience of the user, and tends to become the new industry standard. The wide adoption of SPA has been fueling the development of an ever increasing number of front-end frameworks, which results in an overwhelming plethora of choices available to the developer or company that has to make a decision. The need for fast and informed decisions concerns everyone from the individual to the enterprise and in between, and is of high importance, as monetary, time management and decision-fatigue factors are involved in the process. This project intends to provide a better understanding of how to evaluate frameworks qualitatively, and then compare two popular JavaScript frameworks based on the evaluation findings, thereby adding to the public discussion in a –attempted- constructive and beneficial manner.

1.2 Problem Statement

The aim of this project is to compare two popular client-side JavaScript Frameworks and provide insight and perhaps a blueprint about how to choose the appropriate framework. For this to be made possible, a better understanding of how one can evaluate frameworks depending on different factors has to be gained, for which an investigation about which factors to use when comparing frameworks is necessary. Such an investigation will contribute in giving an overview of what and how to compare, when comparing frameworks, as well as a justification on how to select which frameworks to compare. Following, this thesis will conduct an analysis of two popular client-side web applications frameworks based on the chosen factors, by first evaluating them through the comparison and contrast of their basic functionalities and features, and secondly by applying the frameworks on: 1. A SPA To-Do-list application 2. A SPA-MVC Authentication-Authorization with Registration and CRUD functionalities, in an attempt to capture the selected frameworks' development process.

1.3 Objectives

1. Provide insight for certain qualitative metrics as they are applied on- and are the basis of- the comparison process.
2. Derive conclusions about which frameworks implementation is proven to be the most beneficial for the developer for each one of the chosen metrics.
3. Re-evaluate, enrich and demonstrate findings of 2 as a proof-of-concept by developing two identical applications on each framework.

1.4 Related Work

Many attempts have been done to evaluate JavaScript frameworks in the past. One of the most common ways to evaluate the frameworks is by benchmarking using different metrics. Gizas, Christodoulou, and Papatheodorou tested a few prominent JavaScript frameworks, looking at their performance, quality, and validation. This was done by using specific test tools for this purpose [4].

Benchmarking frameworks, nevertheless, may not provide a realistic representation of the frameworks, as research has shown that JavaScript benchmarks are typically short and behave differently than real-world applications. Furthermore, benchmarks do not always reflect the same behaviors as real-world applications [5].

Surveys, such as those conducted by Pano, Graziotin, and Abrahamsson [6], can also be used to evaluate frameworks. The goal of their survey was to discover the factors that influenced developers' decision to use JavaScript. The poll was based on, among other things, performance and effort expectations, as well as the effects of social factors. The examples above indicate the necessity for and procedures for reviewing frameworks; nonetheless, the author discovered a lack of knowledge concerning when and what should be evaluated when evaluating related studies.

1.5 Scope – Limitation

Due to the project's time and size constraints only two frameworks will be reviewed. Furthermore, there will be only one programming language used, and fewer metrics are chosen for evaluating the frameworks than there would be in a more comprehensive study of a different magnitude.

The metrics researched in this study were chosen based on their popularity (how frequently they appear in related studies), and the metrics that would be more relevant to developers according to the author’s experience and research foundings.

Furthermore, the applications that are being tested are small and basic compared to complex, real-life production applications which would potentially introduce further implications and/or provide different conclusions.

1.6 Thesis Structure

This thesis content is organized into seven chapters.

- Chapter two presents the research process and applied methodology employed on this study including the data collection and analysis and the implementation strategy formulation.
- Chapter three presents basic concepts and technologies related to and surrounding JSfs and this projects’ particular comparison and apps implementation.

- Chapter four presents the process of choosing which frameworks to compare, and proceeds to the chosen frameworks’ presentation.
- Chapter five consists of the analysis of the evaluation process for defining which metrics to consider for comparison and the comparison analysis of the chosen frameworks.
- Chapter 6 presents the implementation of the applications and the analysis and comparison between them
- Chapter 7 presents the conclusions of the comparison process and discusses thoughts around the overall subject as well as on possible further enrichment of the comparison.

Chapter 2 Methodology

The research technique and methodology used in this study are described in this chapter.

2.1 Research Method

In order to accomplish objectives 1 and 2, a qualitative data analysis methodology is used. The document review is the primary technique used in this phase to gather and organize past research on the subject.

2.1.1 Data Collection

The data from prior studies was gathered in the first step of the thesis research technique in order to compare different JavaScript frameworks. On the most popular online academic databases, such as IEEE, ScienceDirect, and Scopus, there is a number of published research studies on JavaScript frameworks' comparison available. The majority of comparative studies, though, may be found on internet blogs, developer support websites, and JS framework online documentation. As a result, the data collection procedure encompasses all relevant documents, including developers' personal blogs, on the subject. The the material in question is fragmented and dispersed throughout the various sources, according to preliminary data analysis. Each source compares the different frameworks in different ways. For example, a source might include qualitative data regarding React and Vue.js frameworks for some of the critical metrics covered in this paper, but not for others.

2.1.2 Data Analysis

Secondly, the data from stage one were analyzed to gain understanding and formulate a view on the subject through inductive reasoning on observations and patterns.
The report then proceeded with organizing the aspects deducted from the second stage, in the two categories defined from objective 1.
The - now organized - data were filtered through the analytic hierarchy process (AHP) [7], developed by Saaty, which was used to help analyse the importance of the aspects' criteria.

The now weighted aspects were pair-wise compared while AHP was again utilized and the process was repeated in an iterative manner until certain minimums as well the defined scope-limitations of the thesis were met.

At this point the objective 1 was met and objective two was thus ready to be approached.

2.1.3Implementation Strategy

For objective two to be fulfilled, the next stage of the project required case study approach. Further research and an empirical inquiry on the chosen aspects were the means applied to reach objective two.

Chapter 3 Theory

This chapter introduces general contextual concepts and integral technologies of the Web development and JS ecosystem.

3.1 Web Application

A web application is a computer program that runs on a web server and accesed through the browser. When opposed to traditional apps, the idea behind a web application is that developers just need to create one client rather than one for each operating system. Because all that is necessary to operate a web application is a web browser, they are relatively simple to use. Browser-supported languages like JavaScript and HTML are frequently used to create web apps.

3.1.1 HyperText Markup Language & Cascading Style Sheets (HTML & CSS)

HyperText Markup Language (HTML) is used to define the content of the web page and the way it is organized while Cascading Style Sheets (CSS) is used to determine how the page is presented. Finally JS is used to specify and enable the actions on the content of the page [8]. Web browsers convert HTML and CSS-encoded pages into a readable 'document' for the user [8]. Modern browsers come with a JavaScript interpreter built in. This is not exclusive to PCs and laptops, since this technology has spread to a variety of other devices, including game consoles, tablets, and smartphones [9].

3.2 JavaScript

JavaScript (JS) is a high-level, lightweight, interpreted/just-in-time compiled programming language used to add dynamic behavior to web pages/applications, thus often described as “the language of the web”. Whereas HTML and CSS provide structure and aesthetic to web pages, JavaScript adds interactive components the user can engage with. As a text-based programming language that allows you to construct interactive web pages on both the client and server sides. Although foremost used as a client-side scripting language, there are numerous use cases of JS engagement in server-side envornments and applications such as Node.js, Deno, Apache, MongoDB, CouchDB, Adobe Acrobat, GNOME Shell etc [7]. Given its widespread use across most modern websites and all modern web browsers [4], this programming language has indisputable popularity. Modern web browsers have an inbuilt interpreter for JavaScript codes that can parse and execute the language [9], which is one of the language's most important characteristics. JavaScript, for example, allows and implements adding new HTML to the web page, editting of the content, and modifying the styles, while responding to user input, such as mouse clicks, pointer movements, and keystrokes. [10].

JavaSript's object oriented nature, the utilization of first-class functions, dynamic typing and the support for multiple programming paradigms, give it a lot of versatility [11]

Considering` the overall characteristics and its potential, the popularity of JS among front-end developers comes as no surpise.

3.2.1 JavaScript XML (JSX)

JSX is a syntax not unlike XML/HTML, but extending ECMAScript to allow shifting the paradigm to integrate HTML into JS instead of the opposite. The notion is that the syntax is converted into standard JavaScript objects that the JavaScript engine can understand. This is useful since it allows you to write HTML/XML in the same file as JavaScript, allowing you to take advantage of the JavaScript features. [12]. It was created by the React team for usage in React apps, although it can also be used in other frameworks, such as Vue [13].

3.2.2 JavaScript Frameworks

JavaScript interacts with the browser through a complicated event-driven mechanism. Due to JS's considerable flexibility [11], as noted in ch. 3.2, writing maintainable code is particulary difficult.

Because JavaScript is so frequently used for developing interactive online applications and because of its popularity, a variety of plug-ins, frameworks, and libraries are created to make the development proccess easier and more effective. A software framework is a set of libraries that utilize their own architectural and design concepts. Hundreds of JavaScript frameworks have been created in recent years to aid front-end developers in swiftly developing applications [3]. The majority of the frameworks are open source, with a number of them having distinguished themselves by becoming a critical component of multiple projects as a result of their features and integration with excellent tools [14].

3.2.3 Framework versus Library

Frameworks and libraries are both pieces of reusable code authored by other programmers that can be utilized to solve common problems and aimplify and abstract basic functionalities.The terminology used to distinguish frameworks from libraries is a little hazy: In online debates and articles, both “library” and “framework” are interchangeably used when reffering to React. Defining the two terms and using them distinctively, would be beneficial for coherency and clarity in the field.

A library is a collection of narrowly-scoped resources that perform a series of well specified operations, through predefined functions and classes. This standard is followed by JavaScript libraries as well. For example, jQuery, which performs DOM manipulation, event handling, CSS animation and AJAX capabilities, serves as a proper representative. On the other hand libraries that are particularly focused to the execution of a certain specific activity, such as JSLint or Mocha, can be categorized as tools and are used to perform a single task like syntax checking, testing etc.

A framework is essentially a collection of libraries, which is organized and executed in a specific manner. They provide a skeleton for developing an application, enforcing or simply suggesting – depending on whether is is “opinionated” or “unopinionated” - certain best practices. The main technical difference bertween the two is in regard to the control flow. Simply put, when using a Library, the developer is in control of the application flow i.e the code “calls” the library. In case of a framework the opposite is the case, justifying the term “Inversion of Control”. The above apply as such to JSfs as well. A framework provides a set of guidelines for the developer to construct the entire web app, whereas a library has no similar overarching control, but instead focuses on acting as a platform where simpler domain-specific operations must be plugged in, or just acting as such specific-standalone tooling [15].

3.3 Template Syntax

A lot of modern JavaScript frameworks utilize something known as the template syntax. The template syntax is a means of defining the dynamic part of our HTML structure. The template will keep track of the component's local state by binding the data in the state to the UI that it represents [16].

3.4 Document Object Model

The DOM is a W3C (World Wide Web Consortium) standard, for accessing HTML and XML documents. It serves as the data representation of the objects that constitute a web document such as a web page. At the same time it is a Web API that provides the ability to dynamically manipulate the structure, style, and content of a page. Essentially it acts as a representation of the document's logical structure in terms of nodes and objects allowing JS to access and alter the appearance and behavior of a website. This is the reason that the DOM is so vital for JavaScript frameworks. In fact the DOM is language-agnostic, as it can be called as a script (although web developers usually use it through JS). It is essentially a conversion of the structure and content of an HTML document into an object model that can then be utilized by the program.

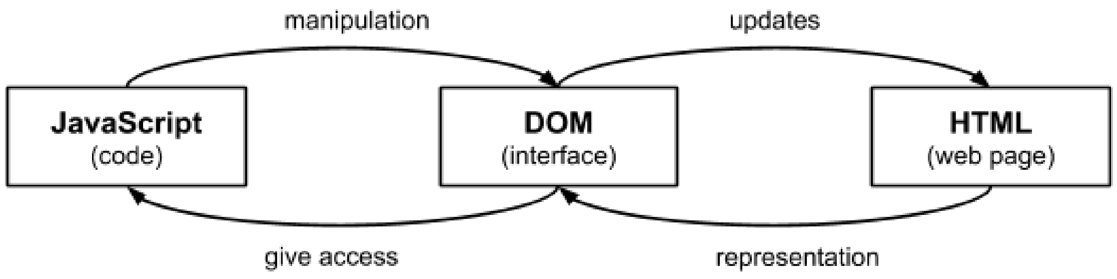


Figure 1. The correlation beween JavaScript, the DOM interface, and the HTML document.

3.5 Model View Controller Pattern (MVC)

The Model-View-Controller pattern (shown in Figure 2) is a generic software design paradigm that applies to the majority of web applications.

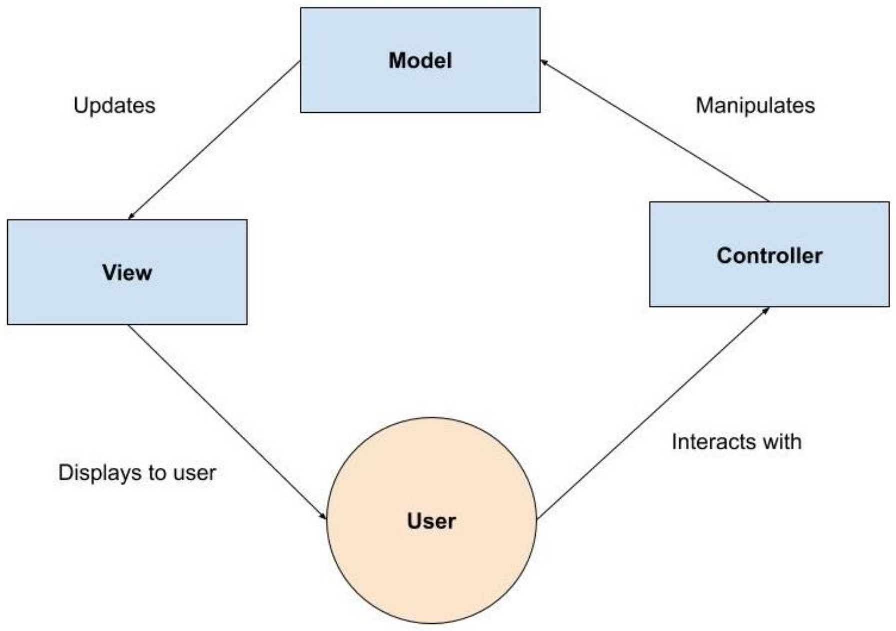


Figure 2. A basic illustration of the MVC structure.

This pattern is a representation of how most web apps handle internal interactions and how users interact with them. The pattern is made up of three parts and the interactions between them: the model, which manages the data and logic according to the controller input, the view, which constitutes of any graphical and informational interpretation of the model that is presented to the end-user, and the controller, which translates the user input to commands for the view or the model thus interacting with the data and altering the view that the user sees. The view ("V") section of the MVC pattern contains the majority of the essential functionalities and features of the frameworks explored in this study. The controller, which defines interactions with the model, might be regarded as dynamic JavaScript functionality allowing for user-model interaction. Although originally created for desktop applications, the MVC paradigm is now widely utilized in web applications. [17].

3.6 Single Page Application (SPA)

In contrast to a Multi-Page Application (MPA) - where the server must send back the entire page at every user request, which must then be reloaded at the web browser- A Single Page Application (SPA) is a web application that consists of discrete components that are loaded in memory on the initial page visit and updated autonomously, avoiding the need for the full page to refresh after each user activity. It makes the application more dynamic with real-time updates, thus providing a better User Experience (UX) which resembles that of native applications- with fast page transitions, without the full- web-app-refreshing interrupting the usage flow, resulting in rich, high performance User Interfaces (UIs). Because the components may be reused, this type of development allows for less server-side code, resulting in a significant reduction in the amount of code required. The SPA was first created and patented in 2002, notably targeting JavaScript as an intended language for its development [18].

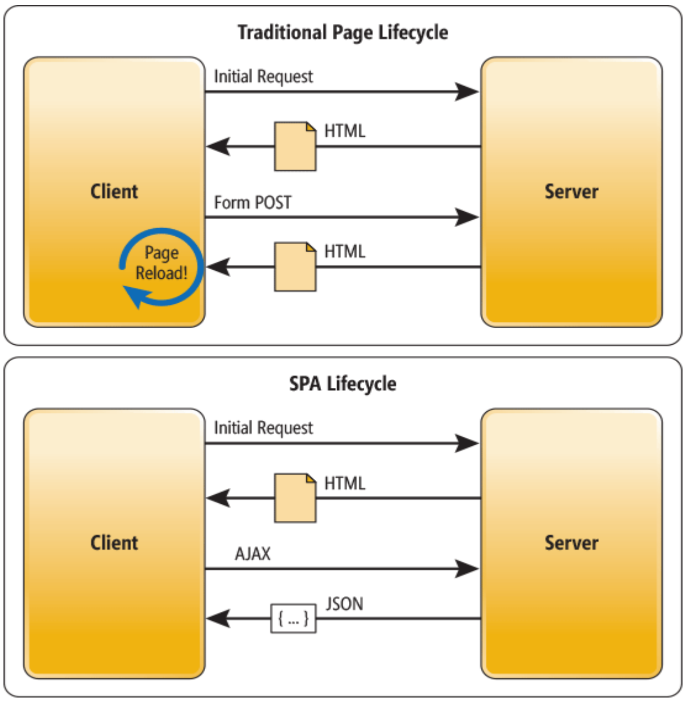


Figure 3. SPA – MPA lifecycle comparison Source: Adapted from [84]

3.7 Asynchronous JavaScript And XML[1] (AJAX)

AJAX is a sum of technologies bundled together to take advantage of the introduction of asynchronous items on the web, rather than a single technology or tool. AJAX uses the XMLHttpRequest (XHR) object to communicate with servers. Because of its "asynchronous" nature, data in a variety of formats, such as JSON, HTML, XML, and text files, may be sent

[1] Extensible Markup Language

and received, exchanging information, and updating the content without reloading the page.
The success of SPAs, which would later become widespread, was primarily due to the earlier AJAX’s breakthroughs in client-server communication.

3.8 HyperText Transfer Protocol (HTTP)

The HTTP is an an application-layer, client-server protocol designed for the exchange of data on the Web. In a typical HTTP communication, a client machine makes a request to a named host, which is located on a server, which then sends a response message including any requested data. A set of methods (GET, POST etc.) are defined and utilized to specify the exact action to be executed for a specific resource, by the queried server. HTTP is transmitted over TCP/TLS over TCP link, though it could potentially be transmitted over any trustworthy transport protocol [19]. It is essentially the foundation of the Web, since it is used to load web pages using hypertext links, as well as for fetching or storing any additional resources.

3.9 Application Programming Interface (API)

API is the abbreviation for Application Programming Interface. A Web API, is an API over the Web which is accessible through the HTTP protocol. It acts as a framework which aids developers in the creation and development of HTTP-based RESTFUL services that communicate with client entities such as browsers. The function of an API is to provide data to the programmers who subsequently make it accessible to the end-user. Any text format such as JSON and XML is supported. A server-side web API is a programmatic interface that exposes one or more endpoints which can be publicly accessed. It specifies a communication system for the client and the server to exchange messages.

3.10 Back-End

A Back-End (backend) is required to serve a single-page application and give a database connection in order to create a fully developed application. In this section, we will introduce a server environment called Node.js, Express which is a JavaScript back-end framework and MongoDB which is a document database. When all three are used together to form the back-end environment it is commonly referred to as the MEVN stack, as results from thein initials.

3.10.1 Node.js

Node.js is a comprised of Google Chrome's V8 JavaScript engine, a platform abstraction library (*libuv*), and a core library written – for the most part - in JS. It is a cross-platform, open-source runtime environment for creating scalable JavaScript back-end and network applications. Node.js is lightweight and efficient thanks to its event-driven, non-blocking input/output (IO) mechanism, which makes it ideal for I/O-intensive - as well as - real-time applications running across multiple devices [20]. Node.js represents a paradigm shift as it allows web applications to be developed with the same programming language in both the client as well as the server side.

3.10.2 MongoDB

In contrast to relational databases (e.g MySQL), MongoDB is a *NoSQL* (non-relational) document-oriented database. This means it stores data in JSON-like documents in a non-tabular manner. MongoDB has a powerful and expressive query language that lets you filter and order documents by any field, while the queries are also in JSON format, so they're easy to compose. MongoDB has full ACID (Atomicity – Consistency – Isolation - Durability) attributes, which are a set of key properties that define DB transactions and ensure they are handled reliably [21].

Chapter 4 JavaScript Frameworks

This chapter contains a justification for the choice of frameworks, React.js (React) and Vue.js (Vue), as well as a description of each framework used in the evaluation.

4.1 Selection of Frameworks

Essentially, to an extent, the same factors that apply to choosing a framework to use, also apply to also choosing one for evaluation.

"Demand, usability, community, and reputation" are the four factors that influence framework selection, according to Duvander and Romhagen [22]. The demand is linked to the skills required for frameworks, how demand is influenced by geography, and the impact project requirements have on the framework selection process. In this study, usability refers to the aspects of the development process that help and improve it, or that hinder and make it difficult, such as documentation and structure, as well as the time it takes to start a project with a framework and the total time it takes to complete a task with a framework. Community of a framework/language is of utmost importance in programming and especially in web development. When there is an active community, the ability to look for solutions on online communities like Stack Overflow is improved. The maintenance of the code is easier when there is a large user community since it is easier to find developers who are familiar with a specific framework. Finally, reputation concerns the view held by developers as well as related professionals for a specific framework and how this view is formulated. The company "behind" a framework can have a role here, as well, as major and well-known companies like Google or Facebook can provide a sense of trustworthiness.

The above regards where taken into consideration for selecting two –out of the plethora of JSfs- to be evaluated and compared.

Of course this implies that more/all JSfs have been initially evaluated –which is not the case. None the less, the author's previous time with the subject, as well as minimal developer experience with one of the selected frameworks, plus the study and literature review conducted for this thesis, allow for a safe assumption that the essentials proposed from the referred studies and experience are met by the selected frameworks, which is of course already self-evident by their massive adoption at professional- production level. After this theoretical basis was met it was a matter of certain characteristics, nuances and correlations between them, as the four themes mentioned above, that made them the author's final choice for evaluation.

4.2 Description of the Selected Frameworks

In the following subsections the two selected frameworks are presented.

4.2.1 Vue.js

Vue is the newest of the two frameworks in question. It is a, progressive framework designed to be incrementally adoptable for integration with existing projects or combining libraries and tooling to enable complex user interfaces (UIs) and Single-Page Applications (SPA) according to its developer [23]. Progressive means that it is simple to use, versatile, and fast, and that it can be used as a library or a framework depending on the requirements.

Vue, like React, makes use of virtual DOM, and some of its capabilities have been refined even further in Vue.

How Vue.js "works"

Vue.js comes with a reactivity system [24]. We must first build a Vue instance, and generate certain attributes when Vue is instantiated. Vue provides getters and setters for data properties (particular properties of the Vue object) when you declare them. These are methods comprised by special keywords (get/set) in correlation with a function, that operate on a JS object, and have the goal of dynamically returning a computed value and dynamically detecting changes in a given property.

Using Node.js, Vue.js can perform Server Side Rendering (SSR) and although Vue.js does not natively support mobile app development, offers a variety of options for creating native iOS and Android apps with Vue.js [25].

Evan You first launched Vue in February 2014, and it is still developed by him and a team of core members [26].

For the following evaluation Vue v2.5.22 is used.

4.2.2 React

React is an open-source JavaScript library, used for constucting interactive user interfaces. As stated in section 3.2.3, the term "library" denotes that it is intended to provide a platform for the developer to build on rather than to cover all aspects of the program. It is declarative, meaning it creates, updates, and renders unique views for every state [27]. React includes state management and DOM rendering by default, however it does not contain routing or some other client-side functionalities.

Because React does not employ regular HTML templates, there is no corresponding HTML-template file present. JavaScript is used to define the entire HTML document code. The use of a virtual DOM is also introduced in React. Because of faster processing, data changes are made first in the virtual DOM before the real DOM. In the following chapter, we'll go into virtual DOM in further depth.

By utilizing Node.js , React.js can implement SSR, as well as support mobile applications' development with *React Native*.

How React "works"

Each component in a React application is in charge of rendering a small, reusable chunk of HTML. Complex applications can be developed using simple building blocks of nested components. A component is also able to record its own internal state - for example, a *TabList* component can keep track of the currently open tab by storing a variable [28].

React uses a virtual DOM, which is essentially a JavaScript representation of a DOM tree. As a result, instead of reading or writing directly to the DOM, it will use its virtual representation, which will then attempt to update the browser's DOM in the most efficient manner possible. React elements, in contrast to browser DOM elements, are simple objects that are "easy" to generate. Finally, the changed objecsts of the virtual DOM will be accordingly updated at the real DOM.

The justification of implementing a virtual DOM, is that the DOM-tree's manipulation is faster this way, due to JavaScript's inherent speed.

React is maintained by Facebook and its developer- and business -community. Jordan Walke published it for the first time in March of 2013. React is used on *Facebook* itself, as well as *Instagram* and *WhatsApp* [29].

For the following evaluation, React v16.6.1 is used.

Chapter 5 Selected JavaScript Frameworks Comparison Analysis & Evaluation

5.1 Selection of Comparison & Evaluation Criteria

Pano et al. [5] used a qualitative interpretative study with 18 users to develop a model of ideal JSf aspects such as usability, cost, efficiency, and functionality. Although the research is not concerned with the comparison of JS frameworks, it does examine the primary considerations developers have when selecting one. Developers, for example, choose frameworks that require less lines of code for specific purposes, favor modularity and readability of code. Automated event handling and DOM manipulation seem to be important, as is clear documentation, and browser support. A JavaScript framework has a large variety of "duties", thus there are a lot of things it has to deal with, which makes it difficult to conduct a thorough and precise analysis. In this assessment, the frameworks are assessed based on a core set of functions that they provide. i.e. their feasibility to perform basic operations and cover common functionalities, as they are empirically defined and documented from the plethora of diverse functions and numerous usage cases in SPAs or MPAs.

The functions that will be evaluated during the code evaluation and the development period are divided into two categories:

- Explicit programming-related functionalities, such as: DOM Manipulation, various Selectors, basic form elements, functions for event handling, Ajax capabilities, routing browser compatibility etc. In further detail, the above are translated in the following functionalities and features of a JS which are then presented, discussed and examined practically-programatically as:
  Components, Data binding and State Management, Routing, Scripting and Rendering, in chapter 5 as an overall introduction to the main –compared- features of the frameworks, while they are further elaborated upon and examined at greater detail in the applications implementation in chapter 6.
- Qualitative metrics not directly connected to the programing efficiency but explicit to the successful adoption of a framework, including: popularity, ecosystem, developer experience etc. as discussed in chapter 5.

5.2 Comparison

In this section the main objective of the thesis begins, which is to present and elaborate on the comparison points and metrics regarding the two frameworks' programing functionalities and overall qualities as they are defined in the previous sections.

5.2.1 Components

The developer interacts with the framework components the majority of the time during the application development process. As a result, a review of component syntax, functionality, and usability is justified.

React offers two – most prevalent -distinct types of Components: Function components and Class components.

The first approach, shown in Listing 5.1, resembles the implementation of a native JavaScript function. The following basic component calculates the sum of two added numbers.

```
function SumOfNumbers (props) {
     return <h1> The sum is {props.num1 + props.num2}</h1>;
}
```

Listing 5.1: *SumOfNumbers - native JavaScript* - component in React

The second approach, involves simple ECMAScript 6 (ES6) classes, which in this case are all child classes of React's Component class, as in Listing 5.2 below.

```
class SumOfNumbers extends React.Component {
     render()  {
          return <h1>   The   sum   is   {this.props.num1   +
this.props.num2}</h1>;
                    }
}
```

Listing 5.2 *SumOfNumbers* - ES6 class - component in React

React allows components to be rendered directly. This is accomplished by passing an object to *ReactDOM's* render method, which includes the name and parameters of the component [30]**.** The render function of *ReactDOM* is demonstrated in Listing 5.3 below.

```
function SumOfNumbers(props) {
     return <h1>The sum is {props.num1 + props.num2}</h1>;
}
ReactDOM.render(
     <SumOfNumbers num1={2} num2={3} />,
document.getElementById ('root'));
```

Listing 5.3 The JSX for the SumOfNumbers component rendering

Listing 5.3 shows JSX syntax for generating an element. JSX appears to be a template language, but it is actually a syntax extension to JavaScript [31], and it has all of its capabilities. React uses it instead of HTML or template literals, because it optimizes compiling, is statically-typed and –for the most part- type-safe. JSX is essentially syntactic sugar for constructing items. In a React application, however, using JSX syntax is not obligatory but it does make developing a React application easier [31]. Components in React take input in the form of properties, which are called props. Using properties is also demonstrated in Listing 5.3 above.

Components have a specific lifecycle in React. The lifecycle has methods, which will run before or after an event: *constructor*(), static *getDerivedStateFromProps*(), render(), and *componentDidMount*() are called when an instance of a component is being created and inserted into the DOM (Mounting) while, static *getDerivedStateFromProps*(), *shouldComponentUpdate*(), render(), *getSnapshotBeforeUpdate*(), *componentDidUpdate*() are called in this particular order when changed props or state are causing an update. Method *componentWillMount*() "is considered legacy and developers are prompted to avoid it in new code “ [32].

The syntax of Vue's Single File Components is simple and straightforward. Vue components, like React components, have their own separated scope with properties. A Vue component is registered with a message property and utilized in the template In Listing 5.4.

```
// Define a new component called todo-item
Vue.component('todo-item', {
       template: '<li>This is a todo</li>'
})
var app = new Vue(...)
```

Listing 5.4 Component registration in Vue

As shown in Listing 5.4, registering a Vue component [33] is simple, although the syntax differs significantly from that of React.

Lifecycle hooks can likewise be used in Vue to access different stages of processing, and are tied to the Vue instance itself, not to components, unlike React. Vue's Lifecycle hooks are: *beforeCreate*, created, *beforeMount*, mounted, *beforeUpdate*, updated, *beforeDestroy* and *destroyed,* [34] the latter two of which are deprecated at v.3.0.0 (Vue3) and later versions, and are now instead called: *activated* and *deactivated* [35].
To summarize, React's component style is the most similar to native JavaScript, and so it is considered the easiest syntax to learn for previous JavaScript developers. On the other hand, Vue's components offer a bit more functionalities. In terms of the fundamental applications, there are no significant challenges in any of the framework's component implementations. The most efficient approach to use components is mostly determined by the developer's prior knowledge and habits; there is no particular benefit to choose one framework over another based on component implementation.

5.2.2 Data-binding and State Management

There could be a significant difference in terms of performance, as well as how communication between a component and a view can and should be implemented. This also applies to data binding in components. This section analyzes and contrasts the differences between the two frameworks' data binding and state management options.
The data in React is supposed to flow downwards, which is termed one-way data flow. Data from any parent component can be passed to a child component, but not vice versa[2]. As a result, the page's input fields have no direct access to the component's state. In other

[2] React's two way data binding solution is officially available at the latest version

words, HTML is unable to modify the component. This paradigm offers advantages in terms of simplicity, but most applications require the ability to change the state of their parent component at some point. In this case, inverted data flow is required. For example moving data from an HTML input field to a component is considered inverse data flow. It's possible with callback functions in React, as well as attaching an event handler to DOM events and retrieving the value of the HTML input field from the event object [36]. Moreover, React officially supports a solution for two way data binding in the latest -at the time of writing- version (v17.0.2) [37].

In below Listing 5.5, the value of the input field is first set by the component's state, and then the data must be moved in the opposite way using a method.

```
render() {
     return <input value={this.state.value}
            onChange()={this.handleChange} />
}
```

Listing 5.5 React input field with *onChange*() event

The data is moved in the opposite direction in Listing 5.5, and the *onChange*() event handler is set to the *handleChange()* method. The event object is passed as a parameter to the *handleChange()* function. When the value of an input field is updated, the state of the component changes in accordance with the event's target value.

Vue on the other hand officially provides a two-way data binding option since its first release. Two-way data binding allows you to modify the state of a component as a result of changing the view, for example, when the value of an input field changes, the component's addressed property changes as well. Event handlers aren't required in this scenario; they do exist, but they're managed by the framework and abstracted away from the developer. Vue's way of two-way binding is shown in Listing 5.6.

```
<input v-model="name" >
```

Listing 5.6. Two-way data binding in Vue

Vue's two-way data binding results in more clean and straightforward code compared to React. However, while it has the advantage of a simpler implementation, it is less manageable and performant. The framework must wire up watchers for each element in order to support two-way data binding. These watchers are used to see if the value of the input element has changed. Watchers require resources from the browser, and as their number increases, in some cases, keeping track of all the many models and views that use the same data may be problematic. The issue is depicted in Figure 4 below.

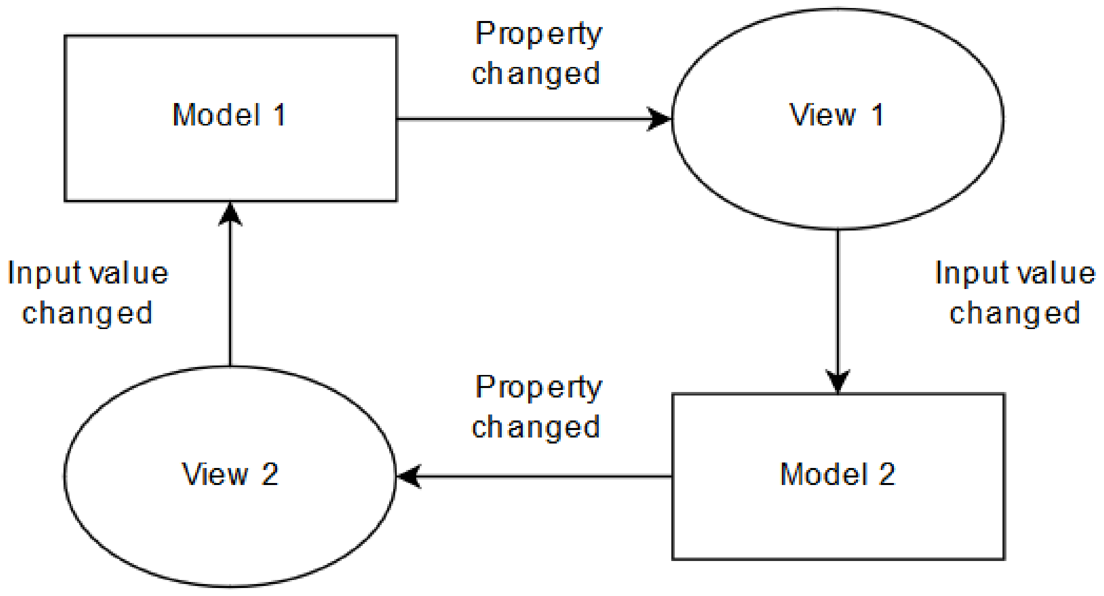


Figure 4. The two-way data binding Infinite loop

As seen in Figure 4, each View makes use of its corresponding Model's property. When the Property is changed it also affects the input value for the next Model, thus altering its state. This leads to further changes in the corresponding property, and thus the next View, leading to an input value change which in turn alters the state of the next model. The result is that a "cascading effect" is triggered which leads to a loop that continues indefinitely if it keeps going in the same direction.

As it can be understood, both ways of data binding have their respective advantages and disadvantages. Because of the dependencies between the models and the views , two-way data binding can lead to unmanageable situations, so it should best be used for more trivial operations. On the other hand, on one-way data flow, more custom code is required but its exactly because the event handlers are set by hand, that a possible error can be easier tracked down to the action that caused it, and the overall unidirectional data flow’s behavior is easier to predict [38].

5.2.3 Routing

Because entry-level routing capabilities are a required component in most applications, the framework's routing options should be investigated. Each framework’s ability and methods of implemention of routing functionality are described and assessed in this section.
Router or comparable functionality is not included in React. React requires a router framework for this, and React Router is likely the most popular alternative. Since the used version, React Router also supports dynamic routing, which enables responsive routes, allowing the content of a website to be instantly updated according to the device’s.
Vue includes an official router in its base installation, whereas React requires an additional library. Dynamic routing, nested routes, navigation routes / guards and route parameters are among Vue’s router features [39] .

5.2.4S Scripting and Rendering

The virtual DOM is React’s approach of constructing the DOM. The term ”virtual” refers to the UI representation that is synced with the “real” DOM, reducing the resources that the browser needs to use. In terms of scripting performance, this provides numerous advantages as it introduces better performance and cross-browser compatibility.
In practice, virtual DOM allows React to detect any state and/or prop changes. All of the modifications that need to be performed are done first in the virtual version. When the content of the element is no longer changing and the difference between the original and altered content is finalized, the updates are batched and sent to the browser (“real”) DOM. By using the virtual DOM, less memory usage is required and the DOM actions are performed significantly faster [40].
Vue's runtime efficiency has also been much improved in this regard. Vue, like React, features a virtual DOM implementation, in an even more optimized way. While React re-renders the entire component sub-tree when the component state changes, in Vue the dependencies are tracked automatically, which means that the re-rendering takes place

only at nodes that are required. The above removes the need for the developers to perform the necessary optimizations themselves [41].

Virtual DOM is used by both React and Vue, with some considerations emerging from a detailed examination of the framework's scripting and rendering capabilities. It seems though that despite the minor differences in implementation strategy, which give Vue a slight boost performance-wise, the differences are significant only when rendering extremely large data structures. As a result, performance discrepancies should not be highlighted in the overall assessment.

5.2.5 Platform Support

For each framework, there are browsers that are completely supported, browsers that are deprecated or not at all supported, and other ones that require the use of polyfill scripts to function properly. Polyfill scripts, often known as *polyfills*, are short compatibility packages that incorporate missing JavaScript functionality for a particular browser. In most cases, this entails providing the –legacy- browser with the code needed to run newer features. The various *polyfills* may perform significantly differently in practice. Support may vary considerably across browser versions.

All major modern browsers are supported in both frameworks by default. On the other hand Internet Explorer 8 and earlier versions are discontinued while support for Internet Explorer 9, 10, and 11 requires *polyfills*. As a result, both frameworks are fairly equal in this scenario. React's requirement for external libraries, on the other hand, must be considered. External libraries are frequently required by React for various functionality, each of which has its own set of browser requirements, therefore the necessity for *polyfills* may change.

5.2.6 External Libraries & Modules

External libraries add functionality to a library or framework that is not available by default. The above includes the addition of developer tools, enhancement of the UX, and also that the performance. Since the User Experience (UX) is such a large aspect of an app, employing an external library can provide more specific and complete implementations for a variety of certain aspects. High-performant graphics for example, also offering wide compatibility, and scalability, can be difficult to create. It is the exclusive focus of certain libraries. This section gives a quick rundown of the frameworks' library requirements and the current supply. Routing libraries, user-experience, user interface libraries, and even libraries for creating new libraries are all available in React. In this regard, the React community is very active, and there is constantly team of programmers that work on a certain feature. These alternatives often provide better optimization than Vue's native approach, which tries to cover all functionalities by default.

Vue offers a large number of external libraries to choose from: ranging *from* UI Components and Utilities, Frameworks for mobile/SSR/Static generators and more, to all kinds of Utilities for state management, HTTP Requests, Typescript, animations, localization, styling, custom events, payments, and development tools etc [41]**.**

Vue, as a framework, seeks to accommodate a wider range of applications, ideally acting totally independently. React on the other hand is referred to as a library, since it does not seek to offer all functionalities, instead serving as a foundation to build upon. A React application will inevitably be complemented with the use of external libraries, whereas Vue can function idependantly at least for basic functionalities as the ones described so far.

5.2.7 Native Applications

One important feature that React has that Vue is lacking, is a robust system for native rendering, which allows the developer to create mobile apps using the same React component paradigm as web apps. Vue does have a means to leverage their component syntax in natively rendered projects, and while they offer numerous alternatives such as *Native Script*, *Vue Native*, and *Weex*, they have not been as thoroughly vetted and hence lag behind React Native. For developing native apps, React is clearly superior.

5.2.8 Localization

In many circumstances, the ability to modify the application's language is required, hence the user experience of the application is also heavily influenced by localization.

React doesn't have any built-in localization functionality, so you'll have to rely on third-party libraries. A basic react localization library [42] is one of the most popular options. Each string is defined for each language independently in react-localization, and API methods are used to alter or change the current language. After that, you can utilize the formatted strings in a component.

Vue depends on third party solutions for localization such as Vuex-i18n and vue-localize, as there is no official solution provided.

5.2.9 Documentation & Community Support

The framework's documentation is a developer's primary source of information, therefore it should be taken into account while assessing the framework's practicality, as well as the size and activity of its community. For newer frameworks in particular, community support may be lacking or fragmented for specific parts, making documentation an even more important consideration. Inadequate documentation can stifle or even stop application development.

The documentation for React is extensive and covers all of the framework's common features [43]**.** It follows a predictable pattern that is simple to read and comprehend. There's also a separate tutorial part [44] with step-by-step instructions that the developer can implement directly in the browser or by creating a project in an editor. The lesson begins with the installation of React and concludes with a fully functional game. It also includes theory parts that have been well-adapted.

Among all JavaScript frameworks, React has one of the most active communities. Users may keep up with the latest news and join discussions on the topic in social media, as well as in React’s own discussion boards and chat.

The documentation for Vue is well-organized, with unified theory and tutorials that provide correct examples for each area as the reader progresses. Writing is done mainly from a theoretical standpoint, but it also includes practical assignments and recommendations that the developer can test localy and stay relevant with their learning. Vue template syntax, instances & event handling, watchers & computed properties, classes, reusability & composition, properties, styles, form bindings, conditional & list rendering, components-basics & in-depth, are all covered in the documentation, as is routing and state management, so it is fair to say that all the essentials as well as more complex features are presented. There's also a plethora of advice for developers on testing, deployment and migration from prior versions. Vue also has a section in its docs where it compares to different frameworks including React [45].

The Vue ecosystem consists of a booming community with officialy-maintained resources, including as a forum, real-time chat, and GitHub repositories, to connect, raise and resolve issues, answer questions and contribute in different manners.
Popularity is also frequently linked to community support. Section 5.2.11 goes over popularity in more depth.

5.2.10 Developer Experience - Learning Curve

The field of JS frameworks is relatively new. At some point a developer who is programming in JavaScript is likely to have some prior experience with the language, but not with frameworks. Learning a new programming technology takes time, however the learning curve varies depending on the particular framework.
In addition to the overall review, the developer experience, which includes installation, error messages, and compilation time is also an important factor.
According to Vue's own development team, it has the shortest learning curve. In comparison to Vue, React is not the most user-friendly solution in terms of learning curve. The use of JSX may have the most impact on the curve. To start with Vue, only experience with HTML and ES5 JavaScript will be required, with the syntax being rather simple and straightforward. Vue's default linter is *ESLint* is a linter tool that is responsible for analyzing code, discovering mistakes, and reporting them. The *ESLint* rules in Vue's default configuration are somewhat strict. Semicolons and unused variables, for example, are not permitted. This may be off-putting to certain developers. The configuration, on the other hand, can be altered.
The Node.js *NPM* package manager can be used to install each framework. Both frameworks are also accessible in other package managers - Yarn being the most common alternative - and they can be installed with ease by running a single command which downloads and installs the necessary dependencies for the particular module. For scaffolding new projects or components, both have their own Command Line Interface (CLI). A framework's compile time is another factor that influences the developer experience. Regarding this comparison, both frameworks are commendable in this regard. Sample applications build in 150-400 milliseconds for React, with Vue being considerably faster at 60-150 milliseconds. Also both React and Vue offer dedicated developer tools for application overview and debugging**:** One thing that separates and gives Vue an advantage

in developer experience is the Vue GUI, which provides the user with an optical representation of the vue-cli, allowing a more clear and intuitive overview than the CLI environment, thus enhancing ease of use and making it user-friendly especially for the new developer.

5.2.11 Popularity and future

How popular it is and what sort of prospects it has are major considerations in the practicality of JavaScript framework for organizations' main development direction, as well as for a developer's educated and intentional choice. Choosing a dying framework may be disastrous for a corporation and certainly not ideal for a developer, and it would almost certainly result in a significant loss of money and/or time in both circumstances, overtly and implicitly. The popularity of an application has a significant impact on community support and, as a result, on the speed with which it is developed. The goal of this section is to look at the frameworks' present popularity and determine how likely it is that the development process will continue in each scenario.

There are various services that gather measurenments and provide stats about JSfs popularity. *Hotframeworks* website [9] is utilized in this section, as a source that provides such data. *HotFrameworks* ranks frameworks according to their received *GitHub* stars and the number of queries considering a certain framework on Stack Overflow. Results are standardized to a 0-100 scale [46].

Based on *HotFrameworks'* scores, Table 1 demonstrates the popularity of React and Vue. [57].

Table 1

Framework	**Github Score**	**Stack Overflow Score**	**Overall Score**
React	**99**	**97**	**98**
Vue.js	**100**	**87**	**93**

React now has the largest share of popularity, as evidenced by *GitHub* stars and Stack Overflow queries combined, despite Vue scoring higher on *Github*.

React Usage Statistics 2020

- For the fourth year in a row, React was voted the best front-end JavaScript framework in the State Of JS 2019 worldwide survey of JavaScript developers. Since 2016, React has been voted the best JavaScript framework, beating out prominent competitors Vue and Angular.
- According to *the State Of JS* 2019 poll, 71.7 percent of JavaScript developers are presently using React, with another 12 percent expressing an interest in learning it in the future. This represents a significant increase of approximately 8% above the previous year's figure of 64% active users.
- With 35.9% of votes in Stack Overflow's developer poll 2020, React was rated the second most popular web framework, trailing only jQuery, which has been losing ground to React.
- As of June 2020, roughly 1.45 - 1.6 million websites - live and historical - were developed with React, according to *Wappalyzer* and *BuiltWith* data.
- At the same time it had 150k stars and 1388 contributors at Github, and was almost reaching its record high of 7.5-8.5 million downloads per week, according to *NPM* Statistics.

Vue Usage Statistics 2020

- For the second year in a row, Vue was voted the best Front-End JSf in the *State Of JS 2019* developer survey, beating out Angular. In 2017, Vue was ranked third, and in 2016, it was placed fifth.
- According to the State Of JS 2019 poll, 40.5% of JavaScript developers are presently developing with Vue and plan to continue so, while 34.5% have expressed strong interest of using it in the future (second most rated behind

Svelte). This represents a nearly 12-percentage-point increase in the previous year's 28.8% of the total users.

- Vue was named 7th most popular all around web framework and 3rd most popular front-end JSf following React and Angular according to Stack Overflow's developer survey 2020.
- As of June 2020, around 427k-693k websites - live and historical - were developed with Vue, according to *Wappalyzer* and *BuiltWith* data.
- At the same time Vue had an astounding 1.63 million weekly downloads on NPM, edging out Angular, as seen in NPM statistics and 166k stars and 293 contributors at Github.

As it is made obvious from the State of JS 2019 survey [47], as well as from number of *npm* downloads, React was- at the time- by far the most popular of the two frameworks, with Vue rising in popularity at a much greater rate. Interestingly Vue had considerably more *Github* stars, which can be attributed to its dedicated community of developers [48].

5.3 Conclusion

As stated at the start of the chapter, some characteristics are more important in evaluating frameworks, whereas others, have a smaller effect on the total rating. The following are important factors to consider while developing a web application:

1. The prerequisites must be met. The framework must be able to implement the following features as a minimum: data transfer to and from the server, routing, and data grid (standalone or from additional libraries) capabilities.
2. The framework's development has to continue. The ongoing maintainance and further enrichment framework's cannot be halted, as this will have severe effects to the framework's users.
3. High efficiency. The development of a specific task must be completed quickly and effectively, taking into account the framework's ready-to-use features, learning curve, documentation, and support from the community.

The comparison of the two JSfs regarding the evaluation criteria defined in 5.1 does not signify any –considerably important- discrepancies, which indicates that the final judgement criteria are –for the most part- subject to personal opinion or company policy, and the task specifications. The study would suggest that Vue.js is most fitted for the integration of a front-end JSf into an existing application while React.js would be the choice for building mobile apps with JavaScript. The two frameworks have unique features and characteristics that may appear as supporting or inhibiting their ability to adequately implement a task, based on the scale and the nature of the desired implementation. There is no one-fits-all solution although the both behave particularly well in almost all of the metrics considered.

Chapter 6 Demonstration & Comparison of Applications Developement

In this chapter we will be further elaborating on the differences between the two frameworks as they were referenced in chapter 5.1 and presented in chapters 5.2.1-5.2.4, and will be further enriching the comparison with new points and inputs.
The method of this comparison will be the-step by step- presentation of the development of two applications in every framework, while commenting on and explaining the functionalities, which are divided into logical/functional units-sections. At the end of every section the main points will be evaluated and compared.

6.1 Selection of Applications

To-Do applications have long time proven their utility as tutorial-example apps for learning and demonstrating purposes in the web development and JS community. This is because - depending on their complexity and tasks they perform, but even in their simplest form, they utilize-and thus help to demonstrate- all the basic functionalities of the technology stack at hand i.e. a JSF in this case.
For the scope of this thesis, a fairly standard To-Do App was developed in React and Vue.
On the other hand, User Authentication and Authorization is a pre-requisite for almost any web application or even the most basic of websites, and represents the most basic level of security and data privacy and personalization. In this spirit, a Authentication-Authorization application with Registration and CRUD[3] functionalities was developed with the two frameworks (although not all of the functionalities are implemented because of the limited scope of the project-thesis).
The four frontends developed with the chosen frameworks have the exact same feature set and behavior –respectively- as pairs (To-Do applications-Authentication applications),

[3] (GET, POST, PUT, DELETE, PATCH)

within the same UI scenarios. The next sections go through how the scenarios were implemented.

For these implementations the following versions of the frameworks were used: React 17.0.2 and Vue 3.2.2. These versions support *React Hooks* (since React 16.8) and Vue *Composition API* [49] (since Vue 3.0.0) which are the new core features that have updated and enriched the way of writing the components, from *class* to *function* for React and *Options API* to *Composition API* for Vue.

6.2 To-Do Applications

The To-Do apps will now be built, focusing on how to add, remove, and edit data, as well as pass data from the parent to the child component as props, and vice verca in the form of event listeners.

The two frameworks will be showcased, and compared back to back on how the four functions of the application are performed in each case.

- Creating the To-Do List
- Add To-Do Item
- Add To-Do at enter press
- Delete Item

The above functions correspond to equal comparison points, as shown below:

- Data Mutation
- Detecting changes and updating state
- Event listeners
- Component communication

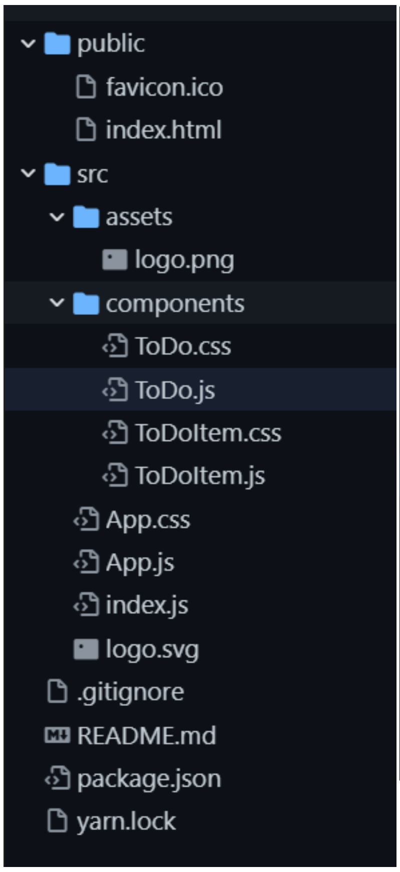
6.2.1 Developing the To-Do Applications

The development of each of the two frameworks' version of the To-Do app will be realized by completing each functionality-section, for each of the respective frameworks and then comparing the implementations before continuing to the next section.

Creating the Apps

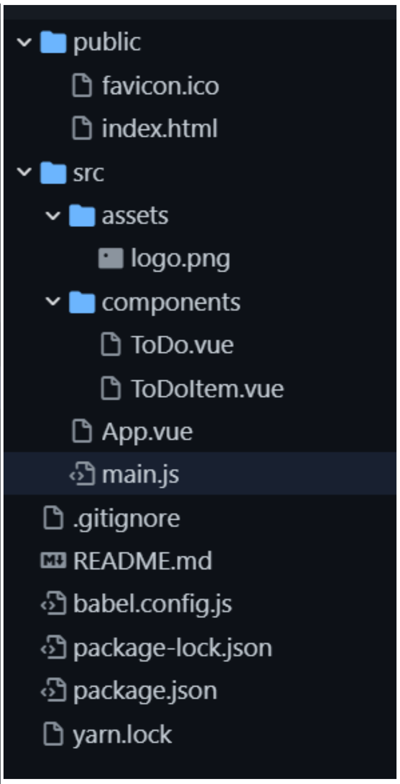
This section compares the file structure of the two frameworks.

The final file structure of the two applications is depicted in Figures 5, 6 bellow. There are two .CSS files for React app, each one corresponding to the respective *.js* file, create-react-app generates its default React components with their own CSS file as seen in Figure 6. In Vue (Figure 5) there are no standalone .CSS files. For its basic Vue components, Vue CLI creates single files including HTML, CSS, and JavaScript, a Vue architecture feature known as *Single File Components* (*SFC*) [50].



```
public
    favicon.ico
    index.html
src
    assets
        logo.png
    components
        ToDo.css
        ToDo.js
        ToDoItem.css
        ToDoItem.js
    App.css
    App.js
    index.js
    logo.svg
.gitignore
README.md
package.json
yarn.lock
```

Figure 6. File structure of React To-Do App



```
public
    favicon.ico
    index.html
src
    assets
        logo.png
    components
        ToDo.vue
        ToDoItem.vue
    App.vue
    main.js
.gitignore
README.md
babel.config.js
package-lock.json
package.json
yarn.lock
```

Figure 5. File structure of Vue To-Do App

In the end, they do the same thing, but the developer in React or Vue has the freedom to format the files differently. It's all a matter of personal preference. There is a lot of debate among developers about how CSS should be written, especially in the context of

React, where there are a variety of *CSS-in-JS* solutions like *styled-components* and *emotion*.

Creating the To-Do List

This section compares how the two frameworks perform data mutation.

The source code for *TodoItem.js* component in Fig.15, essentially represents a typical React component.

The same component in Vue in *Fig.16* represents the structure of the source code of a typical Vue file[4]. Although the *<style> …<style>* part of the component is not shown here, so that it is a complete analogy of React's *ToDoItem.js.*

In React, as seen in Listing 6.1 we are creating a list of *To-Dos* with *useState()* Hook which we import from React. This permits us to maintain our components' local state. It should normally be an empty array. The data inside the array are the initial data (for demonstration purposes). Normaly, we want it tobe an empty array. With *const [list, setList] = useState("…")*, two variables are created: *const list = "…"*, and *const setList* , which is assigned a function that enables list to be recreated(mutated) with a new value.

```
const [list, setList] = useState([
    { id: 1, text: "Pick up John" },
    { id: 2, text: "Buy groceries" },
  ]);
  const [toDo, setToDo] = useState("");
  const [showErr, setShowErr] = useState(false);
```

Listing 6.1 React state: Creating a list of *todos*

In Vue we are using *Composition* API. The *setup*() function, in Listing 6.2, contains and exposes to the app, all of the mutable data for the component and their functions. A *ref*() function is used to wrap each piece of state (data that we want to be able to change) data in our program. This *ref*() function is a Vue import that allows our app to update anytime any of the data changes. Here, *const list ("...")* plays the role of both *list* and *setList* of React's implementation and can be further referenced by calling *list.value.*

[4] The <style> …<style> part of the component –following the </script>- is not shown here, so that it is a complete analogy of React's ToDoItem.js, and the comparison can be clearer

```
setup() {
    const list = ref([
      { id: 1, text: "Pick up John" },
      { id: 2, text: "Buy groceries" }
    ]);
    const todo = ref("");
    const logo = Logo;
    const showErr = ref(false);
}
```

Listing 6.2 Vue state: Creating a list of *To-Dos*

Comparison - Data Mutation

React and Vue are both creating data that can be changed. Whenever a piece of data wrapped inside a *ref*() method is modified, Vue basically combines its own variant of *list* and *setlist*. In order to change state in React, you must first execute *setList*() with the value inside. Vue assumes that the developer will do so.

Add To-Do Item

This section demonstrates how the two frameworks listen for changes and update state. In React, there is a *value* property on the input field. For adding a new *ToDo*, a *onChange*() event listener JSX is used to automatically update this variable whenever its value changes.

```
<Input  type="text" placeholder="I need to..."value="{toDo}"
     onChange()="{handleInput}"/>
```

Listing 6.3 JSX for adding a new *To-Do* in React

Whenever the value is altered, it uses the *handleInput*() function to update state, as shown in Listing 6.4.

```
const handleInput = (e) => {
     setToDo(e.target.value);
};
```

Listing 6.4 Update state in React

The addition of a new item on the list is performed through the *createNewItem*() function, illustrated in Listing 6.5, which is called when the *"+"* button of the UI is selected by the user.

```
const createNewItem = () => {
const toDoId = newId();
     const newItem = { id: toDoId, text: toDo };
     setList([...list, newItem]);
     setToDo("");
};
```

Listing 6.5 Creation of new item in React

The *newId()* function (not shown) creates a new ID that will be passed to the new *To-Do* item.
Then, *setList()* function is run and an array is passed in that includes the entire *list* as well as the newly created *newItem().* Finally *setToDo()* is run and receives an empty string. This is done to ensure that the input value is blank, ready for the new To-Do to be entered.
The *input* field In Vue has a directive named *v-model* on it (Listing 6.6)**.** This enables *two-way binding***.** It ties the input of this field to the *todo* variable created in the *setup()* function and then exposes it as a key inside of the returned object.

```
<input type="text"
placeholder="I need to..."
      v-model="todo"
     v-on:keyup.enter="createNewItem"
 />
```

Listing 6.6. Vue Input field

In listing 6.7, the *stateful* values in the setup function's return object are *list*, *todo*, and *showErr*, while the rest are functions we want to be able to call from other parts of the app.

```
return {
  todo,
  list,
  showErr,
  createNewItem,
  newId,
  onRemoveItem,
  displayErr,
};
```

Listing 6.7 Return from *setup*() (*ToDo.vue*)

At page load, *todo* must be set as an empty string, like follows: *const todo = ref(""). todo.value* is linked to the text entered into the input area. Two-way binding allows the input field to update the ref() value and vice versa.

In Listing 6.8 below, by pushing *todo.value* into *list.value*, the contents of *todo.value* are inserted in the list array, before *todo.value* gets updated to an empty string. The *toDoId()* function is the same as used in the React example.

```
function createNewItem() {
  const toDoId = newId();
  list.value.push({ id: toDoId, text: todo.value });
  todo.value = "";
}
```

Listing 6.8 Create new *To-Do* item in Vue

Comparison - Listening for changes and updating state

In React, listening for changes in the input is accomplished through an *onChange()* event listener while the state updates accordingly through the *handleInput()* function. In Vue both the *input* and the *list* (state) updates are achieved through the two-way-binding feature with the *v-model* directive. Creating the item in the list also requires two steps in React versus one step in Vue. Vue seems to have a more compact and abstractive way of listening for changes and updating state, as seen at the creation of a new *To-Do* item, with its *v-model* directive and the composition API, while React needs to be more verbose to achieve the same functionality, much like in JS.

Add To-Do at *enter*

This section showcases and compares event listeners in the two frameworks.
Adding a *click* event that is triggered from clicking a button (Listing 6.9) and creates a new *To-Do* item in React, resembles handling an in-line *onClick* event in *vanilla JS*.

```
<button
className="ToDo-Add"
onClick=" {createNewItem}"> +
</button>
```

Listing 6.9 Add-*ToDo* button in React

In order for a new *To-Do* to be added when pressing the *enter* button, we setup an event listener to handle the event. This necessitates the input tag handling a *onKeyPress* event, as shown in Listing 6.10.

```
<input  type="text"  placeholder="I need to..."
  value="{toDo}"
  onChange()="{handleInput}"
  onKeyPress="{handleKeyPress}"
/>
```

Listing 6.10 Event listener for key-press in React

The *handleKeyPress* function triggers the *createNewItem()* function whenever the enter key is pressed, as in Listing 6.11 below.

```
const handleKeyPress = (e) => {
  if (e.key === "Enter") {
    createNewItem();
  }
};
```

Listing 6.11 At *enter*, create a new *To-Do* (React)

In Vue we just have to apply the *"@"* symbol, a shortcut for the *v:on* directive, on the type of event-listener we need. In listing 6.12, we add a click event listener at clicking the "+" button, while in Listing 6.13 the same functionality is assigned on pressing the *enter* key, again by using the *v-on* directive.

```
<button class="ToDo-Add" @click="createNewItem"> + </button>
```

Listing 6.12 Assigning functionality to a button with the *v-on (@)* directive in Vue

```
<input type="text"  v-on:keyup.enter = "createNewItem"/>
```

Listing 6.13 Assigning functionality to a key-press with the *v-on* directive in Vue

Comparison - Event listeners

In React, setting up event listeners for basic functions like click events is simple, as it is similar to how an in-line *onClick* event would be handled in *vanilla JS* **.** In such case, similarity with native JS is React's advantage. However, setting up a listener for creating a new *To-Do* item when the pressing the *enter* button, took a little bit longer in React.

In Vue, on the contrary, key-press events are as simple as click events, which are indeed very simple and straightforward as mentioned earlier. This is due to the power and simplicity of Vue directives.

Delete Item

This section compares how components communicate in the two frameworks.

In React, the *removeItem()* function, seen in Listing 6.15, is located inside *ToDo.js*. To be able to refer to it from within *ToDoItem.js* we pass on the *removeItem()* function to the child component (ToDoItem.js) as a prop, as in Listing 6.16 below, to make it accessible.

```
const removeItem = (id) => {
 setList(list.filter((item) => item.id !== id));
};
```

Listing 6.14 The React function for deleting item

```
<ToDoItem key="{item.id}" item="{item}" removeItem="{removeItem}" />
```

Listing 6.15. Deletion of *ToDoItem* component inside React's *ToDo*.js

Then inside <*ToDoItem/*> component we reference *removeItem* of the parent component (*ToDo.js*) function as *props.removeItem*. However, through destructuring (Listing 6.16) we have access to the function directly as *removeItem* (Listing 6.17).

```
const ToDoItem = (props) => { const { item, removeItem } = props; };
```

Listing 6.16. Destructuring

```
<button       className="ToDoItem-Remove"       onClick={()       =>
removeItem(item.id)}>
    -
</button>
```

Listing 6.17. Delete button in React

In Vue, to access the *onRemoveItem*() of *ToDo*.vue, we refer to the function at the very moment of adding the *ToDoItem*.*vue* (*child*) component inside *ToDo*.*vue* (*parent*) (Listing 6.18). Essentially, with the above, we create a custom event-listener that "listens" for any *emit* that the "*remove*" string triggers (Listing 6.19). In such occasion it calls the *onRemoveItem*() function which filters the id from the *list*.*value* array (Listing 6.20).

```
<ToDoItem v-for="item in list" :item="item" @remove="onRemoveItem"
:key="item.id" />
```

Listing Listing 6.18: *ToDoItem* component inside Vue's *ToDo.vue*

```
<button class="ToDoItem-Remove" @click="emit(“remove”,item.id)"> -
</button>
```

Listing 6.19: The delete item button in *ToDo.vue*

```
function onRemoveItem(id) {
  list.value = list.value.filter((item) => item.id !== id);
}
```

Listing 6.20: Filtering out the *To-Do* task that is meant for deletion in Vue

Comparison - Component communication

In both React and Vue, *props* are passed to the *child* component at the time of its creation. Props will give React components access to parent functions. The *props* must first be passed down to make it accessible to the *child* component [51]. In Vue, events, that will normally be "caught" at the *parent* component, have to be emitted from the *child* [52]. In our case Vue utilizes a custom event listener to call the parent's function.

Accessing a parent function i.e. emiting data back to the *parent* takes two steps in both React and Vue. It would actually need three steps in React if we hadn't used the *destructuring* assignment syntax JS expression[5]. The same would also be the case with Vue if the $emit part had not been placed inside the *@click* listener as in Listing 6.19, but was instead implemented as in Listing 6.21:

```
<button   class="ToDoItem-Remove"   @click=”removeItem(item.id))">-
</button>
```

Listing 6.21: The delete item button in *ToDo.vue*

[5] Destructuring is not supported by Internet Explorer

In this scenario, as in Listing 6.22 bellow, a *emit* function would have to be created in the *child* component (*ToDoItem.vue*) as a method:

```
function removeItem(id) {
  emit("remove", id);
}
```

Listing 6.22: Emit function inside *ToDoItem.vue* (alternative approach)

We would then have direct access to the *id* (as a key of the item *props*) by passing the item *props* to the child's *props* array, as follows: *"..., props: ["item"],... "* (see Figure 13 for the complete implementation).

As a conclusion Vue offers more flexibility in component communication, and specifically in emitting data from *child* to *parent*.

The complete source code for React and Vue To-Do applications is illustrated in Fig.12 13 14 15, 16, 17, and 18.

6.3 AUTH Applications

A back-end application will serve the Authentication-Authorization APIs for both front-end applications.

The front-ends will have the following features:

- Statefull components
- LocalStorage
- Protected resources
- Client-Side Routing
- Form validation

6.3.1 Backend

A *Node.js – Express* server is set up/developed for the needs of the front-end frameworks' presentation. It uses *MongoDB Atlas* as a database for storing user information and *Mongoose ODM* (*Object Document Mapping*) to perform CRUD (Create, Read, Update, Delete) operations to the database. For Authentication the application uses *JSON Web Token (JWT)* open standard for Token-based Authentication plus *bcryptjs* to hash and verify passwords.

Technology Stack

- Node.js v14.17.6.
- Express 4.17.1
- bcryptjs 2.4.3
- jsonwebtoken 8.5.1
- mongoose 5.9.1
- MongoDB Atlas

The APIs that the backend provides are shown in Table 2 below:

Methods	**Urls**	**Actions**
POST	/api/auth/signup	Register account
POST	/api/auth/signin	Login account
GET	/api/test/all	Access public content
GET	/api/test/user	Access User content
GET	/api/test/admin	Access Admin content

Table 2. API routes for the front-ends

Fig.7 below, presents a complete representation of SPA architecture that includes communication with a server, which resembles the applications' implementation that follows in section 6.2.2.

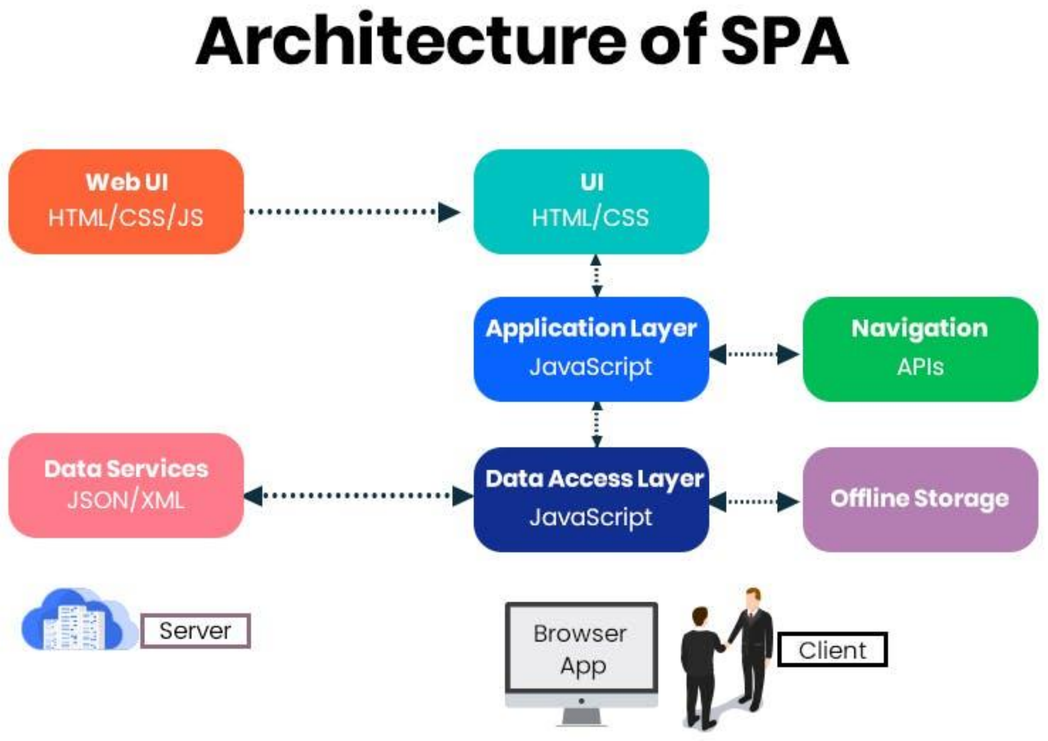


Figure 7. Architecture of SPA

Source: Adapted from [86].

6.3.2 Front-end

The presentation process will be organized into four basic sections, which also define the comparison format. For every (sub-)section, the same components' implementation in each framework will be presented and then compared before moving on to the next one. The application UIs will include the following components and functionalities which correspond to the (sub-)sections for presentation-comparison:

- Components for Authentication:
  - Login
  - Logout
  - Signup

  The front-end will validate the Form data before they are sent to the back-end.

- Public and Authorized-only routes:
    - Admin
    - User

  The routes depend on the user’s roles and the navigation bar’s items change dynamically in accordance.

- State management
- Authentication and Data Services

External Libraries – Packages – Modules

Both apps use the same or corresponding technology stacks which include external packages/modules for state management, routing, HTTP requests, validation and styles. For this project the following modules are used:

▪ **React**

redux 4.0.5	A Predictable State Container for JS Apps
react-redux 7.2	Official React bindings for Redux
redux-thunk 2.3.0	Async middleware for Redux.
react-router-dom 5	DOM bindings for React Router.
Axios 0.19.2	Promise-based HTTP client for the browser and node.js
react-validation 3.0.7	Component for simple form validation of React components.
validator 13.1.1	A library of string validators and sanitizers
bootstrap 4	Tool collection for styling components and creating responsive websites and web applications.

▪ **Vue**

vue-router 4	Vue.js official router
Vuex 4	State management library for Vue.js applications
Axios: 0.21.1	(As above)
vee-validate 4	Form Validation for Vue.js
bootstrap 4	(As above)

6.3.3 Development of the Applications

In the following sections the development process and comparison of the two applications is presented.

Create the Apps

- **React**

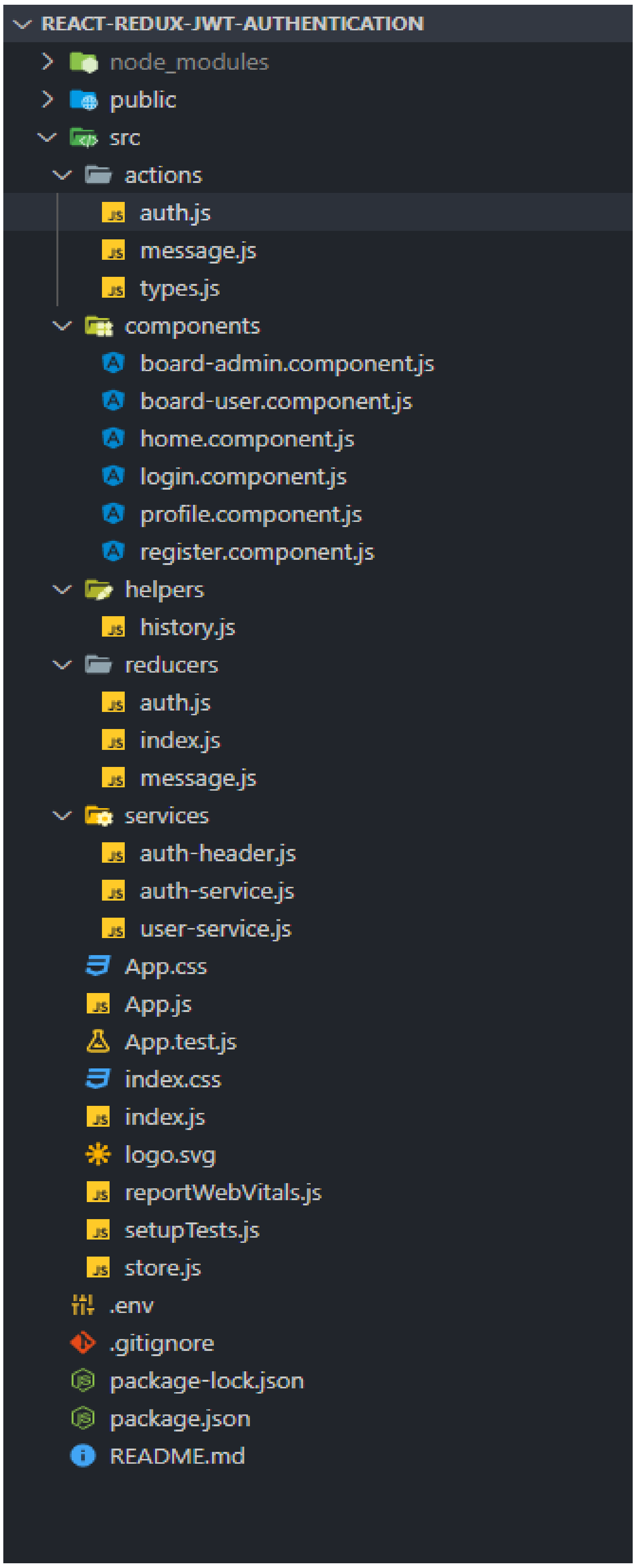
The following command is used for creating the React application:
npx create-react-app react-redux-JWT-authentication
React officially suggests *npx create-react-app* as it installs the latest version by default. *Create React App* creates frontend build pipeline, that can be used it with any backend. It leverages *Babel* and *Webpack* behind the scenes, but the user isn't required to understand or interact with them [53].
React can be used in other ways for different use cases [49].

- **Vue**

The following command is used to create the Vue application:
vue create vue3-Vuex-JWT-authentication
The Vue create command uses the *Vue CLI* which must be already installed from npm (npm install -g @vue/cli). The *CLI (Command Line Interface)* provides the user with certain preset/setup options. For the needs of this application the default preset is selected that provides a basic *Babel -ESLint* setup [54].
There are additional ways for installing Vue according to use-case scenarios [55]. Vue also offers another way for creating an app, with *Vite* instead of the *Vue CLI*, which offers better performance.

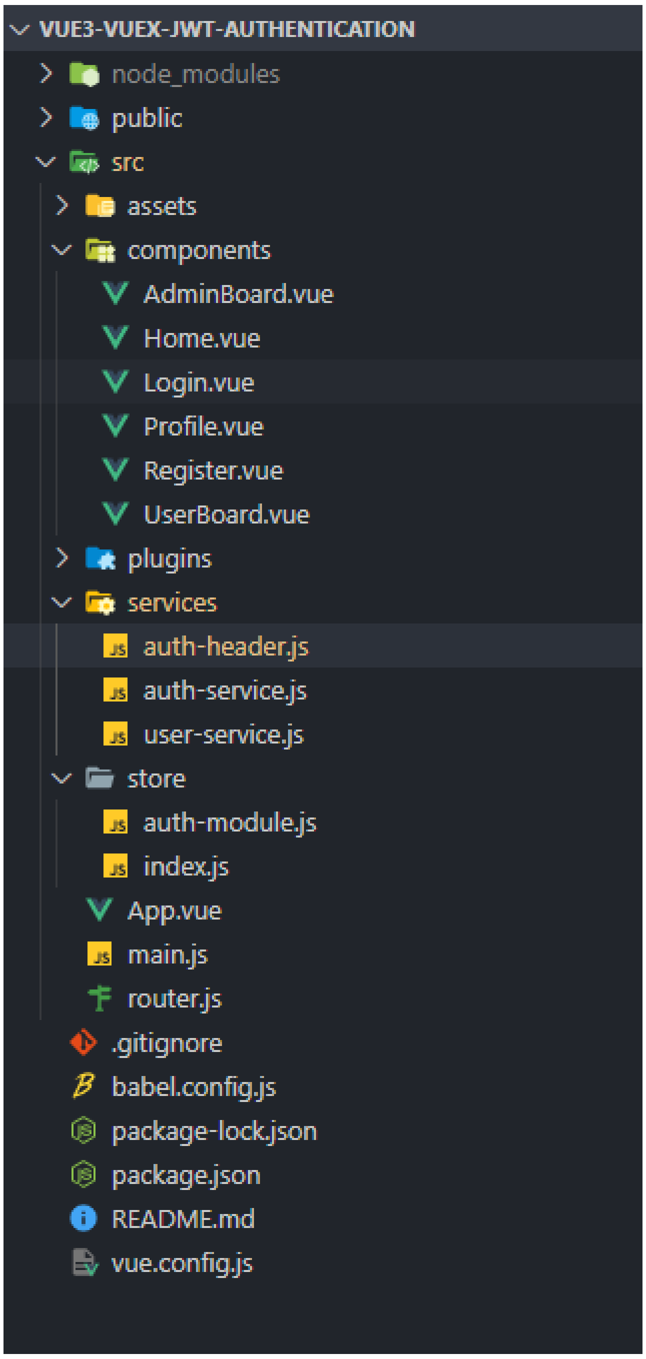
For both frameworks *Node.js* and *npm* must also be installed.
Both *create-react-app* and *vue-cli* scaffold a basic application out of the box which is served in localhost by running the *npm start* / *npm run serve* commands respectively. The apps can now be viewed in the browser at *localhost:8080*[6] or other port depending on the local environment. The *cli* will provide a direct link.
We can now proceed to modify the generated files accordingly and gradually create the applications.
The corresponding *devtools* for React and Vue will be added to the browser for debugging and component inspection.
The final folders & files structure for this React and Vue applications can be seen in Fig.8 and 9 respectively.

[6] If 8080 port is taken, a new one will be automatically assigned, usually 8081 or 8082, and the app will be served there.



```
REACT-REDUX-JWT-AUTHENTICATION
  > node_modules
  > public
  v src
    v actions
        auth.js
        message.js
        types.js
    v components
        board-admin.component.js
        board-user.component.js
        home.component.js
        login.component.js
        profile.component.js
        register.component.js
    v helpers
        history.js
    v reducers
        auth.js
        index.js
        message.js
    v services
        auth-header.js
        auth-service.js
        user-service.js
      App.css
      App.js
      App.test.js
      index.css
      index.js
      logo.svg
      reportWebVitals.js
      setupTests.js
      store.js
    .env
    .gitignore
    package-lock.json
    package.json
    README.md
```

Figure 8. File structure of React - AUTH application



```
VUE3-VUEX-JWT-AUTHENTICATION
  node_modules
  public
  src
    assets
    components
      AdminBoard.vue
      Home.vue
      Login.vue
      Profile.vue
      Register.vue
      UserBoard.vue
    plugins
    services
      auth-header.js
      auth-service.js
      user-service.js
    store
      auth-module.js
      index.js
    App.vue
    main.js
    router.js
  .gitignore
  babel.config.js
  package-lock.json
  package.json
  README.md
  vue.config.js
```

Figure 9. File structure of Vue - AUTH application

In Fig.8 and 9 (only) the folders specific to the application are shown as opened.

The components folders for the two applications is identical, with the exception of the files' suffixes. React has two extra files-actions, reducers- for *Redux* state management, while the Redux store is a single file residing in */src*. In Vue, the *Vuex* store represents a respective folder since it contains two separate files (the authentication module and the store). React's tree also contains a helper folder, where the routing helper is located, while both file trees have the same exact services folder with the two identical files. The file containing the CSS for the React application is located in the */src* folder whereas in Vue there is no such file since the CSS is embedded in every individual component (*SFCs*). The app's main components *App.vue* and *App.js*, as well as the entry points for each application- *main.js* and *index,js*- are also located in the */src* folder for Vue and React respectively. In Vue there is also the separate *router.js* file, for the routing configurations in contrary to React where no separate file is used. For the port configuration React uses the *.env* while Vue has a special *vue.config,js* file.

The rest of the folders and files are created at the apps' creation, from the scaffolding of the *CLI*/*npx* respectively, and are irrelevant for the scope of this application.

The files can also be organized in different ways. For example, another common practice for structuring a React app is to have each component in its own separate directory with its own *index.js* file, *index.*CSS file and presumably an additional *index.test.js* file, if testing configuration is present. Vue developers are also not constrained to using the *SFC* structure, but it is considered common-best practice and community support is based on this structure.

Application Overview

Below is the overview of the application. All steps are the same for each of the examined frameworks, thus presented once, with any framework-specific variations noted:

- The *App* components act as containers with React/Vue *Router*. The apps state is obtained via the *Redux/Vuex Store*. The *navbar* can then be displayed dependent on the current state.
- The Login and Signup components contain a form for submitting the user data, along with validation provided by the react-*validation*/*Vee-Validate* libraries. They send authorization-authentication actions (login/signup) to the *Redux Thunk Middleware* / *Vuex store* which call the API using *auth-service*.
- The *auth-service* methods make HTTP requests using *Axios*. Within these methods, the *JWT* is likewise stored and retrieved from the Browser *Local Storage*.
- All visitors can see the *Home* component.
- After the login action is completed, the *Profile* component displays the user information.

- *The BoardUser and BoardAdmin* components are featured in the navbar - or not - according to the *user roles*. Protected resouses can be accessed from the web API by using the *user service* in these components.
- A helper function is used by the *user service* to insert the *JWT* to the HTTP header. The function, exposes an object that contains the *JWT* of the currently logged in user, as retrieved from the Local Storage.

As made obvious from the above description, the logical structure of the two applications is almost identical in terms of implementation logic.

Development Process

The Applications' Entry Points

First, we setup the entry point for each application, in *index.js* for React and *main.js* for Vue, where we modify the auto-generated code and make the appropriate imports as in Listing 6.23 and 6.24 respectively.

- **React**

The <*Provider*> component in *index.js* (Listing 6.23),is needed so that the appropriate nested components are able to access the *Redux* store. The *index*.css is the application's CSS file.

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import store from "./store";
import "./index.css";
import App from "./App";

ReactDOM.render(
  <Provider store = store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

Listing 6.23 React’s application entry point: *index.js*

- **Vue**

In *src/main.js*, we modify the original code inside as follows (Listing 24):

```
import { createApp } from "vue";
import App from "./App.vue";
import store from "./store";
import router from "./router";
import "bootstrap";
import "bootstrap/dist/css/bootstrap.min.css";

createApp(App)
  .use(router)
  .use(store)
  .mount("#app");
```

Listing 6.24: Vue’s application entry point: *main.js*

Auth & User Services

The *Auth* and *User* services are the applications’ entry points for communication with the API.

The Services, described below, are identical for both React and Vue applications.

In src/services folder, we create the following files:

services/
  user-auth.js
  auth-service.js
  user-service.js

The above files will host the Helper function, the Authentication service, and the Data service respectively.

It should be noted that *Axios* must be installed, before proceeding to interact with the Services.

Authentication service

For HTTP requests, the service uses *Axios*, and for user information and *JWT*, it uses *Local Storage*.

Axios provides the following methods:

- POST: Used to make a post request to the server with the user's data at signup/login , and also save the *JWT* to the browser's *Local Storage* at login.
- Delete: Used to delete the *JWT* from the browser's *Local Storage at logout*.

The above methods are used in signup(), login(), and logout() functions as demonstrated below in Listing 6.25.

```
import Axios from "Axios";
const API_URL = "http://localhost:8080/api/auth/";
class AuthService {
     login(name, pass) {
            return Axios
                   .post(API_URL + "signin", { name, pass })
                   .then((res) => {
                          if (res.data.accessToken) {
     localStorage.setItem("user", JSON.stringify(res.data));
                           }
                   return res.data;
                   });
     }

     logout() {
            localStorage.removeItem("user");
      }

     signup(name, email, pass) {
            return Axios.post(API_URL + "signup", {
                   name,
                   email,
                   pass,
             });
      }
}
export default new AuthService();
```

Listing 6.25: Authentication Service: *auth-service*.js

Data Service

To retrieve data from the server we make use of the available methods. In order to access protected resources, an Authorization header must be included in the HTTP *request.* For this reason a helper function called userAuth() is defined inside the *user-auth.js* file (Listing 6.26).

The function looks for user item in *Local Storage*. It returns the HTTP Authorization header with the JWT accessToken in case of a logged in user present, or an empty object otherwise.

```
export default function userAuth() {
const user = JSON.parse(localStorage.getItem('user'));
if (user && user.accessToken) {
return { 'x-access-token': user.accessToken };
} else {
return {};
       }
}
```

Listing 6.26 *User-auth.js* (Helper function)

Now, in *user-service.js*, we establish a service to provide data access, as shown in Listing 6.27, by using the *userAuth()* function to add an HTTP header when requesting authorized resources.

```
import Axios from 'Axios';
import userAuth from './user-auth';
```

```
const API_URL = 'http://localhost:8080/api/test/';
class UserService {
     getPublicInfo() {
            return Axios.get(API_URL + 'all');
     }

     getBoardUser() {
            return Axios.get( API_URL    +       'user',{headers:
userAuth() });
     }

     getBoardAdmin() {
            return   Axios.get(   API_URL   +   'admin',   {   headers:
userAuth() });
      }
}
export default new UserService();
```

Listing: 6.27: Data Service: *user-service.js*

State Management

- **React**

Redux

"Redux is a predictable state container for JavaScript applications. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger." [56]

Even though *Redux* is not an official direct recommendation for State Management in React, as no other alternative is neither, it is nevertheless hinted as such since it is mentioned many times throughout the official React documentation. In any case, it has been the most used solution in the React community for years, far ahead the "competition" as well [57].

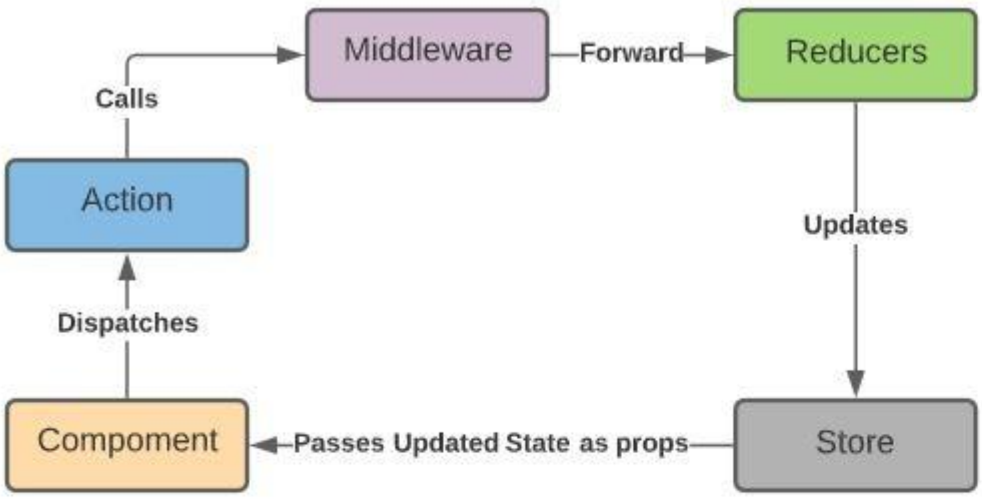


Figure 10. *Redux* overview

Source: Adapted from [87]

Create Redux Actions

The Store's only source of truth is Actions. These are plain JS objects that feature a type field that specifies the type of action to be taken, while the other fields provide information/data. Two kinds of Actions will be created in *src/actions* folder:

actions/

auth.js

message.js

The above files contain the signup/login/logout, and set/clear message actions respectively.

The *types.js* file, which specifies the type field, will also be created.

Action Types

The string constants for determining every action must first be defined, as in Listing 6.28:

```
export const SIGNUP_SUCCEEDED  = "SIGNUP_SUCCEEDED ";
export const SIGNUP_FAILED = "SIGNUP_FAILED";
export const LOGIN_SUCCEEDED  = "LOGIN_SUCCEEDED ";
export const LOGIN_FAILED = "LOGIN_FAILED";
export const LOGGED_OUT = "LOGGED_OUT";
export const SET_MSG = "SET_MSG";
export const CLEAR_MSG = "CLEAR_MSG";
```

Listing 6.28: Action Types file: *actions/type.js*

Following, the Action Creators - functions that create and return the Actions objects- will be defined.

Message Action Creator

The folowing Listing 6.29 shows the *Redux* action creator responsible for the *APIs* notifications .

```
import { SET_MSG, CLEAR_MSG } from "./types";
export const setMsg = ( msg ) => ({
     type: SET_MSG,
     payload: msg ,
});

export const clearMsg = () => ({
     type: CLEAR_MSG,
});
```

Listing 6.29: Message Action Creator file: *actions/message.js*

Authentication Action Creator

In Listing 6.30, the creator for authentication-related operations is presented. We use *AuthService* to send – asynchronous - HTTP requests that result in dispatch(es) which is/are then handled by the reducers.

- *AuthService.signup(...)* is called by the *signup()* method.
    - If the registration is successful, *SIGNUP_SUCCEDED* and *SET_MSG* are dispatched.

    - If the registration fails, *SIGNUP_FAILED* and *SET_MSG* are dispatched.
- *AuthService.login(..)* is called by the *login()* method
    - If the login is successful, *LOGIN_SUCCEEDED* and *SET_*MSG *are* dispatched.
    - If the login fails, *LOGIN_FAILED* and SET_MSG are dispatched

Note that a *Promise* is returned for the components that use them.

```
import {
  SIGNUP_SUCCEEDED ,
  LOGIN_SUCCEEDED ,
  SIGNUP_FAILED,
  LOGIN_FAILED,
  SET_MSG,
  LOGGED_OUT,
} from‟. / types”;
import AuthService from‟.. / services / auth-service”;
export const signup = (name, email, pass) => (dispatch) => {
  return AuthService.signup(name, email, pass).then(
    (res) => {
      dispatch({
        type: SIGNUP_SUCCEEDED
      });
      dispatch({
        type: SET_MSG,
        payload: res.data.msg,
      });
      return Promise.resolve();
    },
    (err) => {
      const msg  = (err.res && err.res.data && err.res.data.msg) ||
err.msg || err.toString();
      dispatch({
        type: SIGNUP_FAILED,
      });
      dispatch({
        type: SET_MSG,
        payload: msg ,
      });
      return Promise.reject();
    });
};
export const login = (name, pass) => (dispatch) => {
  return AuthService.login(name, pass).then(
    (data) => {
      dispatch({
        type: LOGIN_SUCCEEDED ,
        payload: {
          user: data
        },
      });
      return Promise.resolve();
    },
    (err) => {
      const msg  = (err.res && err.res.data && err.res.data.msg) ||
err.msg || err.toString();
      dispatch({
        type: LOGIN_FAILED,
      });
      dispatch({
        type: SET_MSG,
        payload: msg ,
      });
      return Promise.reject();
    });
};
export const logout = () => (dispatch) => {
  AuthService.logout();
  dispatch({
    type: LOGGED_OUT,
  });
};
```

Listing 6.30: Authentication Actions Creator: *actions/auth.js*

Create Redux Reducers

In the *src/reducers* folder, two *reducers* will be present, each of which manages a distinct part of the application state in accordance to dispatched *Redux actions*. The two *reducers* will then be combined at *index.js.*

reducers /

index.js

auth.js

message.js

The reducers correspond to the *signup/login/logout*, and *set/clear message* Actions respectively.

Message Reducer

The *reducer* that revisions the message state according to the previous state and a dispatched message action (Listing 6.31).

```
import { SET_MSG, CLEAR_MSG } from "../actions/types";
const initState  = {};
export default function (state = initState , action) {
     const { type, payload } = action;
     switch (type) {
            case SET_MSG:
            return { msg : payload };
            case CLEAR_MSG:
            return { msg : "" };
            default:
            return state;
      }
```

Listing 6.31: Message Reducer: *reducers/message.js*

Auth Reducer

This reducer (Listing 6.32) is responsible for updating the *isLoggedIn* state, as well as that of the user, according to the message state.

```
import {
  SIGNUP_SUCCEEDED ,
  SIGNUP_FAILED,
  LOGIN_SUCCEEDED ,
  LOGIN_FAILED,
  LOGGED_OUT,
} from‟.. / actions / types”;
const user = JSON.parse(localStorage.getItem(“user”));
const initState  = user
 ? { isLoggedIn: true, user}
 : { isLoggedIn: false, user: null};
export default function(state = initState , action) {
  const {
    type,
    payload
  } = action;
  switch (type) {
    case SIGNUP_SUCCEEDED :
      return {
        ...state,
        isLoggedIn: false,
      };
    case SIGNUP_FAILED:
      return {
        ...state,
        isLoggedIn: false,
      };
    case LOGIN_SUCCEEDED :
      return {
        ...state,
        isLoggedIn: true,
          user: payload.user,
      };
    case LOGIN_FAILED:
      return {
        ...state,
        isLoggedIn: false,
          user: null,
      };
    case LOGGED_OUT:
      return {
        ...state,
        isLoggedIn: false,
          user: null,
      };
    default:
      return state;
  }
}
```

Listing 6.32: Authentication Reducer: *reducers/auth.js*

Combine Reducers

Because a *Redux* application (usually) has only one store, we utilize *reducer* composition rather than using several stores to handle the data processing logic [58] .This is performed in *index.js* file (Listing 6.33).

```
import { combineReducers } from “redux”;
import auth from “./auth”;
import msg  from “./message”;

export default combineReducers({ auth, msg  });
```

Listing 6.33: Combine Reducers: *reducers/index.js*

Redux Store

The *Store* is tasked with connecting *Actions* and *Reducers* together as well as maintaining the application's state.

At this point, *Redux* and *Thunk* Middleware need to be installed. We can also install *Devtool Extension*[7] as a developer dependency as such: “*npm install—save-dev redux-devtools-extension”*.

After the modules’ installalation, combineReducers() is now imported, which was used in order to combine the two *reducers* into one in the previous section, and pass it to *createStore*() (Listing 6.34):

```
import { createStore, applyMiddleware } from “redux”;
import { composeWithDevTools } from “redux-devtools-extension”;
import thunk from “redux-thunk”;
import rootReducer from “./reducers”;

const middleware = [thunk];
const store = createStore(rootReducer,
            composeWithDevTools(applyMiddleware(...middleware))
            );
 export default store;
```

Listing 6.34: Redux Store*: src/ store.js*

[7] Helps us to use visualize our redux store in the browser using the corresponding browser extension.

- **Vue**

Vuex

“*Vuex* is a state management pattern + library for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion. It also integrates with Vues’ official devtools extension to provide advanced features such as zero-config time-travel debugging and state snapshot export / import” [59].

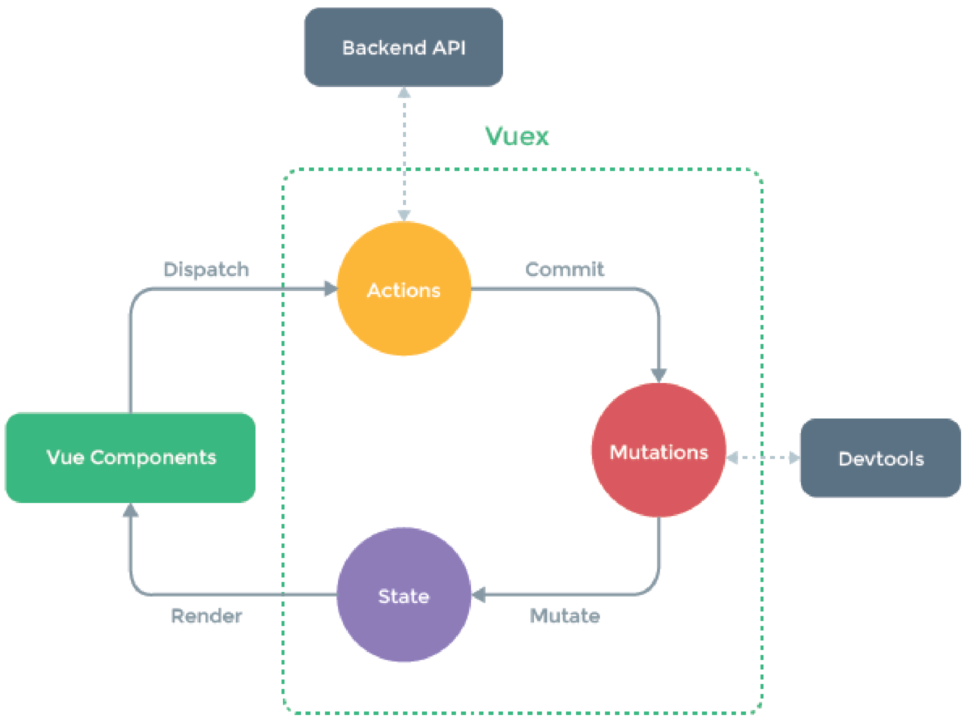


Figure 11. Overview of *Vuex*

Source: Adapted from [60]

***Vuex* Authentication module**

The Vuex authentication-module is placed in *src/store* folder:

store/

auth.module.js

index.js

The above two files contain the authentication module and the Vuex Store - that contains all the relevant modules - respectively.

In Listing 6.35 below, we import *auth.module* to main *Vuex Store* in *index.js*.

```
import { createStore } from "Vuex";
import { auth } from "./auth.module";

const store = createStore({
  modules: {
    auth,
  },
});
export default store;
```

Listing 6.35 Vuex store: *index.js*

Then, we define the *Vuex Authentication* module (Listing 6.36), which includes the following:

- *state*
- *actions*
- *mutations*

More specifically the status & user state, the login/logout/signup actions, and the loginSucceeded/loginFailed/loggedOut/signupSucceeded/signupFailed mutations, respectively.

AuthService is utilized to pefrorm authentication requests:

```
import AuthService from '../services/auth-service';

const user = JSON.parse(localStorage.getItem('user'));
const initState  = user
  ? { status: { loggedIn: true }, user }
  : { status: { loggedIn: false }, user: null };
```

```
export const auth = {
  namespaced: true,
  state: initState ,
  actions: {
    login({ commit }, user) {
      return AuthService.login(user).then(
        user => {
          commit('loginSucceeded', user);
          return Promise.resolve(user);
        },
        err => {
          commit('loginFailed');
          return Promise.reject(err);
        }
      );
    },
    logout({ commit }) {
      AuthService.logout();
      commit('logout');
    },
    signup({ commit }, user) {
      return AuthService.signup(user).then(
        res => {
          commit('signupSucceeded');
          return Promise.resolve(res.data);
        },
        err => {
          commit('signupFailed');
          return Promise.reject(err);
        }
      );
    }
  },
  mutations: {
    loginSucceeded(state, user) {
      state.status.loggedIn = true;
      state.user = user;
    },
    loginFailed(state) {
      state.status.loggedIn = false;
      state.user = null;
    },
    logout(state) {
      state.status.loggedIn = false;
      state.user = null;
    },
    signupSucceeded(state) {
      state.status.loggedIn = false;
    },
    signupFailed(state) {
      state.status.loggedIn = false;
    }
  }
};
```

Listing 6.36: Vuex Authentication module: *auth.module.js*

State Management Comparison

First of all because *Redux* is a standalone library, not specific to React, we need to import other libraries e.g the *React-Redux* library so that React components are able to interact with the store, as well as middlewares for *async* logic (*Thunk*), for it to be functional [61]. *Vuex* on the other hand is tightly coupled with Vue, hence there is less boilerplate code for *Vuex* to get started, which is clearly seen in the import statements of files *store*.js and *index*.js respectively.

Another thing to be noted here is that with first-party solutions there are less decisions to be made on behalf of the developer. For example on React side, the first decision to be made is the choice of a State management library e.g. *Redux* or *Mobx*? Then, let's assume *Redux* is chosen, another decision needs to be made in regards of middleware: *Thunk* or *Saga*? This is, interestingly, the same main "problem" this thesis is examining, just on a higher level -meaning it is not as equally an important decision- but the reasoning is exactly same: How to navigate and make choices in a world full of numerous frameworks/libraries/packages etc. Of course there is the other side of the coin in this situation also, which is that there is less flexibility with the more integrated solutions like

Vue’s. So, in the end it is a tradeoff. That being said, *Vuex* is also not without its alternatives, like the lightweight *Pinia* [62] and other- mostly lightweight- variations of *Vuex* [63] although none of them is widely adopted.

React has a more modular approach as is to be expected since it is a library so it requires external libraries/packages for further functionality. Moreover, its modular logic is also present “internally” to React’s structuring logic as well as that of its various solutions libraries like *Redux*. Vue on the other hand has the logic of providing more compact solutions as it is single file component implementation seems to also reflect on its state management solution (*Vuex*). As we can see in *Vuex* the whole logic is contained in the same file (*auth.module.js*) whereas in *Redux* it is divided into two separate files each one representing a single logical unit (*actions* and *reducers*).

Of course, as said before, nothing inhibits the developer from also separating logic in *Vuex* by abstracting *actions* and *mutations* to separate files and import them into the store but again, this is the best practice and the intended structuring way of the library’s creator(s) [64].

As for similarities, they share an important concept: both *Redux* and *Vuex* –usually- use only one store -a single state management “*object store*”-, thus the global state is a “*single source of truth*”, even though they are both based on the *Flux* pattern in which multiple stores are a common practice [65].

They also both have their corresponding *devtools*, the standalone *Redux devtools* and the –seamless- integration with Vue.js *devtools* for *Vuex*.

As for *mutating state*, while *Redux* uses *reducers*, *Vuex* uses *mutations*. In *Redux*, as in React, state is always immutable, while in *Vuex* committing mutation by the store is the only way to change data, since there is no dispatcher like *Redux*. We can see how *Vuex* is directly assigning new values to state variables in auth.modules.js (in the corresponding mutations part) while Redux *Reducers* calculate a new state -taking the previous state and an action as input – to be then combined with the other reducers at the store, as a root reducer through the *combineReducers*() function. As before in the *To-Do* applications, here again the same pattern is noticed: Vue tends to “directly” access values (either through bi-directional -as in *two-way binding*- or *uni-directiona*l -as in *Vuex*- data flow) while React takes a “longer route” following the functional programming paradigm through the use of

pure functions[8] as *Redux's reducers**: "React and Redux both need pure functions coupled with immutability to run in a predictable fashion" [66].
Although *Redux* is a framework agnostic library there seems to be a correlation between its logic of implementation and that of React's**,** that's why it has been the standard choice for state management for years, even resulting in many developers mistaking it as a React specific-implementation, as *Vuex* is for Vue.
From the above, it can be safely deduced that, comparing *Redux* to *Vuex*, although not directly a React-Vue comparison, is still relevant for our scope and adds complementary value and meaning to the overall comparison.

Components for Authentication

Rather than utilizing *Axios* or AuthService directly, the Authentication Components communicate and dispatch calls to Redux Thunk Middleware for React, which in turn gets State from Redux and makes the requests to *auth-service*. Instead they communicate with *Vuex* Store directly, in Vue's case, returning status through this.$store.state.auth and making the http requests to *auth-service*.

- **Login Page**
  - **React**

In the *components* folder at the *src* level, the following files are added to host the respective components:
components/
  login.component.js
  signup.component.js
  profile.component.js

The login page presents a form with two fields: username and password. The user's inputs are validated in real-time. Once the validation passes, the User gets through Authentication

[8] A function that is free from side-effects and for the same input value it always produces the same result.

process and - provided that the process is successful - has now access to protected resources.

Form Validation

For input fields validation we use react-validation [67], a library –a component actually- that provides basic form validation for React components. According to the Controlled Components [68] approach, the displayed value is tied to the state of the component. A function associated to the form element's *onChange()* event handler is called to update the value. Updates. The state property gets updated by the onChange() function, and in turn the form element’s value is also updated.

As the react-validation prompts in the instructions [67], we define the required() function which is then passed as an array in the validations attribute in the form. The form itself is implemented by the <Form/> component, which is a wrapper around the native form component, and is imported from react-validator, as is the Input component which hosts the two input fields: username & password.

The Form validateAll() method is then used to verify the validation functions in validations. The CheckButton component is used to verify whether or not the form validation was successful, and it does not appear on the form (display: none). It uses React’s ref property to reference the Form component.

Login

A form with a username and password is found on the Login page.

– The form’s fields will be verified as required fields.

– If the validation is successful, the user gets forwarded to the Profile page after completing the login process, which involves a login action being dispatched.

For accessing the application state we use Redux *connect()* function with *mapStateToProps()*:

– Redirect user to Profile page by checking *isLoggedIn.*

– Show response message with *msg.*

First of all, since we have chosen to define React’s components as classes, we need to implement a constructor to be able to initialize state and bind methods, as seen in Listing

6.37. At the constructor’s implementation, *super(props)* must be called before any other statement. The constructor gets called first and then the component is mounted [69].
The *connect()* function links a React component to a Redux store [70], for managing state, while the *mapStateToProps()* function selects the part of the data from the store that the connected component need. At *render()* lifecycle method we check if user is logged in, according to the state coming from the props that are getting passed to it through *mapStateToProps()*. If true, the user gets redirected to Profile page by the Redirect component of react-router-dom module which is imported.
If false, the user will proceed to provide username and password. The submission of the new data in the form fields triggers the *onChangeName()* and *onChangePass()* functions respectively which set the new values of the state variables through the setState() function [69]. Since the state is immutable and cannot be updated directly in React, The *setState()* fuction is the only proper way to do it.
At the form submission, *handleLogin()* function is called which sets the loading state, calls the forms’ *validateAll()* function and if the validation passes, a login action is dispatched. The user gets redirected to the profile page through the *history.push()* function of react-router-dom. In any case, the loading state is again updated.
The Login button element has an effect on it which shows on screen or not, according to the loading state value (true/false) and finally the appropriate response message is shown.

```
import React, { Component } from “react”;
import { Redirect } from ‘react-router-dom’;

import Form from “react-validation/build/form”;
import Input from “react-validation/build/input”;
import CheckButton from “react-validation/build/button”;

import { connect } from “react-redux”;
import { login } from “../actions/auth”;

const required = (value) => {
  if (!value) {
    return (<div className=”alert alert-danger” role=”alert”>
This field cannot be empty!
```

```
</div>);
  }
};
class Login extends Component {
  constructor(props) {
    super(props);
    this.handleLogin = this.handleLogin.bind(this);
    this.onChangeName = this.onChangeName.bind(this);
    this.onChangePass = this.onChangePass.bind(this);
    this.state = {
      name: “”,
      pass: “”,
      loading: false,
    };
  }
  onChangeName(e) {
    this.setState({
      name: e.target.value,
    });
  }
  onChangePass(e) {
    this.setState({
      pass: e.target.value,
    });
  }
  handleLogin(e) {
    e.preventDefault();
    this.setState({
      loading: true,
    });
    this.form.validateAll();
    const {
      dispatch,
      history
    } = this.props;
    if (this.checkBtn.context._errors.length === 0) {
      dispatch(login(this.state.name, this.state.pass)).then(() => {
          history.push(“/profile”);
            window.location.reload();
          }).catch(() => {
          this.setState({
            loading: false
          });
        });
      }
      else {
        this.setState({
          loading: false,
        });
      }
    }
    render() {
      const {
        isLoggedIn,
        msg
      } = this.props;
      if (isLoggedIn) {
        return <Redirect to="/profile" />;
    }

return (
<div className="col-md-12">
<div className="card card-container">
<img
  src="//ssl.gstatic.com/accounts/ui/avatar_2x.png"
  alt="profile-img"
  className="profile-img-card"
  />
<Form
  onSubmit={this.handleLogin}
  ref={c =>
  {
  this.form = c;
  }}
  >
  <div className="form-group">
    <label htmlFor="name">Username</label>
    <Input
      type=”text”
      className=”form-control”
      name=”name”
      value={this.state.name}
      onChange()={this.onChangeName}
      validations={[required]}
      />
  </div>
  <div className=”form-group”>
    <label htmlFor=”pass”>Password</label>
    <Input
      type=”pass”
      className=”form-control”
      name=”pass”
      value={this.state.pass}
      onChange()={this.onChangePass}
      validations={[required]}
      />
  </div>
  <div className=”form-group”>
    <button
      className=”btn btn-primary btn-block”
      disabled={this.state.loading}
      >
    {this.state.loading && (
    <span className=”spinner-border spinner-border-sm”></span>
    )}
    <span>Login</span>
    </button>
  </div>
  {msg  && (
  <div className=”form-group”>
    <div className=”alert alert-danger” role=”alert”>
      {msg }
    </div>
  </div>
  )}
  <CheckButton
  style={{ display: “none” }}
```

```
  ref={ c => {
  this.checkBtn = c;
  }}
  />
</Form>
< /div> < /div>);
}
}
  function mapStateToProps(state) {
    const {
      isLoggedIn
    } = state.auth;
    const {
      msg
    } = state.msg;
    return {
      isLoggedIn,
      msg
    };
  }
  export default connect(mapStateToProps)(Login);
```

Listing 6.37: React Login page: *login.component.js*

- **Vue**

In the *components* folder at *src* level, the following files will host the respective Vue components:
components/
  Login.vue
  Signup.vue
  Profile.vue

Form Validation

The form and fields of the page are built with the VeeValidate's *Form* and *Field* components. They are imported from VeeValidate 4.x and are used to validate input. These are higher-order components (HOC) that - based on the name of the field - automatically hook into the validation rules (schema). As seen in Listing 6.38 the Form component has two Field components: username & password. If a field is invalid, an error message is displayed. The *ErrorMessage* component allows you to display error messages without having to use scoped slots on the *Form* or *Field* components [71]. Nothing is rendered if there are no messages concerning the associated field. The validation schema with the appropriate messages are defined in the *data()* function by Yup, which is also imported.

Login

Initially we check the user logged-in status using the declared computed property *loggedIn()* and *Vuex* Store as such*: "this.$store.state.auth.status.loggedIn"*. Because the store option is provided to the root instance (in *index.js*) the above is made possible. The store will thus be injected into all child components who will be able to access it as *this.$store* [72]. If the status from the store changes, *loggedIn* value updates accordingly. This occurs because *Vuex* stores are reactive, which means that if the store's state changes, the retrieved state will update reactively and effectively. If the status is true, it means the user is logged in, thus the *created()* lifecycle hook [73] and Vue Router are utilized to forward them to the *Profile* page.
In case that the user is not already logged in, at form submission, the task is being handled by the *handleLogin()* method. First the loading state is set to true which affects the appearance and behavior of the login button as seen in the template in Listing 6.38. Then, *Vuex* dispatches the login action for the user and the router redirects the user to profile page on success, else the loading state is updated to false and the appropriate message is shown as it is defined in the *Vuex* authentication module.

```
<template>
  <div
     ...
      />
      <Form @submit="handleLogin" :validation-schema="schema">
        <div class="form-group">
          <label for="name">Username</label>
          <Field name="name" type="text" class="form-control" />
          <ErrorMessage name="name" class="error-feedback" />
        </div>
        <div class="form-group">
          <label for="pass">Password</label>
          <Field name="pass" type="pass" class="form-control" />
          <ErrorMessage name="pass" class="error-feedback" />
        </div>

        <div class="form-group">
          <button      class="btn      btn-primary      btn-block"
:disabled="loading">
            <span
              v-show="loading"
              class="spinner-border spinner-border-sm"
            ></span>
            <span>Login</span>
          </button>
        </div>

        <div class="form-group">
          <div     v-if="message"     class="alert     alert-danger"
role="alert">
            {{ msg }}
          </div>
        </div>
      </Form>
    </div>
  </div>
</template>

<script>
import { Form, Field, ErrorMessage } from "vee-validate";
import * as yup from "yup";

export default {
  name: "Login",
  components: {
    Form,
    Field,
    ErrorMessage,
  },
  data() {
    const schema = yup.object().shape({
      name: yup.string().required("Username cannot be empty !"),
      pass: yup.string().required("Password cannot be empty !"),
    });

    return {
      loading: false,
      msg : "",
      schema,
    };
  },
  computed: {
    loggedIn() {
      return this.$store.state.auth.status.loggedIn;
    },
  },
  created() {
    if (this.loggedIn) {
      this.$router.push("/profile");
    }
  },
  methods: {
    handleLogin(user) {
      this.loading = true;

      this.$store.dispatch("auth/login", user).then(
        () => {
          this.$router.push("/profile");
        },
        (err) => {
          this.loading = false;
          this.msg =
            (err.res &&
              err.res.data &&
              err.res.data.msg) ||
            err.msg ||
            err.toString();
        }
      );
    },
  },
};
</script>

<style scoped>
...
</style>
```

Listing 6.38: Vue Login page: *Login.vue*

Login Page Comparison

Vue, again -as in the Redux-*Vuex* comparison- has less boilerplate code, since its router and state management solutions are first-party-solution libraries. React on the other hand has to import all corresponding companion libraries.

Vue's -roughly-equivalent for React's constructor is *beforeCreate()* and it usually does not have to be explicitly called. Unlike React where the constructor sets up the class, Vue handles the class creation for you.

For form validation they both import and use the "same" wrapper-components, although their respective libraries implementations used here are based on different approaches (HOC for *VeeValidate* and Controlled Components for *react-validation*).

In both approaches the displayed value is bound to component state. *VeeValidate* abstracts (”automates”) the binding to the input values. Where v-model would normally be used is now not needed because *VeeValidate* implements it under the hood by constructing an internal model, which "watches" and keeps the <Field /> component's field instances and the input synced [74]. To update the value under the Controlled Component's react-validation approach, a function linked to the *onChange*() event handler on the form element is called. The *onChange*() function modifies the state property, which modifies the value of the form element.

Vue's –actually *VeeValidate's*- implementation is clearly more abstract, easier and requires far less code in comparison to React's/react-validation's verbose implementation. On readability side, still the same is true but there could be a caveat to Vue's completely abstracted implementation: The reviewing of the code by another developer could introduce an "obstacle" to understanding the functionality, if they are not familiar with *VeeValidates* implementation. Of course it would require just a quick review of the documentation to understand how this is working, but then again in a comparison like this, concerning first and foremost the developer's experience- where the required time and effort always represent an important factor-, everything counts and ads up to form the overall evaluation.

Finally Vue uses a separate < *ErrorMessage* > component for handling error messages while React uses the < *CheckButton* > component to check if verification passes. One "point" for each here as: On the one hand Vue uses an extra component for error displaying which React encapsulates into the Redux messages, while on the other hand React uses the – indeed "strange"- (at least to the authors' eyes*) <CheckButton>* component and *checkbtn()* function for checking if the validation is successful, which Vue accomplishes elegantly by *VeeValidates'* abstraction, with again the same problematic.

For the actual login process, React passes the *onSubmit* event to the *handleLogin()* function in contrary to Vue that passes the user as an argument. Then, Vue can update the loading value directly while react has to use the *setState()* function. In Vue we can call the dispatch directly –because of direct access to the store through *$store*- with only the user as an argument since it contains all the user state properties (*name*, *pass*). In React we call dispatch through the imported *Login Auth Action*, where we pass the name and password properties explicitly through the state object. The general pattern shows that Vue's reactivity allows for things being performed "easier", with fewer steps i.e. less code.

As for redirecting to *Profile* page, Vue uses programmatic navigation i.e. the router's instance methods, which enable accessing the router instance through $router. This allows for *this.$router.push* [75] to be called without having to import the router, which is the equivalent of *window.history.pushState* (of the *window.history* API) that is used in React's implementation.

In both the above cases (login and redirection) Vue's solution is again cleaner and less verbose.

Finally React performs the evaluation of loading state, which then acts as a conditional for the rendering of the spinner, outside of the render. On Vue, *v-show*, one of its directives, is used to achieve the same result, with the exact same (number of) steps i.e. lines of code. So this comes down to personal preference. It should though be noted, that the *if* statement used in React, is much more intuitive and closer to the native JS experience, and thus –presumably- to the developer's prior knowledge and familiarity.

- **Registration Page**
    - **React**

The implementation of the *Registration* Page is quite the same as the *Login* one. Thus in Listing 6.39 below only the differences/add-ons from *Login* page are shown, the rest of the code is exactly the same.
There are a few more details necessary for Form Validation:
username *(name)*: Is required, length: 3 -30 characters.
email: Is required, in email-format.
password *(pass)*: Is required, length: 8 - 40 characters.
To validate the email, we also use the validator's *isEmail()* function. The required function from react-validation is used as a pattern to form the corresponding validation-check functions *const email, vname, vpass* with their respective value restrictions. The new *onChange()*- functions are added to handle the new field inputs. According to the result *(SIGNUP_SUCCEEDED /SIGNUP_FAILED)* in the *Auth Actions Creator* to the *handleSignup*() function, the message is set to its value, passed down and eventually shown to the user.

```
 ...
import { isEmail } from "validator";
....
import { signup } from "../actions/auth";


....


const email = (value) => {
  if (!isEmail(value)) {
    return (<div className="alert alert-danger" role="alert">
This is not a valid email.
</div>);
  }
```

```
};
const vname = (value) => {
  if (value.length < 3 || value.length > 30) {
    return (<div className=”alert alert-danger” role=”alert”>
The username must be between 3 and 30 characters.
</div>);
  }
};
const vpass = (value) => {
  if (value.length < 8 || value.length > 40) {
    return (<div className=”alert alert-danger” role=”alert”>
The password must be between 8 and 40 characters.
</div>);
  }
};
class Signup extends Component {
  constructor(props) {
      super(props);
      this.handleSignup = this.handleSignup.bind(this);
      this.onChangeName = this.onChangeName.bind(this);
      this.onChangeEmail = this.onChangeEmail.bind(this);
      this.onChangePass = this.onChangePass.bind(this);
      this.state = {
        name: “”,
        email: “”,
        pass: “”,
        valid : false,
      };
    }

...

  onChangeEmail(e) {
      this.setState({
        email: e.target.value,
      });
    }

...
```

```
  handleSignup(e) {
      e.preventDefault();
      this.setState({
        valid : false,
      });

...

   if (this.checkBtn.context._errors.length === 0) {
        this.props.dispatch(signup(this.state.name,
this.state.email, this.state.pass))...render() {
            const {
              msg
            } = this.props;
            return (

...

<Form
  onSubmit={this.handleSignup}
  ref={ c =>
  {
  this.form = c;
  }}
  >
  {!this.state.valid  && (
  <div>
    <div className="form-group">
      ...
      <div className="form-group">
        <label htmlFor="email">Email</label>
        <Input
        type="text"
        className="form-control"
        name="email"
        value={this.state.email}
        onChange()={this.onChangeEmail}
        validations={[required, email]}
        />
      </div>
      <div className=”form-group”>
        <label htmlFor=”pass”>Password</label>
        <Input
        type=”pass”
        className=”form-control”
        name=”pass”
        value={this.state.pass}
        onChange()={this.onChangePass}
        validations={[required, vpass]}
        />
      </div>
      <div className=”form-group”>
        <button    className=”btn    btn-primary    btn-block”>Sign
Up</button>
      </div>
    </div>
    )}
    {msg  && (
    <div className=”form-group”>
      <div  className={  this.state.valid   ?  “alert  alert-valid”  :
“alert alert-danger” } role=”alert”>
      {msg }
    </div>
  </div>
  )}
  ....
</Form>
</div>
</div>
);
}
}
function mapStateToProps(state) {
const { msg  } = state.msg;
return {
msg
};
```

```
}
export default connect(mapStateToProps)(Signup);
```

Listing 6.39: React Registration Page: *signup.component.js*

- **Vue**

This page as seen in Listing 6.40 resembles the Login Page, except that for the Registration's form validation, the validation schema contains some extra details:

username *(name)*: Is required, length: 3-30 characters

email: Is required, in email-format, maximum length: 50 characters

password (*pass*): Is required, length: 8-40 characters

The user *loggedIn()* status is retrieved from *Vuex* store (auth.module.js) module-through authentication service (auth.js) which uses *Axios* for the HTTP call and then reponse to auth.module.js-in the *computed()* function and its value is checked at *mounted()* [35]. The user is forwarded to the Profile page if they have been logged in already. Else, on submitting the form, the *handleSignup()* function is called and, as with the Login page, the fields are checked with the validation schema and corresponding messages are shown. If the validation passes, *auth/signup Vuex* action is dispatched with the new values of the state variables set accordingly, and the success message is rendered as per the result of the *v-if* tag.

```
...
<template>
...
<Form @submit="handleSignup" :validation-schema="schema">
        <div v-if="!valid ">
          <div class="form-group">
            <label for="name">Username</label>
            <Field name="name" type="text" class="form-control" />
            <ErrorMessage name="name" class="error-feedback" />
          </div>
          <div class="form-group">
            <label for="email">Email</label>
            <Field name="email" type="email" class="form-control" />
            <ErrorMessage name="email" class="error-feedback" />
          </div>
          <div class="form-group">
            <label for="pass">Password</label>
            <Field name="pass" type="pass" class="form-control" />
            <ErrorMessage name="pass" class="error-feedback" />
          </div>
<div class="form-group">
            <button      class="btn      btn-primary      btn-block"
:disabled="loading">
              <span
                v-show="loading"
                class="spinner-border spinner-border-sm"
              ></span>
              Sign Up
            </button>
          </div>
        </div>
      </Form>
<template>

<script>
export default {
  name: "Signup",
  components: {
    Form,
    Field,
    ErrorMessage,
  },
data() {
    const schema = yup.object().shape({
      name: yup
        .string()
        .required("Username cannot be empty!")
        .min(3, " Username must be at least 3 characters!")
        .max(30, " Username cannot be longer than 30 characters!"),
      email: yup
        .string()
        .required("Email cannot be empty !")
        .email("Email is invalid!")
        .max(50, "Email cannot be longer than 50 characters!"),
      pass: yup
        .string()
        .required("Password cannot be empty!")
        .min(6, " Password must be at least 8 characters!")
        .max(40, " Password cannot be longer than 40 characters!"),
    });
return {
      valid : false,
      loading: false,
      msg : "",
      schema,
    };
  },
  computed: {
    loggedIn() {
      return this.$store.state.auth.status.loggedIn;
    },
  },
  mounted() {
    if (this.loggedIn) {
      this.$router.push("/profile");
    }
  },
  methods: {
    handleSignup(user) {
      this.msg = "";
      this.valid  = false;
      this.loading = true;

      this.$store.dispatch("auth/signup", user).then(
        (data) => {
          this.msg = data.msg;
          this.valid  = true;
          this.loading = false;
        },
        (err) => {
```

```
...
}
</script>
<style scoped>
...
</style>
```

Listing 6.40: Vue Registration page: *Signup.vue*

Registration Page Comparison

In Vue, as in Login Page but even more obvious here, since there are more fields, the validation is enabled one time at the <*Form*> component and includes all fields, while in React each corresponding validation has to be explicitly declared at each field.
Vue's and *VeeValidate's* convenience, deriving from their tendency to abstraction comes with the "side effect" of not being clear to a potential reviewer, as discussed in the Login Page comparison analysis, in contrary to React where, we can see that with JSX, we can utilize more "natural" constructs like if statements, functions, loops, and so on than with Vue code.

- **Profile Page**
    - **React**

At the user's Profile page the active *User* is retrieved from *localStorage* using the *Redux* store's application state (props). The data are originally retrieved at *auth-service*, which passes them as a response to the *auth actions creator* (*auth.js*).
According to the response, it checks if a logged-in user exists, and shows user information along with the token from *localStorage*, or redirects user to perform the login action in the opposite scenario. The source code is presented in Listing 6.41 below.

```
import React, { Component } from "react";
import { Redirect } from 'react-router-dom';
import { connect } from "react-redux";
class Profile extends Component {
render() {
  const { user: userLoggedIn } = this.props;
  if (!userLoggedIn) {
    return <Redirect to=”/login” />;
  }
  return (
  <div className=”container”>
     <header className=”jumbotron”>
        <h3>
           <strong>{userLoggedIn.name}</strong> Profile
        </h3>
     </header>
     <p>
      <strong>Token:</strong>
{userLoggedIn.accessToken.substring(0,   20)} ...{” “}

{userLoggedIn.accessToken.substr(userLoggedIn.accessToken.length  -
20)}
     </p>
     <p>
        <strong>Id:</strong> {userLoggedIn.id}
     </p>
     <p>
        <strong>Email:</strong> {userLoggedIn.email}
    </p>
     <strong>Authorities:</strong>
     <ul>
        {userLoggedIn.roles &&
        userLoggedIn.roles.map((role, index) =>
          <li key={index}>{role}</li>
        )}
     </ul>
  </div>
  );
 }
}
function mapStateToProps(state) {
  const { user } = state.auth;
  return {user,};
}
export default connect(mapStateToProps)(Profile);
```

Listing 6.41: React Profile page: *profile.component.js*

Vue

This page gets the current user from *Vuex* Store with the computed property *userLoggedIn*(). Then, when the component has been mounted at the DOM (at *mounted*() hook), it checks if the user exists i.e. is logged in, and shows user information: name, access token, id, email, authorities (user roles). It redirects to the Login page in the opposite case. Listing 6.42 shows the source code.

```
<template>
  <div class="container">
    <header class="jumbotron">
      <h3>
        <strong>{{userLoggedIn.name}}</strong> Profile
      </h3>
    </header>
    <p>
      <strong>Token:</strong>
      {{userLoggedIn.accessToken.substring(0,20)}}              ...
{{userLoggedIn.accessToken.substr(userLoggedIn.accessToken.length  -
20)}}
    </p>
    <p>
      <strong>Id:</strong>
      {{userLoggedIn.id}}
    </p>
    <p>
      <strong>Email:</strong>
      {{userLoggedIn.email}}
    </p>
    <strong>Authorities:</strong>
    <ul>
      <liv-for="role             in             userLoggedIn.roles"
:key="role">{{role}}</li>
    </ul>
  </div>
```

```
</template>

<script>
export default {
  name: 'Profile',
  computed: {
    userLoggedIn() {
      return this.$store.state.auth.user;
    }
  },
  mounted() {
    if (!this.userLoggedIn) {
      this.$router.push('/login');
    }
  }
};
</script>
```

Listing 6.42: Vue's Profile Page: *Profile.vue*

Profile Page Comparison

In React an extra component- *<Redirect />* from the *react-router-dom* library is used for redirecting the user the login, adding to boilerplate code.

React's way of defining the "Authorities" with the *map()* function i.e. creating a list of the roles is much more verbose than the simple *v-for* directive in Vue, which obviously wins in readability, ease of use and implementation. Yet another instance of a trade-off between simplicity and abstraction (Vue) versus the explicit, native feeling of JavaScript (es6) (React).

All in all, React uses more (lines of) code to achieve the same result.

Components for Accessing Resources

In this section the components for accessing public and authorized content are developed. They make use of the *UserService* to get data from the API, as described below:

-*HomePage*: A page that displays public content. This page may be viewed without logging in.

The following two pages contain private/protected information:

-*BoardUser* page: Accessible as logged-in user/admin. Fetches the data through *UserService.getBoardUser().*

-*BoardAdmin* page: Accessible as logged-in admin. Fetches the data though *UserService.getBoardAdmin().*

For React we create the components in the corresponding folder:

components/

home.component.js

board-user.component.js

board-admin.component.js

The same process for Vue:

components/

Home.vue

BoardUser.vue

BoardAdmin.vue

- **Home Page**
  - **React**

As demonstrated in Listing 6.43, after the state is initialized in the constructor and the component output is rendered to the DOM, React calls the *componentDidMount*() [76]. Inside it, the Data Service's (*user-service.js*) *getPublicInfo*() method is called, which retrieves the public content from the server, via a *Axios* HTTP call. The content state is set to the response (*res*) data or an error (*err*) message (*msg*) and the UI update is scheduled. Consequently the render method is triggered again and the new content is displayed on the page.

```
import React, { Component } from "react";
import UserService from "../services/user-service";
export default class Home extends Component {
  constructor(props) {
    super(props);
    this.state = {
      content: “”
    };
  }
  componentDidMount() {
    UserService.getPublicInfo().then(
      res => {
       this.setState({
         content: res.data
       });
      },
  err => {
    this.setState({
      content:
        (err.res && err.res.data) ||
         err.msg || err.toString()
        });
    }
  );
 }

 render() {
  return (
    <div className=”container”>
      <header className=”jumbotron”>
        <h3>{this.state.content}</h3>
      </header>
    </div>
   )
 }
}
```

Listing 6.43: React Home page: *home.component.js*

- **Vue**

After the Vue instance is created and setup and the component output is inserted into the DOM, the *mounted()* lifecycle hook is called. Inside it, the Data Service’s (*user-service.js*) *getPublicInfo()* method is called, which retrieves content from the server via *Axios* HTTP call. The content property is set to the response data or an error message depending on its value. The page's content is then re-rendered. The implementation of the Vue Home page is demonstrated in Listing 6.44 below.

```
<template>
  <div class="container">
    <header class="jumbotron">
      <h3> {{ content }} </h3>
    </header>
  </div>
</template>

<script>
import UserService from "../services/user-service";

export default {
  name: "Home",
  data() {
    return {
      content: "",
    };
  },
  mounted() {
    UserService.getPublicInfo().then(
      (res) => {
        this.content = res.data;
      },
      (err) => {
        this.content =
          (err.res &&
            err.res.data &&
            err.res.data.msg) ||
          err.msg ||
          err.toString();
      }
    );
  },
};
</script>
```

Listing 6.44: Vue Home Page: *Home.vue*

Home Page Comparison

Here, for Vue, the *mounted()* hook is used to call *UserService* for the page's content as soon as the component is mounted on the DOM, and the equivalent lifecycle method *componentDidMount()* is utilized to perform the same task in React, at the same point in the lifecycle. Both Vue's hook and React's method run as soon as the component's output render to the DOM has already happened for the first time.

For data-binding and rendering the variables or expressions Vue uses the *mustache*-like double braces syntax in the template (as some other JavaScript libraries do). React apps, as previously stated, do not employ templates and instead need the developer to design the DOM in JavaScript, usually using JSX. There the (state) variable (e.g. *this.state.content*) is used inside curly braces.

- **Role Based Pages**

In role-based pages we are requesting authorized resources. This demands that, along with the corresponding *Axios* HTTP call to the API, the *userAuth()* function is also used to confirm the presence of a logged-in user, as well as to retrieve user state and token as demonstrated previously the Data Service implementation (Listing 6.27).

- **React**

Bellow in Listing 6.45 are shown only parts of the User Page that are different from the Home page, while the Admin Page is similar and thus is omitted.

```
import ...
export default class BoardUser extends Component {
  constructor(props) {
...
  componentDidMount() {
    UserService.getBoardUser().then(
      res => {
 ...      });
  },
  err => {
  ...
  });
        }
      );
  }
  render() {
...
  }
```

Listing 6.45: React User page component: *board-user.component.js*

- **Vue**

The same parts- as in React- of the Vue User Page, are shown in Listing 6.46.

```
<template>
  ...
</template>

<script>

export default {
  name: "User",
  data() {
    return {
      content: "",
    };
  },
  mounted() {
    UserService.getBoardUser().then(
      (res) => {
     ...
}
```

```
</script>
```

Listing 6.46. Vue User page: *BoardUser.vue*

Routing, Navbar, & The complete App Component

The App components, App.js and App.vue, are the top level components acting as containers for the rest of ther components of the respective applications. This means the rest of the components (pages) have to be imported here, so that they can be actually rendered and shown on the screen.

In this section we will define the routing and add the dynamic navigation bar in the *App* component. The *navbar* changes dynamically based on the current User's login status and roles.

Below is an overview of which links are shown in the *navbar* under which conditions:

- *Home*: Present at all times.
- *Login & Sign Up*: Present if the user is not yet signed in.
- *User*: Present if there is a logged-in user.
- Admin Board: Present only if *ROLE_ADMIN* is included in the user *roles.*

- **React**

Create React Router History

The terms “*history*” and "*history object*" pertain to React Router’s *history package*, while “*browser history*” is a particular implementation for the DOM, which can be used in web browsers that are compatible with the *HTML5 history* API.

The custom *history* object used by the React Router for navigation is shown in Listing 6.47.

```
import { createBrowserHistory } from “history”;
export const history = createBrowserHistory();
```

Listing 6.47: helpers/history.js

The actual routing for React is implemented -with the help of the custom history object- in the following *App* component in Listing 6.48.

The App Component with navbar

The *navbar* changes dynamically based on the current User's roles as obtained from the *Redux Store* state.

All the components of the application are imported here, in the main *App* component. Out of them, the Role-based components, along with the Profile page one, are lazy loaded, through the *React.lazy()* function and the dynamic import of the aforementioned components. The dynamic import is used for *Webpack* to -automatically- perform the code-splitting necessary for the lazy-loading. The *React.lazy* function allows a dynamic import to be rendered as a regular component. Components that are lazy-loaded are wrapped with a *Suspense* component, that allows the rendering of some kind of fallback content (e.g. a loading indicator) to be shown until the main (lazy) component is fully loaded [77].

At *componentDidMount()* hook, *user* gets state from *props* and if a user value is present in the application state, *userLoggedIn* and *showBoardAdmin* update their state accordingly (true/false), else the initial state is used as the condition for the routing.

The *<Router>* component is used to synchronize the custom history with *Redux*. Here, it is the one responsible for rendering the *App* component, while the *<Route>* component defines which component will be rendered for each route or path. The *Switch* statement makes sure only one component will be rendered at a time. The history object [78] contains the session history and is used for navigation.

The *<Link>* component is handling all the internal links of the *navbar* making them accessible. Both *<Router>* and *<Link>,* although they are components, are not rendered on the page. The result is that the links with the paths to public content, React AUTH (*/*) and Home (*/home*) are always present on the page. The role-based Admin Board (*/admin*) and User (*/user*) links are (not) shown according to the user state, as are the links for Authentication – Login (*/login*), Sign Up (*/signup*) and Logout (redirects to */login*). The Profile page (*/profile*), which renders as the *userLoggedIn* name, also requires a logged-in user for it to show on the *navbar*.

A message action is dispatched when the user clicks on different links (at url-change), which uses react router's location[9] [79] to detect route changes, and eventually update the message state (clears the message). The implementation is presented in Listing 6.48 below.

```
import React, { Component } from “react”;
import { connect } from “react-redux”;import React, { Suspense, lazy
} from 'react';
import { Router, Route, Switch, Link } from “react-router-dom”;
import “bootstrap/dist/css/bootstrap.min.css”;
import “./App.css”;
import Login from “./components/login.component”;
import Signup from “./components/signup.component”;
import Home from “./components/home.component”;
//lazy-loaded
constProfile=React.lazy=> import(“./components/profile.component”);
const  BoardUser  =  React.lazy(()=>  import  (“./components/board-
user.component”);
const     BoardAdmin=React.lazy(()     =>     (“./components/board-
admin.component”);
import { logout } from “./actions/auth”;
import { clearMsg } from “./actions/message”;
import { history } from ‘./helpers/history’;

class App extends Component {
  constructor(props) {
   super(props);
   this.logOut = this.logOut.bind(this);
    this.state = {
      showBoardAdmin: false,
      userLoggedIn: undefined,
    };

    history.listen((location) => {
      props.dispatch(clearMsg());
    });
```

[9] Locations denote where the app is currently, as well as its past and future state.

```
}

componentDidMount() {
  const user = this.props.user;
  if (user) {
    this.setState({
      userLoggedIn: user,
      showBoardAdmin: user.roles.includes("ROLE_ADMIN"),
    });
  }
}

logOut() {
  this.props.dispatch(logout());
}

render() {
  const { userLoggedIn, showBoardAdmin } = this.state;
   return (
    <Router history={history}>
<Suspense fallback={
 <div>Loading...</div>
 }>
 <div>
    <nav className="navbar navbar-expand navbar-dark bg-dark">
       <Link to={"/"} className="navbar-brand">
       React AUTH
       </Link>
       <div className="navbar-nav mr-auto">
          <li className="nav-item">
             <Link to={"/home"} className="nav-link">
             Home
             </Link>
          </li>
          {showBoardAdmin && (
          <li className="nav-item">
             <Link to={"/admin"} className="nav-link">
             Admin
             </Link>
```

```
            </li>
            )}
            {userLoggedIn && (
            <li className=”nav-item”>
               <Link to={”/user”} className=”nav-link”>
               User
               </Link>
            </li>
            )}
         </div>
         {userLoggedIn ? (
         <div className=”navbar-nav ml-auto”>
            <li className=”nav-item”>
               <Link to={”/profile”} className=”nav-link”>
               {userLoggedIn.name}
               </Link>
            </li>
            <li className=”nav-item”>
               <a         href=”/login”         className=”nav-link”
onClick={this.logOut}>
               LogOut
               </a>
            </li>
         </div>
         ) : (
         <div className=”navbar-nav ml-auto”>
            <li className=”nav-item”>
               <Link to={”/login”} className=”nav-link”>
               Login
               </Link>
            </li>
            <li className=»nav-item»>
               <Link to={”/signup”} className=”nav-link”>
               Sign Up
               </Link>
            </li>
         </div>
         )}
      </nav>
      <div className="container mt-3">
         <Switch>
            <Route exact path={["/", "/home"]} component={Home} />
            <Route exact path="/login" component={Login} />
            <Route exact path="/signup" component={Signup} />
            <Route exact path="/profile" component={Profile} />
            <Route path="/user" component={BoardUser} />
            <Route path="/admin" component={BoardAdmin} />
         </Switch>
      </div>
   </div>
   </Suspense>
</Router>
);
}
}
function mapStateToProps(state) {
const { user } = state.auth;
return {
user,
};
}
export default connect(mapStateToProps)(App);
```

Listing 6.48: React App component: *App.js*

- **Vue**

Routing

For Vue, the routes for the application are defined in src/router.js (Listing 6.49).

First, from *vue-router*, the *createRouter()* function, which genarates a Router instance that the Vue app can use, and the *createWebHistory()* function, that creates an HTML5 *history*[10], are imported as shown in Listing 6.49 below. Following, all of the app's components are imported, with the difference that the role-based and the profile pages will be lazy-loaded.

[10] API that provides access to the browser navigation history via JS.

Lazy-loading allows for the selection of the parts of JavaScript we need for the first page load, thus reducing the page load time [80]. *Code splitting* is also used in conjunction, and is the process of splitting the app into lazily loaded chunks [81]. The above are achieved through dynamic imports and the use of a function to call them on the component.

Then, we define the routes as an objects array. The application's routes are represented by the array, while its items are the route objects. The path property represents the *url* (for example, the "/" at the first object indicates the base *URL*), the component property defines which component will be rendered when this url path is visited, and the name property indicates the route's name.

At the end of the file, the router is created with *createRouter()* method which takes the routes object as its parameter.

```
import { createWebHistory, createRouter } from "vue-router";
import Home from "./components/Home.vue";
import Login from "./components/Login.vue";
import Signup from "./components/Signup.vue";
// lazy-loaded
const Profile = () => import("./components/Profile.vue")
const BoardAdmin = () => import("./components/BoardAdmin.vue")
const BoardUser = () => import("./components/BoardUser.vue")

const routes = [
  {
    path: "/",
    name: "home",
    component: Home,
  },

  {

    path: "/signup",

    component: Signup,

  },

  {
    path: "/login",

    component: Login,

  },
  {
    path: "/home",
    component: Home,
  },

  {
    path: "/profile",
    name: "profile",
    // lazy-loaded
    component: Profile,
  },

  {

    path: "/user",

    name: "user",

     // lazy-loaded

    component: BoardUser,

  },

  {
    path: "/admin",
    name: "admin",
    // lazy-loaded
    component: BoardAdmin,
  },

];

const router = createRouter({
  history: createWebHistory(),
  routes,εε
});

export default router;
```

Listing 6.49: Routes for Vue-Router: *router.js*

The complete App component with navbar

The *navbar* changes dynamically according to the current User roles as determined in the *Vuex* Store state.

In Vue, the *userLoggedIn()* and *showBoardAdmin()* functions are calculated as computed properties that access the user state from *Vuex* store and determine if there is a logged-in user (user state), and if the user has Admin Role, respectively. Based on these inputs the Vue template utilizes the *v-if* directive to determine which links will show on the *navbar* and which corresponding routes are available to the user, while the <*Router-link*> component is used for enabling the navigation in the app. As for rendering the appropriate component, according to the Vue Router documentation: "The <*router-view*> component is a functional component that renders the matched component for the given path" [82]. It is essentially the main <div> that contains all the components, and it returns the component that matches the current path. The overall source for the *App* component is shown in Listing 6.50 below.

```
<template>
  <div id="app">
    <nav class="navbar navbar-expand navbar-dark bg-dark">
      <a href="/" class="navbar-brand">VueJS AUTH </a>
      <div class="navbar-nav mr-auto">
        <li class="nav-item">
          <router-link to="/home" class="nav-link">
             Home
          </router-link>
        </li>
        <li v-if="showBoardAdmin" class="nav-item">
          <router-link     to="/admin"       class="nav-link">Admin
Board</router-  link>
```

```
        </li>

        <li class="nav-item">
          <router-link  v-if="userLoggedIn"  to="/user"  class="nav-
link">User</router-link>
        </li>
      </div>

      <div v-if="!userLoggedIn" class="navbar-nav ml-auto">
        <li class="nav-item">
          <router-link to="/signup" class="nav-link">
             Sign Up
          </router-link>
        </li>
        <li class="nav-item">
          <router-link to="/login" class="nav-link">
             Login
          </router-link>
        </li>
      </div>

      <div v-if="userLoggedIn" class="navbar-nav ml-auto">
        <li class="nav-item">
          <router-link to="/profile" class="nav-link">
                       {{ userLoggedIn.name }}
          </router-link>
        </li>
        <li class="nav-item">
          <a class="nav-link" @click.prevent="logOut">
             LogOut
          </a>
        </li>
      </div>
    </nav>

    <div class="container">
      <router-view />
    </div>
  </div>
```

```
</template>

<script>
export default {
  computed: {
    userLoggedIn() {
      return this.$store.state.auth.user;
    },
    showBoardAdmin() {
      if (this.userLoggedIn && this.userLoggedIn['roles']) {
        return this.userLoggedIn['roles'].includes('ROLE_ADMIN');
      }

      return false;
    },

      return false;
    }
  },
  methods: {
    logOut() {
      this.$store.dispatch('auth/logout');
      this.$router.push('/login');
    }
  }
};
</script>
```

Listing 6.50: Vue *App* component: *App.vue*

App component, Routing & navbar Comparison

React-Router v4 requires the declaration of the routes right in the HTML (unlike previous versions of *React-Router*), in a switch statement. Vue on the other hand uses a separate -dedicated- file (*router.js*), where the routes are declared as objects. This is not mandatory, since the routes object could be declared in the <*script*> part of the *App.vue SFC*, and used as such, so the developer has the freedom to choose and act accordingly. A separate file is obviously a neater way to do it, keeping the html cleaner and less crowded, but it adds the

overhead of a new file which comes with an extra import, plus Vue's routes declaration is a bit more verbose. There is a trade-off between lines of code and readability here. Other than that both frameworks use corresponding components for the same functions: *Vue Router* uses <*Router-link*> for the navigation, and <*router-view*> for rendering the component of the selected path, while React Router has <*Link*> and <*Route*> for the respective functionalities. One thing is that React also uses the <Router> component for integration with Redux. Vue has no need of similar solution since its state management solution is -de facto- integrated and tightly coupled to begin with.
Finally, both make extensive use of the HTML5 *History* API.
The *lazy loading* and *code-splitting* features require a bundler, like *Webpack*, that both frameworks enable with the app's creation and no further tweaking is required. Lazy loading components also provided right out of the box in both frameworks and they both use *Webpack* dynamic imports for the code-splitting. One difference is that React requires its *React.lazy()* function for the rendering. The <*Suspense*> component used in React as a fallback while waiting for the component to load, is currently a new feature in Vue, still in experimental stage, and has the same exact functionality, although it will only be used with async components as made clear in the documentation [83]. For *lazy loading* routes *Vue Router* again provides the feature right away as secondary routes are already configured to be lazily loaded by default (when using the *Vue CLI*). *React Router* as well has no problem in integrating the component's lazy loading even without the Vue's convenience.
As for rendering the matching component, Vue's *v-if* directive was again a cleaner and more intuitive solution for matching and showing data than the state variable acting as conditional outside of the render in React, which was not something difficult to conceive/understand and implement, just more explicit. Some, of course would argue that the JSX way resembles more to what they are already accustomed to from native JS, which is also true, so in the end it is a matter of preference.
As seen earlier, React does not have a concept similar to Vue's computed properties, so the *userLoggedIn*() and *showBoardAdmin*() values are "computed" inside the *componentDidMount*() method which technically means that in React this happens later (after the initial render) than in Vue(before the initial render), which does not affect the functionality in any way.

- **React**

CSS style for React Components

The CSS for the application’s styling is located in *src/App.*CSS which is imported in the *App* component (*App.js*)

Port Configuration for the Web API

As CORS is utilized in most HTTP servers, allowing them to specify any origins i.e domains, ports e.t.c, from which resourses can be loaded, a port must likewise be configure a for this application.

A *.env* file is created in the project’s root folder, which contains the port value as such: *PORT=8081*

Now the app is set to run at port 8081.

- **Running the App**

At the *CLI*, in the folder where the app is created, running *npm start* creates a frontend build pipeline, which can be used it with any backend, in our case Node.js. This essentially "runs" the App which is now ready to be viewed.

At the server side, while in the folder where the app is created, the CLI command *node server.js* starts the development server at *localhost: 8081* which now serves the React front-end at *localhost: 808*,* where * is set according to the configuration and the available ports on the network, so it may vary.

- **Vue**

CSS style for Vue Components

The styling of the Vue application takes place in the corresponding <*style*> section of each component, according to the *SFC* implementation.

Port Configuration for the Web API

As in React's case above, the port for the Vue app needs to be configured as well.

A *vue.config.js* file is created in the project's root folder with the following lines of code, as shown in Listing 6.51:

```
module.exports = {
  devServer: {
    port: 8081
  }
}
```

Listing 6.51: Vue configuration file: *vue.config.js*

The app will now be running on port 8081.

Running the App

At the CLI, in the folder where the app is created, running *npm run serve* runs the app according to the *package.json* configuration, which in this case executes the default preset "*vue-cli-service serve"* script*.*

The application is now served at *localhost*: *808**, and will be automatically connected to the backend-API running at *localhost: 8081.*

Both React & Vue applications can be served from the API - on a different port- at the same time.

Comparison- Final Steps

For the styling of the application, Vue follows its own *SFC* logic while for React the traditional-default approach was followed here, while there are other solutions as mentioned in 6.2.1.

As for the port setup, React's port configuration is obviously simpler with just a value assignment.j

Chapter 7 Conclusion

React and Vue, two of the most successful JavaScript frameworks in recent times, were investigated in this thesis. This chapter concludes with a summary of the findings, a discussion on the results, and the study's general conclusion and suggestions, as well as some final overall comments on the subject.

7.1 Findings & Results

The summarized results of the comparison analysis conducted in the previous corresponding sections are presented here.

Components: React seems to provide more syntax freedom/choices in terms of creating components as it provides two distinct ways to do it. This contributes to the freedom/flexibility which react is "renounced" for, which is of course always accompanied with a certain degree of “responsibility”, so it depends on the point of view whether it is a positive or a negative factor.

Reacts' JSX way of handling html and writing components has been a point of disagreement among developers since day one. One the one hand it gives more power over the html template on the other it adds a gnostic overhead to the learning volume resulting to a steeper learning curve. Both arguments are valid.

Vue's *Single File Components* are very developer friendly, intuitive and easy to use, because of their arrangements of JavaScript, html and CSS adding for an easier and quicker adoption of the framework.

Data Binding: Essentially both frameworks provide the choice of one or two way data binding. React is significantly favorable to *one way data binding*, to the point that the general sense for many years was that two-way data binding was not possible in React. Vue on the other hand is well known and even preferred for this exact feature of making two way binding easy and effective thus making the code more readable and simpler while losing in performance and manageability. This is perhaps why, in version 3 of Vue, the team proposes one way binding as a better solution.

Routing: This is where the main -de facto- difference of the two "contestants" makes its appearance. React, being a library itself, does not come with a native routing solution thus an external library is required, and available. Vue, as a complete framework, includes an official router in its default installation. Both solutions support dynamic routing.

Scripting and Rendering: Both frameworks utilize the creation of Virtual Dom with nuances that become insignificant – performance-wise – in most cases, with the exception of rendering large data structures in which case Vue may be more optimal, because of the way it specifies the re-rendering of the components.

Platform Support: No significant differences in terms of support for the browsers are spotted. Each framework is fully supported in all modern browsers, and requires *polyfills* for legacy ones. However it should be notice that React, being a library, introduces another layer of required compatibility for the necessary external libraries, which is indeed in most cases provided.

External Libraries & Modules: React has a huge ecosystem of external solutions for each and every possible use case. Vue does not fall short on this regard either while not on the same level as react. Both ecosystems are vibrant and being actively enriched. React's nature as a library, means it uses external libraries specifically and independently developed for their own specific case. This may be interpreted as being more optimized in comparison to Vue's default native solutions for the same functionalities which of course comes with its own inherent quality of a tightly coupled and robust solution specifically developed for and applied to Vue. One might have to choose a "specificity side" to declare winner here.

Localization: Both frameworks rely on external localization libraries.

Documentation & Community Support: Vue has arguably one of –if not the best documentation in the JSf ecosystem, while React dos not fall short on this regard either, even if not as impressively well documented and thorough as Vue's. React has the largest community in the ecosystem while Vue's community is also vibrant and dedicated, as made obvious from the number of *Github* stars presented in 6.2.10.

Learning Curve & Developer Experience: Vue is considered the most beginner friendly of all the major JSfs. The level of abstraction it provides as well as the structure of its components make it intuitive and relatively easy to adopt for someone coming from a native (*vanilla*) JavaScript background. React’s steeper learning curve is mostly attributed to the use of JSX which is not mandatory in any case, but does play well with React components. In terms of development experience both frameworks check all the boxes, with Vue going the extra mile with its Graphical User Interface (GUI) that accompanies the *vue-cli*.

Popularity and future: Both of the frameworks' popularity metrics promise a promising future. Both are being actively developed and regularly updated. A factor that could play its part and make the difference here is the backing of React by Facebook. To this point though this important fact does not seem to play as an important role as one would imagine because of the intrinsic qualities of the open-source ecosystem.

7.2 Summary

As hinted from the introduction of this chapter there is no announcement of a clear winner in this study. Both frameworks perform well in the examined programming functionalities with one having the edge over the other in some but again in specific cases that cannot therefore be generalized. The same applies to the rest qualitative metrics that arguably play a key part in deciding whether to adopt a framework or not. What React lacks in terms of completed and default solutions it gains in modularity and configuration capabilities. This, which is in the author's opinion their biggest difference and point for debate, is not really a difference of React and Vue, per se, meaning that it is intrinsic of their different nature as a library and a framework respectively. That being said, they still serve the same purpose which makes the comparison between them justifiable and valid.

The only point where there seems to be a clear winner is in native applications development where React would be the best solution.

So how would one decide?

The answer of the author would be, according to the use case. Vue for example is suitable for rapid prototyping and progressive introduction to a project due its progressive nature. React's stong point is its flexibility to add packages on the go, and not be bound to specific solutions. There are numerous other factors that could potentially weight in the decision making e.g. the developers' knowledge of ECMAscript 2015 (es6) introduced features, that are more prominent in React than Vue. Other-completely different and non-technical factors- such as social factors e.g. the job market, local adoption (Vue is largely used in China and the USA but not in Europe) etc. play a critical role in decision for adoption and could be further studied in future work. In other words, as vague as it might seem in an academic paper, each has its own place and time, and at the same time - and on the contrary one might add- more often than not it boils down to personal preference which of course has as a prerequisite the ("tedious") task of dedicating at least a short period of time to experimenting with both.

7.3 Future Work & Discussion

There is only so much that can fit to a paper of this magnitude thus this study has been far from exhaustive. Numerous technical -and non- points for comparison lie in the heart of the JavaScript frameworks' ecosystem and in that of the author's. Specific features that were left out of this study -for various reasons, varying from the research methodologies' findings, to the authors own technical inadequacies or preferences- include: Server Side Rendering capabilities, hybrid applications and Progressive Web Apps development in React & Vue, Vue & React as front-end frameworks for Decentralized Apps (Dapps), as well as further enriching the evaluation model by adding more frameworks, such as the established Angular and the upper-coming and promising Svelte. The environment in which the applications where developed could also further include more browsers and different set of configurations, integration with different backend technologies and databases The deployment of the applications to various serves and environments, integration of CI/CD pipelines and A/B testing should perhaps be the first step to enriching, solidifying or challenging the validity of the findings for comparison, since these are the necessary steps and elements that eventually bring a project built with a front-end JavaScript framework to life, and the field were the advantages and the limitations converge and prepare to meet the true cause of their existence: The user.

The JavaScript ecosystem, particularly the one of its numerous frameworks, is fertile ground for research and development and will continue to be so due to its main quality of powering the Web. Their relationship and interaction is such that creates a positive feedback loop that will fuel new exciting capabilities in the years to come, in which academia will surely play an important role, that of providing objective reasoning and grounding for the numerous projects and technologies that eventually-to a greater or lesser extent- affect our lives, directly and/or indirectly, in a multitude of ways.

Bibliography

[1] Broadbandsearch, "Mobile vs Desktop Internet Usage (Latest 2022 Data)," Broadbandsearch.net, 2022. [Online]. Available: https://www.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics. [Accessed 7 January 2022].

[2] M. W. Docs, "Introduction to client-side frameworks," MDN Web Docs, [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction. [Accessed 12 August 2021].

[3] M. W. Docs, "SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms," Mozilla and individual contributors, 2021. [Online]. Available: https://web.archive.org/web/20210616121411/https:. [Accessed 15 9 2021].

[4] C. S. P. T. Gizas A., "Comparative evaluation of JavaScript frameworks," in *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web Companion*, 2012.

[5] P. R. a. B. L. a. B. Zorn, "Conference on Web Applications," in *JSMeter: Comparing the behavior of JavaScript benchmanrks with real web applications*, 2010.

[6] A. &. G. D. &. A. P. Pano, "What leads developers towards the choice of a JavaScript framework?," 2016.

[7] R.W.Saaty, "The analytic hierarchy process—what it is and how it is used," *Mathematical Modelling,* vol. 9, no. 3-5, 1987.

[8] &. B. M. S. Oney, "FireCrystal: Understanding interactive behaviors in dynamic web pages," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2009.

[9] C. L. Mariano, "Benchmarking JavaScript Frameworks (Masters dissertation)," Dublin, Ireland, 2017.

[10] I. Kantor, "javascript.info," 13 June 2021. [Online]. Available: https://javascript.info/intro. [Accessed 1 9 2021].

[11] Techterms, "techterms," 7 March 2013. [Online]. Available: https://techterms.com/definition/framework. [Accessed 3 9 2021].

[12] React, "Introducing JSX," Facebook Inc, 2021. [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed 3 9 2021].

[13] Vue3, "Render Functions," Evan You, [Online]. Available: https://v3.vuejs.org/guide/render-function.html#jsx. [Accessed 14 August 2021].

[14] D. A. P. Graziotin, "Making sense out of a jungle of JavaScript frameworks–towards a practitioner-friendly comparative analysis,," in *International Conference on Product Focused Software Process Improvement*.

[15] A. R. Heale, 2015.

[16] Vue, "Template Syntax," [Online]. Available: https://vuejs.org/v2/guide/syntax.html.

[17] C. Mocenni, "The Analytic Hierarchy Process".

[18] S. Fluin, "Why Developers and Companies Choose Angular," medium.com, 25 December 2017. [Online]. Available: https://medium.com/angular-japan-user-group/why-developers-and-companies-choose-angular-4c9ba6098e1c. [Accessed 5 9 2021].

[19] M. W. Docs, "An overview of HTTP," Mozilla and individual contributors, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. [Accessed 14 9 2021].

[20] Tutorialspoint, "Node.js - Introduction," [Online]. Available: https://www.tutorialspoint.com/nodejs/nodejs_introduction. [Accessed 7 9 2021].

[21] MongoDB, "MongoDB The application data platform," MongoDB, Inc., 2021. [Online]. Available: https://www.mongodb.com/. [Accessed 29 8 2021].

[22] a. O. R. J. Duvander, Interviewee, *“What affects the choice of a JavaScript framework: Interview with developers” ,*. [Interview]. 2019.

[23] Vue, "What-is-Vue-js," Evan You, [Online]. Available: https://vuejs.org/v2/guide/#What-is-Vue-js. [Accessed 5 9 2021].

[24] A. Amaechi, "Your guide to reactivity in Vue.js," Evan You, 17 December 2020. [Online]. Available: https://blog.logrocket.com/your-guide-to-reactivity-in-vue-js/. [Accessed 27 August 2021].

[25] Vue3, "Mobile," Evan You, 2021. [Online]. Available: https://v3.vuejs.org/guide/mobile.html#mobile.

[26] E. You, "github," Github, Inc, [Online]. Available: https://github.com/vuejs/vue/releases/tag/0.6.0.

[27] D. Deutsch, "Understanding MVC Architecture with React," 2017. [Online]. Available: https://medium.com/of-all-things-tech-progress/understanding-mvc-architecture-with-react-6cd38e91fefd.

[28] GeeksforGeeks, "React.js (Introduction and Working)," 30 7 2021. [Online]. Available: https://www.geeksforgeeks.org/react-js-introduction-working/.

[29] G. S. Padam, "tutorialslink.com," 9 June 2020. [Online]. Available: https://tutorialslink.com/Articles/What-is-React-Introduction-to-React/1529.

[30] React, "Rendering Elements," Facebook Inc., 2021. [Online]. Available: https://reactjs.org/docs/rendering-elements.html. [Accessed 26 August 2021].

[31] React, "Introducing JSX," Facebook Inc., [Online]. Available: https://reactjs.org/docs/introducing-jsx.html. [Accessed 25 August 2021].

[32] React, "The Component Lifecycle," [Online]. Available: https://reactjs.org/docs/react-component.html#the-component-lifecycle.

[33] Vue, "Composing with Components," Evan You, [Online]. Available: https://vuejs.org/v2/guide/#Composing-with-Components.

[34] Vue, "Instance Lifecycle Hooks," [Online]. Available: https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks.

[35] Vue3, "Lifecycle Hooks," [Online]. Available: https://v3.vuejs.org/api/options-lifecycle-hooks.html#mounted.
[Accessed 1 September 2021].

[36] Meks, "Inverse Data Flow in React," 11 November 2020. [Online]. Available: https://dev.to/mmcclure11/inverse-data-flow-in-react-mg7.

[37] React, "Two-way Binding Helpers," 2021. [Online]. Available: https://reactjs.org/docs/two-way-binding-helpers.html. [Accessed 13 9 2021].

[38] H. a. t. e. d. o. t.-w. d. binding?, "What are the exact demerits of two-way data binding?," 2015. [Online]. Available:

https://hashnode.com/post/what-are-theexact-demerits-of-two-way-data-binding-ciibz8fnq01f8j3xthmjjs6di . [Accessed September 2021].

[39] V. Router, "Vue Router," Evan You, Eduardo San Martin Morote, 2021. [Online]. [Accessed September 2021].

[40] B. Krajka, "The difference between Vrtual DOM and DOM," 12 Oktober 2015. [Online]. Available: https://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/. [Accessed 29 August 2021].

[41] Vue, "Comparison with Other Frameworks," [Online]. Available: https://vuejs.org/v2/guide/comparison.html
. [Accessed 4 September 2021].

[42] Stefalda, "react-localization," [Online]. Available: https://github.com/stefalda/react-localization. [Accessed 9 September 2021].

[43] React, "Hello World," Facebook, Inc, 2021. [Online]. Available: https://reactjs.org/docs/hello-world.html. [Accessed 27 August 2021].

[44] React, "Tutorial: Intro to React," 2021. [Online]. Available: https://reactjs.org/tutorial/tutorial.html. [Accessed 27 August 2021].

[45] Vue, "Introduction," [Online]. Available: https://vuejs.org/v2/guide/. [Accessed 28 August 2021].

[46] Hotframeworks, 2021. [Online]. Available: https://hotframeworks.com/. [Accessed 12 September 2021].

[47] S. o. JS, "Demographics," 2019. [Online]. Available: https://2019.stateofjs.com/demographics/. [Accessed 22 August 2021].

[48] S. Daityari, "Angular vs React vs Vue: Which Framework to Choose in 2021," 15 March 2021. [Online]. Available: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/. [Accessed 8 September 2021].

[49] Madewithvuejs, "Vue 3 – A roundup of infos about the new version of Vue.js," 28 April 2021. [Online]. Available: https://madewithvuejs.com/blog/vue-3-roundup. [Accessed 28 August 2021].

[50] Vue3, "Single File Components," Evan You, [Online]. Available: https://v3.vuejs.org/guide/single-file-component.html#single-file-components. [Accessed 25 August 2021].

[51] React, "Thinking in React," Facebook Inc., 2021. [Online]. Available: https://reactjs.org/docs/thinking-in-react.html. [Accessed 1 September 2021].

[52] Vue, "Components Basics," Evan You, [Online]. Available: https://vuejs.org/v2/guide/components.html#Listening-to-Child-Components-Events. [Accessed 23 August 2021].

[53] React, "Create a New React App," Facebook Inc., 2021. [Online]. Available: https://reactjs.org/docs/create-a-new-react-app.html#create-react-app. [Accessed 22 August 2021].

[54] V. CLI, "Creating a Project," Evan You, [Online]. Available: https://cli.vuejs.org/guide/creating-a-project.html#vue-create. [Accessed 23 August 2021].

[55] Vue3, "Installation," 23 August 2021. [Online]. Available: https://v3.vuejs.org/guide/installation.html#installation. [Accessed 24 Agust 2021].

[56] Redux, "Getting Started with Redux," Dan Abramov and the Redux documentation authors, [Online]. Available: Redux is a predictable state container for JavaScript apps.. [Accessed 3 September 2021].

[57] J. Potter, "mobx vs redux," [Online]. [Accessed 9 September 2021].

[58] Redux, "Redux Fundamentals, Part 3: State, Actions, and Reducers," Dan Abramov and the Redux documentation authors, [Online]. Available: https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers#splitting-reducers. [Accessed 4 September 2021].

[59] Vuex, "What is Vuex?," [Online]. Available: https://vuex.vuejs.org/#what-is-vuex. [Accessed 4 September 2021].

[60] Vuex, "What is a "State Management Pattern"?," [Online]. Available: https://vuex.vuejs.org/#what-is-a-state-management-pattern. [Accessed 5 September 2021].

[61] Alexandros Gougousis, "Design patterns in action: Decorator and Redux Middlewares," 15 August 2018. [Online]. Available: https://blog.gougousis.net/design-patterns-in-action-decorator-and-redux-middlewares/. [Accessed 5 September 2021].

[62] E. John, "Pinia vs. Vuex: Is Pinia a good replacement for Vuex?," LogRocket, Inc, 6 July 2021. [Online]. Available: https://blog.logrocket.com/pinia-vs-vuex/. [Accessed 26 August 2021].

[63] Openbase, "openbase.com," Openbase, Inc., [Online]. Available: https://openbase.com/js/vuex/alternatives. [Accessed 27 August 2021].

[64] Vue3, "Single File Components," Evan You, [Online]. Available: https://v3.vuejs.org/guide/single-file-component.html#why-sfc. [Accessed 25 August 2021].

[65] Michal, "Vuex vs. Redux - similarities and differences," 8 June 2021. [Online]. Available: https://www.merixstudio.com/blog/vuex-vs-redux/. [Accessed 15 September 2021].

[66] Newbedev, "ReactJS/Redux - Pure vs Impure Javascript functions?," newbedev, [Online]. Available: https://newbedev.com/reactjs-redux-pure-vs-impure-javascript-functions. [Accessed 7 September 2021].

[67] React-validation, "npmjs.com," [Online]. Available: https://www.npmjs.com/package/react-validation. [Accessed 16 September 2021].

[68] React, "Forms," Facebook Inc., 2021. [Online]. Available: https://reactjs.org/docs/forms.html#controlled-components. [Accessed 16 September 2021].

[69] React, "React.Component," Facebook Inc., 2021. [Online]. Available: https://reactjs.org/docs/react-component.html#setstate. [Accessed 14 September 2021].

[70] React-Redux, "connect()," Dan Abramov and the Redux documentation authors.. [Online]. [Accessed 3 September 2021].

[71] Vee-validate, "ErrorMessage," [Online]. Available: https://vee-validate.logaretm.com/v4/v4/api/error-message#errormessage. [Accessed 9 September 2021].

[72] Vuex, "State," [Online]. Available: https://vuex.vuejs.org/guide/state.html#single-state-tree. [Accessed 28 August 2021].

[73] Vue3, "Lifecycle hooks," 2 March 2021. [Online]. Available: https://v3.vuejs.org/api/options-lifecycle-hooks.html#created. [Accessed 16 Septenber 2021].

[74] Vee-validate, "Form Values," Abdelrahman Awad, [Online]. Available: https://vee-validate.logaretm.com/v4/guide/components/handling-forms#form-values. [Accessed 3 September 2021].

[75] V. Router, "Programmatic Navigation," Evan You, Eduardo San Martin Morote, [Online]. Available: https://router.vuejs.org/guide/essentials/navigation.html#programmatic-navigation. [Accessed 4 September 2021].

[76] React, "React.Component," Facebook Inc., 2021. [Online]. Available: https://reactjs.org/docs/react-component.html#componentdidmount. [Accessed 2 September 2021].

[77] React, "Code-Splitting," Facebook Inc., 2021. [Online]. Available: he lazy component should then be rendered inside a Suspense component, which allows us to show some fallback content (such as a loading indicator) while we're waiting for the lazy component to load.. [Accessed 8 9 2021].

[78] Remix, "history," GitHub, Inc, 14 August 2021. [Online]. Available: https://github.com/remix-run/history#history---. [Accessed 4 September 2021].

[79] R. Router, "Hooks," React Training, 2021. [Online]. Available: https://reactrouter.com/web/api/location. [Accessed August 25 2021].

[80] M. W. Docs, "Lazy loading," Mozilla and individual contributors, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading. [Accessed 12 September 2021].

[81] M. W. Docs, "Code splitting," Mozilla and individual contributors, [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Code_splitting. [Accessed 15 September 2021].

[82] V. Router, "API Reference," Evan You, Eduardo San Martin Morote, [Online]. Available: https://router.vuejs.org/api/#router-link. [Accessed 8 September 2021].

[83] Vue3, "Suspense," Evan You, 7 3 2021. [Online]. Available: https://v3.vuejs.org/guide/migration/suspense.html#suspense.

[84] Paweł, "What are Single Page Applications(SPA)?," Forem, 15 September 2019. [Online]. Available: https://dev.to/kendyl93/what-are-single-page-applications-spa-32bh. [Accessed 15 August 2021].

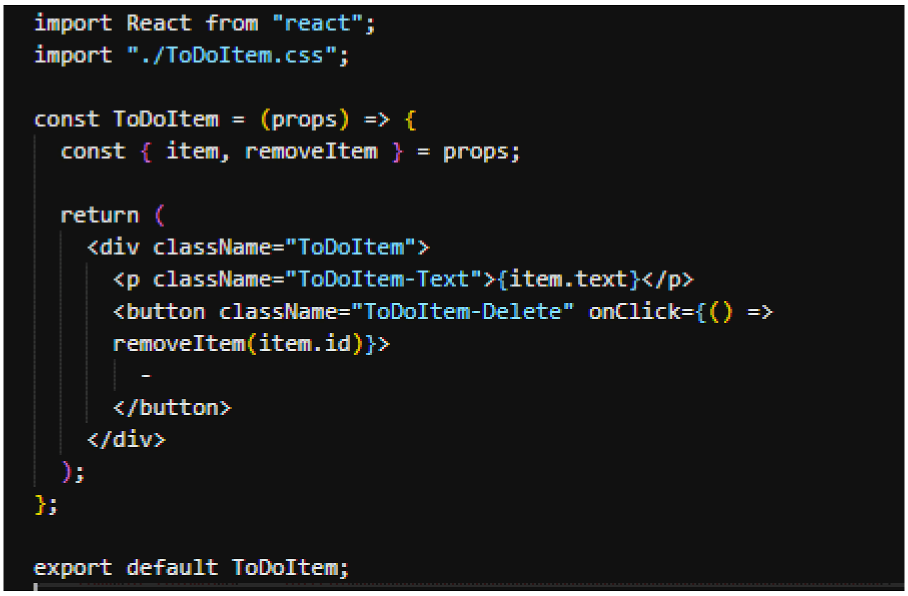
[85] A. HelloJS, 2017. [Online]. Available: https : / / blog . hellojs . org /.

[86] H. Gerstaecker, "Single Page Applications: The Rise of Web Apps in 2020," 1 March 2020. [Online]. Available: https://hackernoon.com/single-page-applications-the-rise-of-web-apps-in-2020-un6c32gm. [Accessed 26 August 2021].

[87] P. CHANDRA, "Redux Architecture Overview," LinkedIn , 22 March 2021. [Online]. Available: https://www.linkedin.com/pulse/redux-architecture-overview-priyesh-chandra-1c/. [Accessed 30 August 2021].

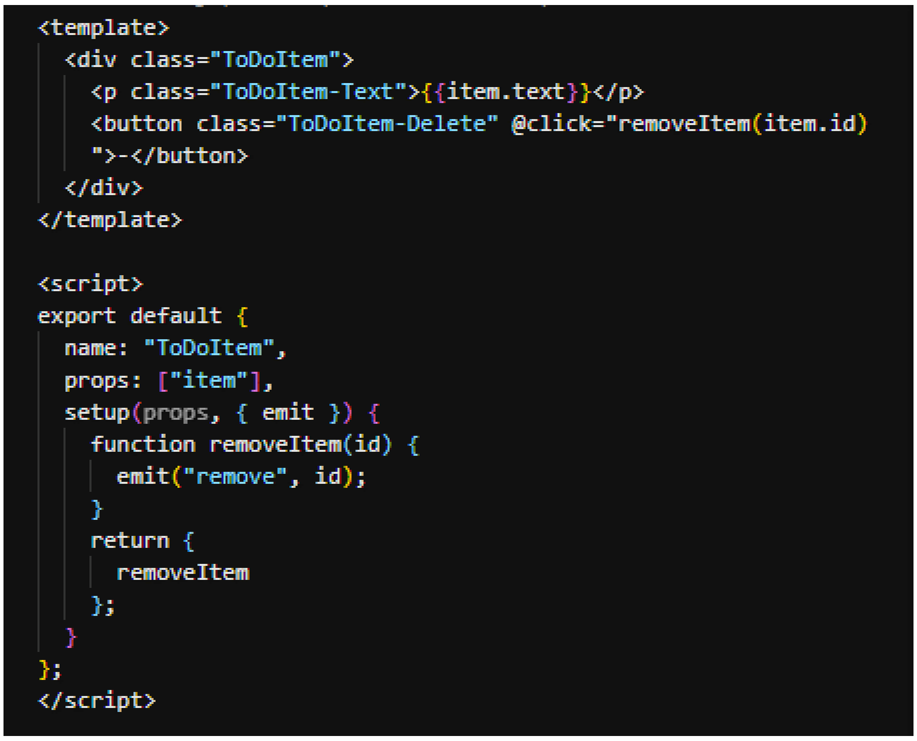
Appendix

Source Code of To-Do Applications



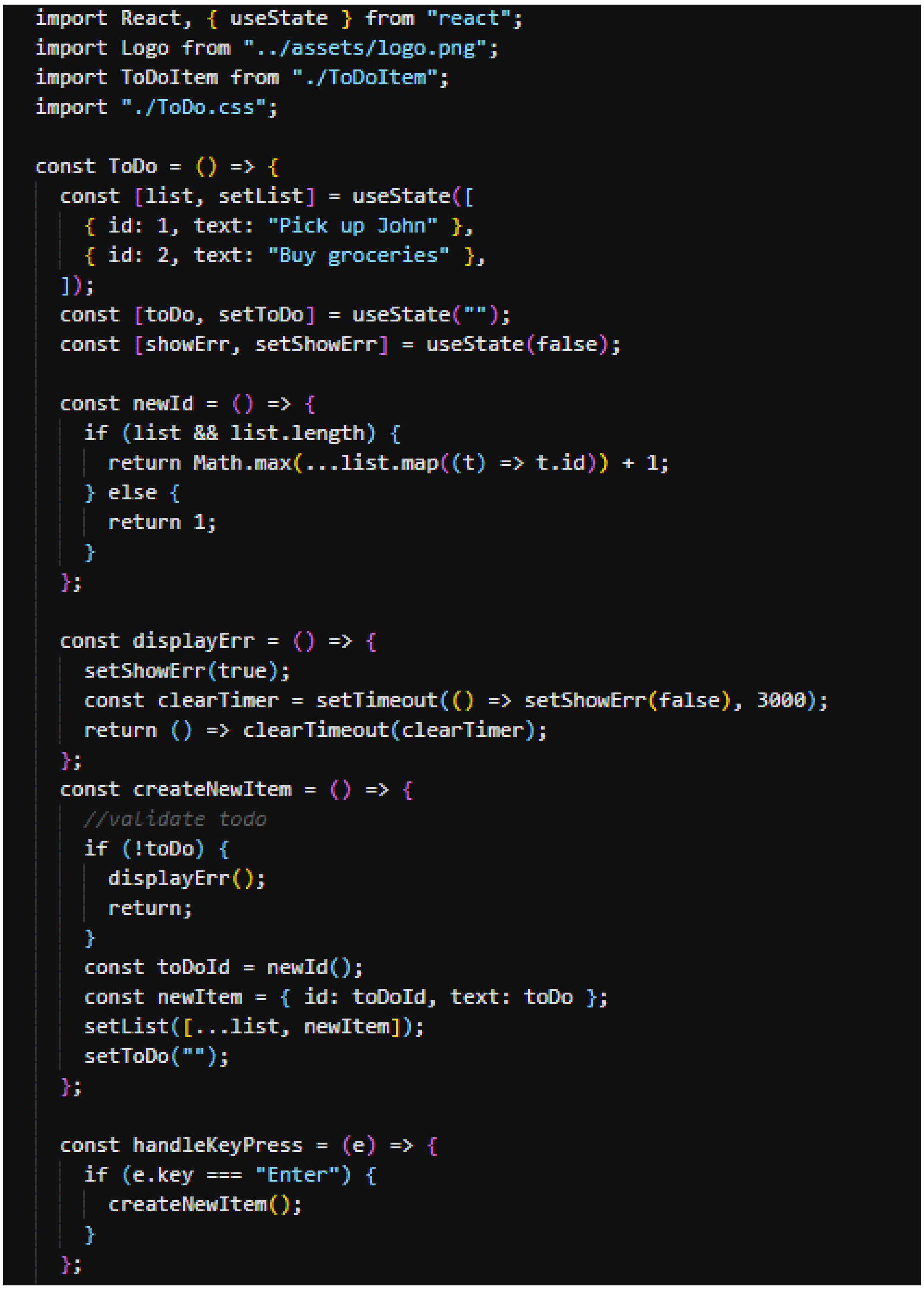
```
import React from "react";
import "./ToDoItem.css";

const ToDoItem = (props) => {
  const { item, removeItem } = props;

  return (
    <div className="ToDoItem">
      <p className="ToDoItem-Text">{item.text}</p>
      <button className="ToDoItem-Delete" onClick={() =>
      removeItem(item.id)}>
        -
      </button>
    </div>
  );
};

export default ToDoItem;
```

Figure 12. React's *ToDoItem.js* source code



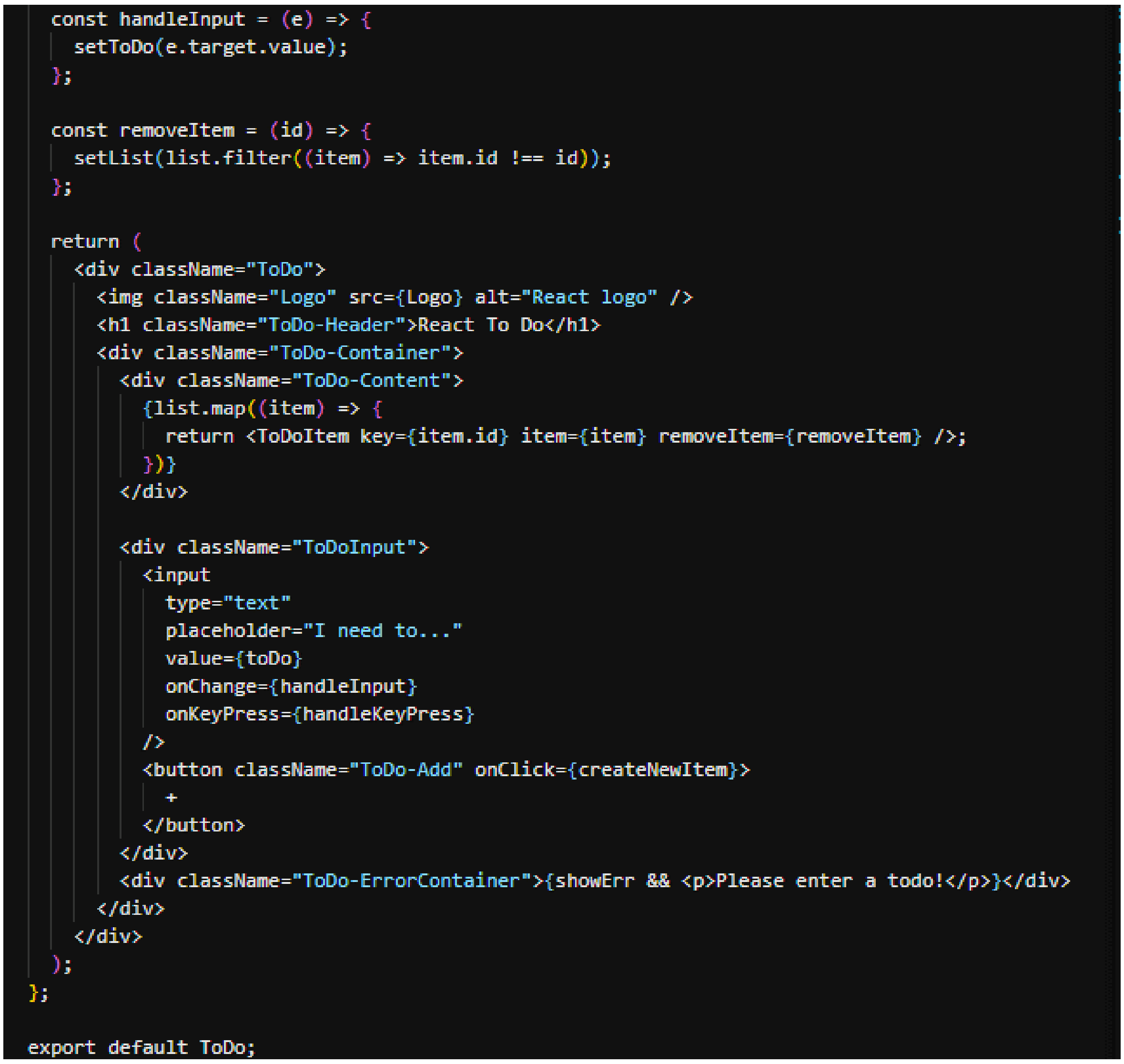
```
<template>
  <div class="ToDoItem">
    <p class="ToDoItem-Text">{{item.text}}</p>
    <button class="ToDoItem-Delete" @click="removeItem(item.id)
    ">-</button>
  </div>
</template>

<script>
export default {
  name: "ToDoItem",
  props: ["item"],
  setup(props, { emit }) {
    function removeItem(id) {
      emit("remove", id);
    }
    return {
      removeItem
    };
  }
};
</script>
```

Figure 13. Vue's *TodoItem.vue* source code



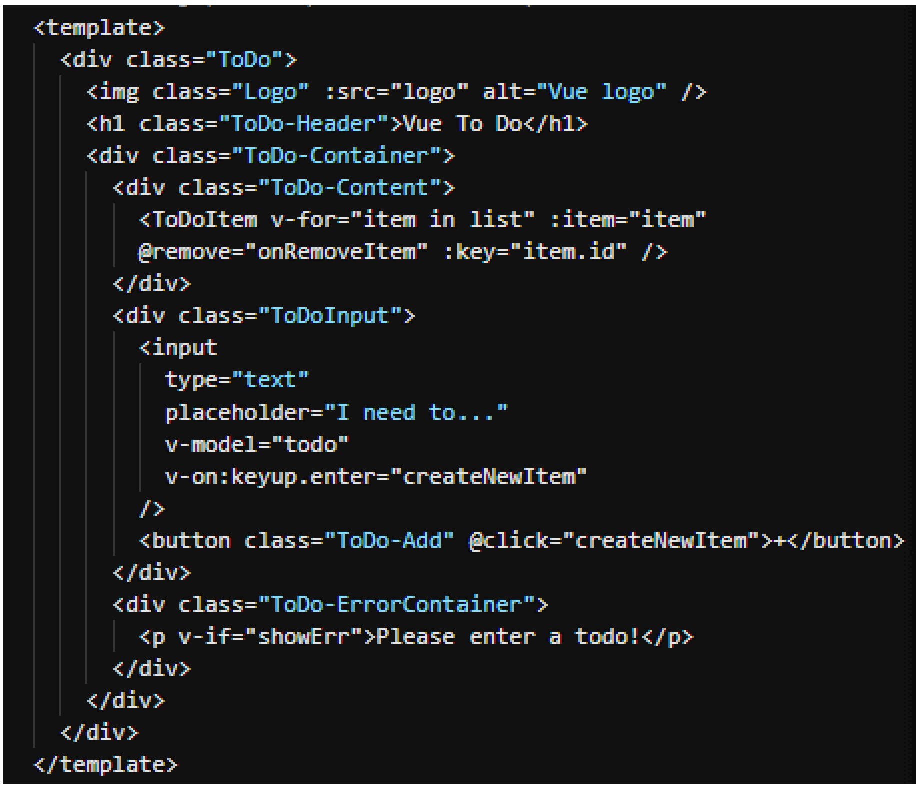
```
import React, { useState } from "react";
import Logo from "../assets/logo.png";
import ToDoItem from "./ToDoItem";
import "./ToDo.css";

const ToDo = () => {
  const [list, setList] = useState([
    { id: 1, text: "Pick up John" },
    { id: 2, text: "Buy groceries" },
  ]);
  const [toDo, setToDo] = useState("");
  const [showErr, setShowErr] = useState(false);

  const newId = () => {
    if (list && list.length) {
      return Math.max(...list.map((t) => t.id)) + 1;
    } else {
      return 1;
    }
  };

  const displayErr = () => {
    setShowErr(true);
    const clearTimer = setTimeout(() => setShowErr(false), 3000);
    return () => clearTimeout(clearTimer);
  };
  const createNewItem = () => {
    //validate todo
    if (!toDo) {
      displayErr();
      return;
    }
    const toDoId = newId();
    const newItem = { id: toDoId, text: toDo };
    setList([...list, newItem]);
    setToDo("");
  };

  const handleKeyPress = (e) => {
    if (e.key === "Enter") {
      createNewItem();
    }
  };
```

Figure 14. React’s *ToDo.js* (first half)



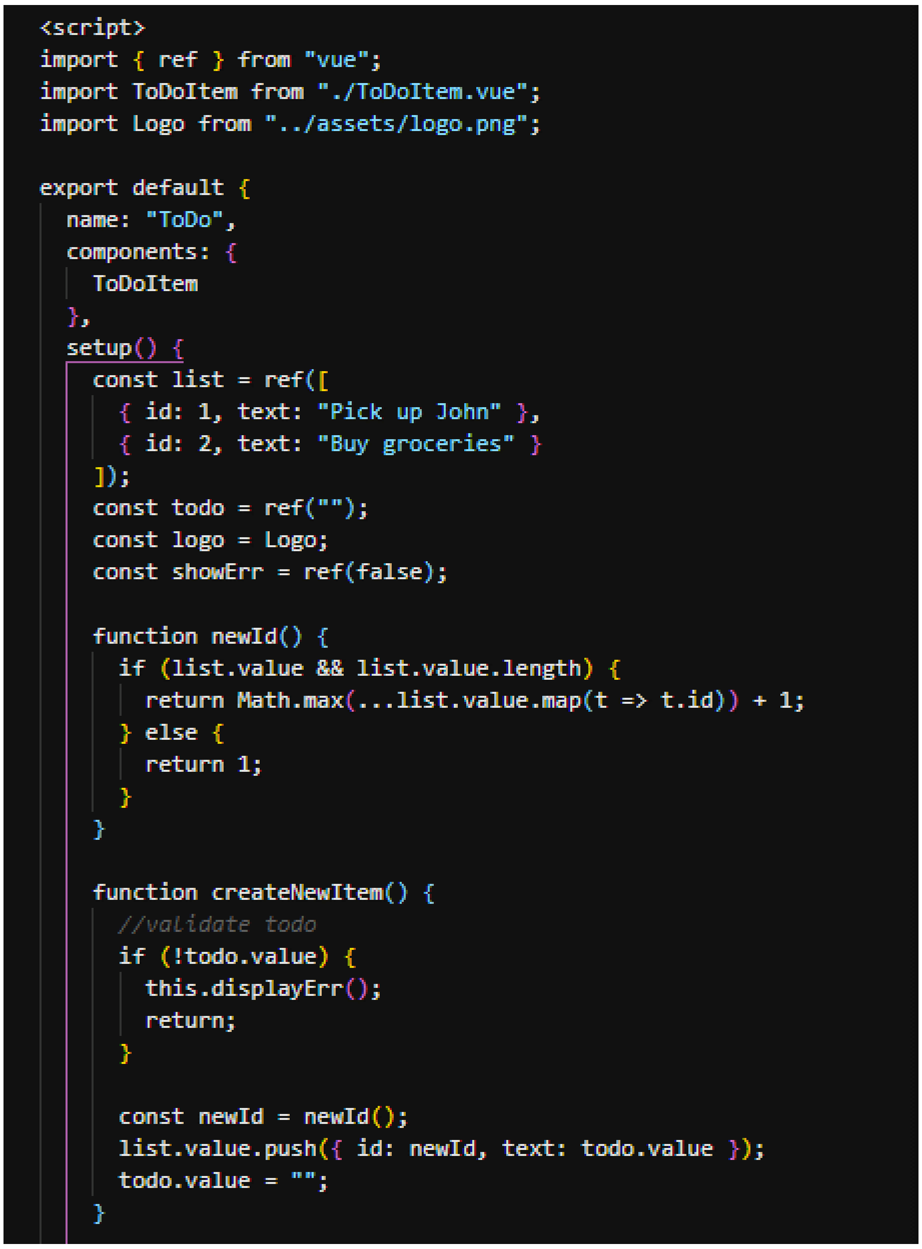
```
  const handleInput = (e) => {
    setToDo(e.target.value);
  };

  const removeItem = (id) => {
    setList(list.filter((item) => item.id !== id));
  };

  return (
    <div className="ToDo">
      <img className="Logo" src={Logo} alt="React logo" />
      <h1 className="ToDo-Header">React To Do</h1>
      <div className="ToDo-Container">
        <div className="ToDo-Content">
          {list.map((item) => {
            return <ToDoItem key={item.id} item={item} removeItem={removeItem} />;
          })}
        </div>

        <div className="ToDoInput">
          <input
            type="text"
            placeholder="I need to..."
            value={toDo}
            onChange={handleInput}
            onKeyPress={handleKeyPress}
          />
          <button className="ToDo-Add" onClick={createNewItem}>
            +
          </button>
        </div>
        <div className="ToDo-ErrorContainer">{showErr && <p>Please enter a todo!</p>}</div>
      </div>
    </div>
  );
};

export default ToDo;
```

Figure 15. React’s *ToDo.js* (second half)



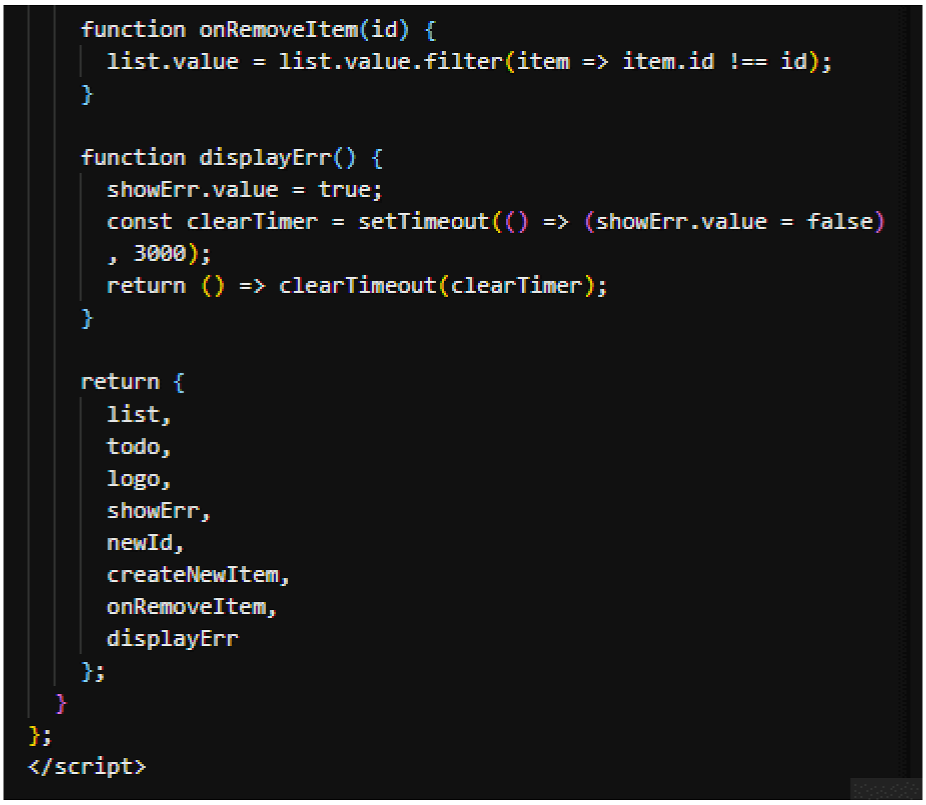
```
<template>
  <div class="ToDo">
    <img class="Logo" :src="logo" alt="Vue logo" />
    <h1 class="ToDo-Header">Vue To Do</h1>
    <div class="ToDo-Container">
      <div class="ToDo-Content">
        <ToDoItem v-for="item in list" :item="item"
        @remove="onRemoveItem" :key="item.id" />
      </div>
      <div class="ToDoInput">
        <input
          type="text"
          placeholder="I need to..."
          v-model="todo"
          v-on:keyup.enter="createNewItem"
        />
        <button class="ToDo-Add" @click="createNewItem">+</button>
      </div>
      <div class="ToDo-ErrorContainer">
        <p v-if="showErr">Please enter a todo!</p>
      </div>
    </div>
  </div>
</template>
```

Figure 16. Vue's Template part of *ToDo.vue*



```
<script>
import { ref } from "vue";
import ToDoItem from "./ToDoItem.vue";
import Logo from "../assets/logo.png";

export default {
  name: "ToDo",
  components: {
    ToDoItem
  },
  setup() {
    const list = ref([
      { id: 1, text: "Pick up John" },
      { id: 2, text: "Buy groceries" }
    ]);
    const todo = ref("");
    const logo = Logo;
    const showErr = ref(false);

    function newId() {
      if (list.value && list.value.length) {
        return Math.max(...list.value.map(t => t.id)) + 1;
      } else {
        return 1;
      }
    }

    function createNewItem() {
      //validate todo
      if (!todo.value) {
        this.displayErr();
        return;
      }

      const newId = newId();
      list.value.push({ id: newId, text: todo.value });
      todo.value = "";
    }
```

Figure 17. Vue’s script part of *ToDo.vue* (first half)



```
      function onRemoveItem(id) {
        list.value = list.value.filter(item => item.id !== id);
      }

      function displayErr() {
        showErr.value = true;
        const clearTimer = setTimeout(() => (showErr.value = false)
        , 3000);
        return () => clearTimeout(clearTimer);
      }

      return {
        list,
        todo,
        logo,
        showErr,
        newId,
        createNewItem,
        onRemoveItem,
        displayErr
      };
    }
  };
  </script>
```

Figure 18. Vue’s script part of *ToDo.vue* (second half)