Markus Pikkanen

React and Vue performance comparison

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

1 June 2021

Abstract

Author:	Markus Pikkanen
Title:	React and Vue performance comparison
Number of Pages:	31 pages
Date:	1 June 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Janne Salonen, Head of School

The objective of this thesis was to measure and compare the performance of React and Vue JavaScript libraries. The thesis starts with a short introduction to both libraries. How the performance was measured and what was used to measure it are discussed next.

Two types of tests were made to test the libraries, Ranking tests and smaller tests. Ranking tests' test subject was a web application named Ranking. From Ranking application, measurements were taken with the help of a tool called Lighthouse. Smaller tests' test subjects were two types of pages that display random text paragraphs. These test subjects were tested by using Performance API found in web browsers. Both tests have their own sections, which discuss the details of how the tests were conducted, and what the results were.

As a result of this final year project, it can be concluded that both libraries perform rather well. Moreover, sampling of these test was not enough to determine which library performs better. React thrived in some tests and Vue in others. Finally, it was not clear how accurate the results were in one of the tests. As for that test the measurements relied on library functions.

Keywords: React, Vue, Lighthouse, Performance

Tiivistelmä

Tekijä:	Markus Pikkanen
Otsikko:	React ja Vue kirjastojen suorituskyky vertailu
Sivumäärä:	31 sivua
Aika:	1.6.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto-ja viestintätekniikka
Ammatillinen pääaine:	Software Engineering
Ohjaajat:	Osaamisaluepäällikkö Janne Salonen

Tämän opinnäytetyön tavoitteena oli mitata ja vertailla React- ja Vue JavaScript -kirjastojen suorituskykyä. Opinnäytetyö alkaa molempien kirjastojen lyhyellä kuvauksella. Sen jälkeen alustetaan siihen, kuinka suorituskykyä mitattiin ja mitä käytettiin sen mittaamiseen.

Kahdentyyppistä testiä tehtiin testaamaan näitä kirjastoja. Ensimmäinen on ranking-testit ja toinen on pienemmät testit. Ranking-testeissä testauksen kohteena oli ranking-niminen selainsovellus. Ranking sovelluksesta otettiin mittauksia käyttäen Lighthouse nimistä työkalua. Pienempien testien testikohteena oli kaksi erityyppistä sivua, jotka molemmat listaavat satunnaisteksti kappaleita. Näitä sivuja mitattiin selaimista löytyvää Performance-nimistä rajapintaa käyttäen. Molemista testeistä on omat osionsa, joissa käydään läpi, kuinka testit suoritettiin ja mitä saatiin tulokseksi testeistä.

Lopputulema testeistä oli, että molemmat kirjastot suoriutuvat hyvin, eikä testien otanta ollut tarpeeksi suuri, jotta voisi päätellä, kumpi olisi kaiken kaikkiaan tehokkaampi. React menestyi toisissa testeissä ja Vue taas toisissa. Lisäksi heräsi epäilyjä tulosten paikkaansapitävyydestä niissä testeissä, joissa turvauduttiin kirjastojen omiin funktioihin mittauksia otettaessa.

Avainsanat: React, Vue, Lighthouse, Suorituskyky

Contents

List of Abbreviations

API	Application Programming Interface is an interface that defines interactions between multiple software applications.
CSS	Cascading Style Sheets is language that expresses how document written in markup language should be presented e.g. HTML.
FCP	First Contentful Paint is a measurement of how long it takes to render first content on a web page.
HTML	HyperText Markup Language is markup language for documents ment to be displayed in web browsers.
LCP	Largest Contenful Paint is a measurement of how long it takes to render the largest image video or text block on a web page.
Performance API	Peformance Api is an interface inside website context that provides access to performance-related information for the current web page.
SPA	Single-Page Application is a web page that interacts with the user by dynamically altering the contents of the page document.
Speed Index	Speed Index is measurement of how quickly content of a web page is being loaded into webpage’s view.
TBT	Total Blocking Time total sum of time user input is being blocked between FCP and TTI. Time considered to be

	blocking time is the time 50 milliseconds after the start of an any JavaScript task until completion of the task.
TTI	Time to Interactive is a measurement of how long it takes to website to become fully interactive.

1 Introduction

This thesis aims to find out if there is a performance difference between two major web user interface libraries React and Vue. As according to "State of JS 2020" survey [1], React being the most used frontend framework and Vue steadily growing in popularity and becoming a reasonable rival for React. By seeing this growth of Vue’s popularity a question came up into mind. Is there benefits for using one over another? There are different ways a library could be evaluated: usability, maintainability etc. This thesis focuses on comparing the performance aspect of these libraries.

This thesis starts by short introduction to the libraries, first React and then Vue. Then comes section for Ranking test and introduction to its test subject “Ranking application”. Ranking section divides into first and second test sections with their own parts for results and summary. After Ranking section comes section for the smaller tests. In smaller tests section there’s introduction to test types and test subjects, part for test results and summary of the results. Finally there is conclusions in of all the tests and results.

2 Introduction to React and Vue

React and Vue are libraries that are made to aid in creating interactive user interfaces for web pages. Both are modular and can be integrated with other JavaScript libraries. Modularity also enables to create only parts of a web page with the library or the complete page can be created with using these libraries. Pages with majority if not all the page content created through these libraries are called Single-Page Applications (SPA). All the test subjects created for this thesis's test are Single-Page Applications by nature. [2, 3]

React’s and Vue’s base building bocks for both are things called components. In essence, these components do four things. First and second thing is to render HTML or other components. Third thing is to handle modify and handle component’s own internal data. Fourth thing is to handle external data given to the component. This external data is called properties for both React and Vue. Components can be programmed to display data and react on data changes whether it is internal data or external data in form of properties. The way how changes into the data made is up to the developer. One common way is to make changes by adding event listeners to the HTML that change the data. e.g. button click event listener. [4, 5]

There are two ways a component can be written in React. These ways are class components and functional components. Class components are components that base on ES6 classes. These eventually are JavaScript objects which have certain functions in it. Function called “render” is the only one that is required to be defined. This function is made to return the description of the content it should render. Other functions are optional and are about adding logic that trigger on different component lifecycle events e.g. “componentDidUpdate()” function is triggered after the component has updated. There is also state (internal data) and props (properties) objects attached to the component and, as mentioned before, component can be programmed to display content according to the contents of these objects. Functional components in the other hand are more simple way of writing react component. It is given props as argument and should return a description of the content to be rendered. If more *component centric* functionality is needed to be added into functional components, then functions called “hooks” can be used inside functional components. With the introduction of hooks into react, functional components have become the major way of writing React and is recommended to be used by creators of React. React code written for the test subjects are written with using functional components. [6, 7]

In Vue, there is currently one way of writing a component and it is more similar with the React's class component than a functional component. Vue's component consist of data, props, template, style and event functions. Data is component's internal data, props is external data given to the component, template is description of the content and style is description of the CSS

(Cascading Style Sheets) attached to the component. One notable difference Vue components have compared to React’s is that Vue has its own way of attaching CSS to components. Attaching CSS into React components is done by writing own separate CSS files or by use of some library that enables similar usage as in Vue. [8]

3 Introduction to thesis

The first tests called “Ranking tests” are done by using website auditing library called “Lighthouse” which is developed by Google. The target for these tests is web application called “Ranking”. Two versions of the Ranking application have been made for both React and Vue. These versions are then measured using Lighthouse and values compared between library versions.

The second tests are smaller tests made with using API existing in browsers called Performance API. Test subjects for these smaller tests are two types of pages that both list random word paragraphs. Again, both of these have their own versions for React and Vue. Performance API is used in similar way as Lighthouse. Measurements collected with it and then the results are compared with each other.

The data collecting in the tests are automated by running a node program that starts a chrome browser with the aid of Puppeteer. A high level API to control Chrome browser programmatically [9]. After starting the browser the program executes the needed commands for the tests against the web pages and then stores the results into files. In Ranking tests the browser instance is given to Lighthouse to control and Lighthouse is configured to make the tests. In smaller tests code made for the tests is executed in the browser context that then returns Performance API results for further processing.

A warm-up phase is used in all of the tests before taking measurements as initially executed code has higher variability in performance. This variability is due to processor and engine optimization which makes the previously executed code effect the currently executed code. It is favourable to have as consistent result as possible so the machine is *warmed up* for the measurements to always get as performant results as possible. So called *cold starts* are eliminated from the measurements. [10 p. 2, 11 p. 1]

All of the target pages have been made with React version 17.0.1 and Vue version 2.6.12. Also all the processes and programs needed for the tests are ran from the same computer. The computer in question is ThinkPad X1 Carbon 6th generation running Windows 10 with hardware specifications as follows:

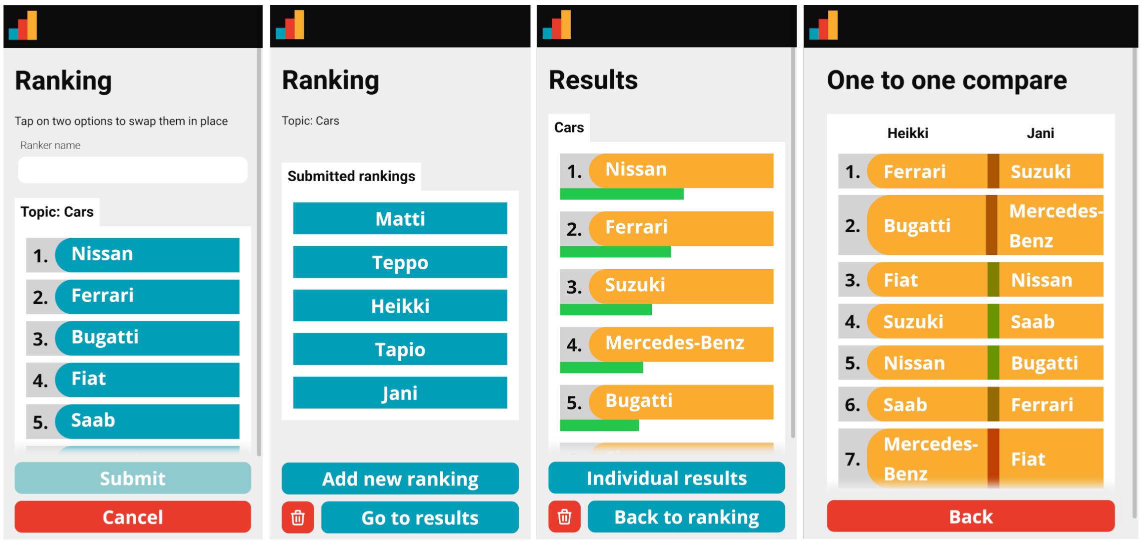
- 16 Gigabytes of LPDDR3 2,133 MHz RAM
- Intel i7 8550U CPU
- Samsung MZVLB1T0HBLR-000L7 M.2 NVMe SSD.

4 Ranking tests

4.1 Test subject: Ranking application

The application used as subject for the tests is personal project named Ranking. Ranking is website where you can create topics, add things into topic and rank out the things in the topic with multiple people. After submitting rankings into the topic ranking results can be observed. Ways of observing the results are: how things ranked out overall and comparing single submissions to each other and to overall results.

Steps for using the Ranking application is as follows. First a topic is created by giving it a name then options are added into the topic. At least three options need in topic to create it. When enough options are given and topic is created, submitting rankings into the topic can begin. When submitting a ranking into topic a ranker name is given and then topic's options can be sorted by ranker’s preference as seen in the first view of Figure 1. Submitting rankings into the topic can be done as many times as wanted but at least two rankings are needed to continue. In second view in Figure 1. there is situation were five rankings have been added. In that view possible actions are to add more, edit previous submissions or go to results. Finally, when there is topic in place and rankings submitted to it the results can be viewed. There are three ways the results can be observed. Fist is a view of overall result of rankings. Here all the options in submissions are scored and sorted by most favoured to least favoured options. This sort of view can be seen in third view fo Figure 1. Second result view is were individual submissions can be observed and compared to the overall results. In the third results view individual submissions can be compared to one another. In Figure 1. fourth view is a view of this results view. In both second and third results view a colour band is used to indicate how similarly or differently the options are ranked out.



	Heikki	Jani
1.	Ferrari	Suzuki
2.	Bugatti	Mercedes-Benz
3.	Fiat	Nissan
4.	Suzuki	Saab
5.	Nissan	Bugatti
6.	Saab	Ferrari
7.	Mercedes-Benz	Fiat

Figure 1. Views from Ranking application

Two versions of the test application have been made, one with React and the other with Vue. Both are made identical to each other from the user perspective. Visual look and behaviour of the application versions do not differ from each other. For development process, there were few rules set for both React and Vue versions of Ranking application before they were made. First is to structure the project the same way. Division between components, routing and styling are structured the same way. Second rule is to have as similar coding experience as possible between the projects. Third rule is to maximize ease and speed of development. This means to use project initializers and libraries to speed up development. Fourth rule is to write the projects with clean code as much as possible. Fifth rule is to not try to make any pre-optimization. Sixth rule is to make the application work with as few network requests as possible.

After the ranking site has been loaded to user's browser. The whole site works locally. Submissions are done from same browser and stored locally to browser's localStorage using Web storage API. As all of the required content to use the page is loaded as the page initially loads. Only time that user's network speed and bandwidth affects the performance is at the initial load. This is good when making a broader tests on the test subjects as network requests might introduce variance to the performance that is not caused by the underlying library that is being used and measured.

The data structure used in Ranking application to sore used submitted data is as follows. Everything starts from object that contains all topics with topic name as key for the topic object. The topic object also includes topic name as an attribute “name”. Other attributes for topic object are “options” and “rankings”. Options attribute is array of strings that are the names of the topic's options. Rankings attribute is array of ranking objects. Ranking object consist of attributes “creatorName” which is name of the ranker in string format and “options” which is array of option name strings in that order which the ranker ranked them. An example of this data structure can be seen in Listing 1. The storage object in Listing 1. has single topic with name “Favourite candies” and two rankings submitted to it.

```
{
  "favourite-candies": {
    "name": "Favourite candies",
    "options": [
      "Dumle",
      "Salmiakki",
      "Marianne",
      "Macaron",
      "Gummi bears"
    ],
    "rankings": [
      {
        "creatorName": "Murphy",
        "options": [
          "Gummi bears",
          "Macaron",
          "Dumle",
          "Salmiakki",
          "Marianne"
        ]
      },
      {
        "creatorName": "Verna",
        "options": [
          "Marianne",
          "Dumle",
          "Gummi bears",
          "Macaron",
          "Salmiakki"
        ]
      }
    ]
  }
}
```

Listing 1. Example storage object for Ranking application

Both application versions were initialized with available build tools that ease the creation of production ready code and helps out on the development process. For Vue it is “@vue/cli” (version 4.5.6) package's “vue create” command and for React it is script named “create-react-app” (version 3.4.1). Both of these create a template project with all sorts of useful and needed configurations and libraries to develop SPA. To reach some of the goals mentioned before, some modifications and adding of extra libraries to the projects were needed. For styles (CSS) to be applied the same way, library named “classnames” was added to the React project. Another style related change was “postcss-normalize”. This library makes base CSS to behave the same despite the browser. This library was included by default in React project so it was also added to Vue project. Navigation between different parts in the application is done with routing libraries. Either React or Vue, have this sort of functionality built-in and either of the project initializers included them. So these sort of routing libraries needed to be added. For React it is “react-router-dom” and for Vue it is “vue-router”. Lastly, there is one major configuration that affects what sort code the browser will actually execute. This configuration is called “browserslist”. This configuration tells the different project compilers what will be the target browser. In other words, what JavaScript and ECMAScript features can be used in the compiled code. By default the “vue-cli” and “create-react-app” generated different “browserslist” configurations. To make the application versions run the same in compiled form, both configurations were made to match each other with minimal changes.

While developing the Ranking application versions, tests were made for each page in the application with testing library called cypress. Both React and Vue versions were tested while and after developing to ensure they behaved the same. Cypress is a library made to create tests for anything that runs on the web. In Ranking application the behaviour of the application was tested by ensuring correct content is displayed when pages are opened and interactions with the page do everything that is expected.

4.2 Lighthouse

Ranking tests were measured using Google's Lighthouse tool. There are many different implementations of Lighthouse. In this test the Lighthouse Node module was used. With the Node module version of the Lighthouse the whole fine detailed result data is available for further processing.

Lighthouse measures and observes many different aspects of website. The main categories are performance, progressive web app capabilities, accessibility and how search engine optimized the website is [12]. In this test only the audits in performance category are being used except for Cumulative Layout Shift. As Cumulative Layout Shift is mostly about how the webpage is structured so the underlying framework has little to do with this audit. The performance audits that can be used to measure the frameworks are then First

Contentful Paint, Speed Index, Largest Contentful Paint, Time to Interactive and Total Blocking Time.

First Contentful Paint (FCP) measures how long it takes to render first content on the page. What are considered as content in FCP measurement are images, non-white canvas elements and SVGs. [13]

Largest Contentful Paint (LCP) is similar to FCP with slight difference. As FCP tracks how long it takes for the first content to render. LCP measures how long it takes to render the largest image video or text block on the page. What is considered as largest content on page is the block of content that takes most of the space in the browsers viewport. As the website progressively introduces new content to the page while it loads, if new content ends up being bigger than the previously largest, it will place it as the largest content. [14]

Speed Index measures how quickly in general the content of the page is being loaded. Lighthouse captures video of the page being loaded and uses module called “Speedline” to generate the score for Speed Index. The Speedline's Speed Index algorithm takes visual progress of the page and calculates score by comparing how complete the intermediate steps are compared to the final image of the page. [15, 16]

Time to Interactive (TTI) measures how long it takes to website to become fully interactive. As FCP, LCP and Speed Index measure how fast the page is visually complete. It might not be ready for user interaction. Website is considered fully interactive when the page displays first useful content (FCP), event handlers are registered for most of the visible elements and page can respond to user interactions within 50 milliseconds. [17]

Total Blocking Time (TBT) measures total sum of time user input is being blocked between FCP and TTI. Time considered to be blocking time is the time 50 milliseconds after the start of an any JavaScript task until completion of the task. So from task that takes 80 milliseconds to execute, the blocking time of that task is 30 milliseconds. [18]

Lighthouse tests work by giving Lighthouse address for the page to test. With the Node module you can adjust also what you want to test and create your own audits to it. As the Lighthouse tests start, Lighthouse opens the given page and runs all the defined audits against the page. As the page as fully loaded test will end and Lighthouse will output all the results.

Before each test execution. Both application's are populated with same dataset. Each page is tested individually as Lighthouse does not do or cannot be made to do any input on the page e.g. click link to change page etc. It just observes the page while it loads.

4.3 Test environment

Both React and Vue versions of the applications are made available locally with serve package that starts up HTTP server which serves files from given directory. As network requests, file reading etc. affect the result, the Lighthouse test are run also against reference website which is also ran with serve. The reference website is called “plank page” and responds with text file that has one single character in it. The size of this request payload is 1 byte. With values from the reference page results it can be seen how much network requesting and server processing has affect on the results. Subtracting these reference page values from the React and Vue values results in a more framework focused results.

4.4 First test

Test results are categorized as follows. Single measurement is one snapshot of single page. Values for each audit is taken once in single measurement. These measurements are taken multiple times from same page. Collection of these measurements from single page is called test run. Test run in this first test is 10 measurements. Reason for taking multiple measurements per page is to get average value that represents the page as accurately as possible. As there is many *moving parts* in computers and rarely if ever a single measurement is exactly the same as another. Finally there is test batch. Test batch is a collection of test runs from each page. In Ranking application this means eight test runs in single test batch. For reference page, only one test run is in test batch as it has only one "page".

Before each test run a warm-up rounds for the targeted page is made. In this test, three warm-up rounds are done. Warm-ups are executed exactly the same way as actual measurement with one exception. The results for the warm-ups are not recorded and stored.

In the Ranking tests data which is used to populate the application mimics an expected state of the application after one ranking has been completed. The data has one topic, topic has seven options and five rankings are submitted into the topic.

As stated before, the tests are conducted by Node program that starts browser which is used to collect the results. In Ranking tests, before the measurements are taken data is inserted into the browser's local storage so that the application has content to display. After that Lighthouse is started which tests the page and then outputs the results in JavaScript object format. This data is then stored to files and further processed and filtered with scripts.

4.4.1 Results

In Table 1. we can see mean, median, standard deviation and median absolute deviation audit values for the plank page. Except for the Total Blocking Time, all values for the audits are 624.29 milliseconds for mean, 623.65 milliseconds for median, 1.16 milliseconds for standard deviation and 0.21 for median absolute deviation. In relative to mean, standard deviation is just ~0.185%. All of the values for reference being the same is to be expected as there is almost nothing for the browser to process in the plank page. Most if not all time delay in the measurements comes from requesting the page from the server. So this could be considered as baseline for the other tests as it is mostly about communication between the browser and the server and not about what is being processed in the browser witch is what the tested libraries are all about.

The reason for TBT being different from other audits is in how it is measured. All other audits are measured from the beginning of page request. As described before, TBT measurement starts from LCP onwards until the end of last long task. And in case of reference page, there seems to be no long tasks after LCP as TBT value is zero. [18]

Table 1. Plank page mean, median, standard deviation and median absolute deviation

Category	Mean	Median	Standard deviation	Median absolute deviation
First Contentful Paint	624.29 ms	623.65 ms	1.16 ms	0.21 ms
Largest Contentful Paint	624.29 ms	623.65 ms	1.16 ms	0.21 ms
Speed Index	624.29 ms	623.65 ms	1.16 ms	0.21 ms
Time to Interactive	624.29 ms	623.65 ms	1.16 ms	0.21 ms
Total Blocking Time	0.00 ms	0.00 ms	0.00 ms	0.00 ms

In Table 2. we can see mean, median, standard deviation and median absolute deviation for React version. All the values are derived from all measurements grouped together from all test runs. FCP, Speed Index and TTI have nearly the same values in each statistic. LCP took slightly longer than the other categories with ~37 millisecond difference on mean and ~55 millisecond difference on median. Also some TBT ended up cumulating in React application with median of 2 milliseconds and mean of 1.513 milliseconds.

Table 2. Ranking application React version mean, median, standard deviation and median absolute deviation

Category	Mean	Median	Standard deviation	Median absolute deviation
First Contentful Paint	1,709.16 ms	1,718.50 ms	40.40 ms	15.75 ms
Largest Contentful Paint	1,746.52 ms	1,773.00 ms	45.98 ms	0.50 ms
Speed Index	1,709.16 ms	1,718.50 ms	40.40 ms	15.75 ms
Time to Interactive	1,709.25 ms	1,718.50 ms	40.29 ms	15.75 ms
Total Blocking Time	1.51 ms	2.00 ms	0.95 ms	0.50 ms

In Table 3. we can see the mean, median, standard deviation and median absolute deviation for Vue version. Values are calculated the same way as in react version, all measurements values from all test runs. All Vue's statistics are quite uniform across all audits when TBT is not taken into consideration. TTI has slightly higher mean and median and a bit lower median absolute deviation than the other categories. Standard deviation ranges from 20.75 to 21.15 milliseconds. No TBT ended up accumulating to for Vue.

Table 3. Ranking application Vue version mean, median, standard deviation and median absolute deviation

Category	Mean	Median	Standard deviation	Median absolute deviation
First Contentful Paint	1,839.03 ms	1,835.35 ms	21.15 ms	12.25 ms
Largest Contentful Paint	1,839.33 ms	1,835.35 ms	20.75 ms	12.25 ms
Speed Index	1,839.03 ms	1,835.35 ms	21.15 ms	12.25 ms
Time to Interactive	1,839.54 ms	1,835.65 ms	20.90 ms	11.82 ms
Total Blocing Time	0.00 ms	0.00 ms	0.00 ms	0.00 ms

When looking at the result per page as shown in Figure 2. which has React and Vue median values of FCP, LCP, Speed Index and TTI with plank page values subtracted from them. We can see that Vue continues to have fairly uniform results across all pages. In the other hand, React's higher variance seems to come from results being more varied between different pages. Variance of the mean values is roughly ~150 millisecond between the pages. TBT was not included in Figure 2. as the scale is so different to other values and Vue had none.

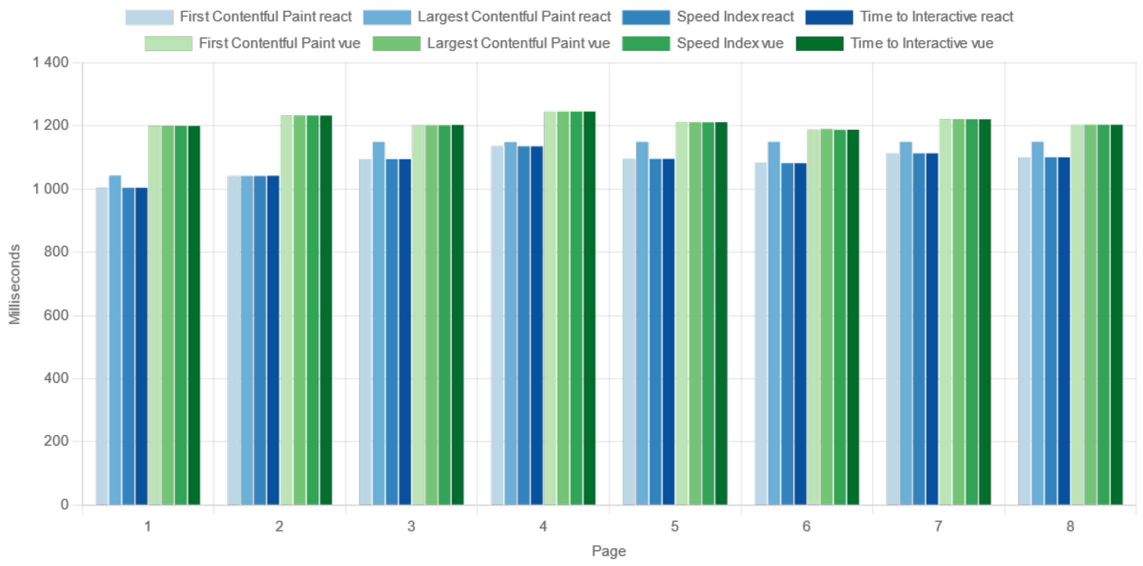


Figure 2. React and Vue median values per page with plank page subtracted

4.4.2 Summary

As seen in results (Table 2. Table 3. and Figure 2.) we can see that there are few ways that React and Vue perform and handle differently in this test. These differences are on speed, variance of speed and uniformity between audits.

React ended up having on average 2 milliseconds of TBT when the Vue had none. This difference might be explained by the difference in FCP, as TBT is measured from the FCP onwards until the end of last long task [17]. So if React would've been a bit slower on FCP it might not also have any TBT also.

When looking at overall speed, we can see that on average React performs slightly faster than Vue. In FCP, speed index and TTI around ~117 milliseconds faster and in LCP around ~62 milliseconds faster. These differences compared to overall time of the categories is ~6.3% in FCP, Speed Index, TTI and ~3.3% in LCP.

Although React is faster it has more variability in its values than Vue. React's standard deviation ranges from 40.40 to 45.98 milliseconds when Vue's ranges from 20.75 to 21.15 milliseconds. That is roughly twice compared to Vue. When compared to mean, standard deviations are from ~2.36% to ~2.63% for React and from ~1.13% to ~1.50% for Vue. Overall it could be considered that the variability is fairly small on both.

When looking at how different categories compare to each other, on both React and Vue. The values are almost or completely the same except for one. That is React's LCP, which has mean of ~38 milliseconds and median of ~55 milliseconds more compared to React's other categories. This indicates that React and Vue handles the page content creation differently. As Vue's FCP and LCP are the same it indicates that Vue creates the whole content at once. React in the other hand has FCP smaller than LCP. This indicates that React creates the content in phases.

As seen there being differences in how page content is created, how fast overall execution is and what sort of variance the executions have. It shows that React is ~62-117 milliseconds faster than Vue depending on category. React also has roughly twice the variance with value of ~2.5% compared to overall measurements. To look this from more broader perspective. The differences between React and Vue seem have very little impact on the user experience at this scale. Empirical tests done in both React and Vue versions support this. By normally using these applications, no difference in performance could be observed by just interacting with the applications. Also for example what is considered to be good LCP value is below 2.5 seconds and poor is above 4 seconds. Both go well under the good threshold. But if taken that measured values and content needed to be rendered correlates with each other. Then Vue would start hitting worse results slightly sooner.

4.5 Second test

In the first test, no significant difference could be seen between React and Vue. Also both were so fast at rendering the pages that no major difference between different categories could be seen. Goal of the second test is see how the results change when the amount of data is increased, or in other words, the amount of things needed to be rendered increases. In the first test only one test batch were taken with the predefined dataset. In second test multiple test batches were taken with each having different sized dataset.

All the datasets in the second test are indicated by number. It tells how many topics the data has, also how many options and rankings are in the first topic. For example test data with number five tells that first topic has five options with five rankings submitted. In addition to first topic, there's four other small topics summing up to five topics in total. Other than the first topic do not need to be scaled has only the first topic is used to test most of the pages.

In the second test, measurements are taken in a similar way than in the first test. Test run for each page, three warm-ups before each test run and 10 measurements taken in one test run. One difference compared to the first test is that the second page, "create topic" page, is not used in this test. Reason for this is that "create topic" page does not load storage data so measurements from it won't be affected by changing the size of data.

4.5.1 Results

In Figure 3. we can see how the measurements went overall for the second test with plank page median values subtracted from them. Some values where almost exactly the same with difference at worst 0.14%. These values are FCP and Speed Index for React and FCP, TTI and Speed Index for Vue. For readability sake, these values are indicated by one plot in Figure 3.

Same as in the first test, React continues to be the faster one in every category. Both React and Vue values increase fairly predictably along with increasing data size. With the increased data size, we can start seeing some variability between different categories. With React, roughly at 150 data size LCP and TTI start to get noticeably more slower than FCP and Speed Index. Vue stays more the same with what has seen in first test. FCP, TTI and Speed Index stay the same throughout different data sizes. LCP, in the other hand, starts getting a bit slower when scaling up.

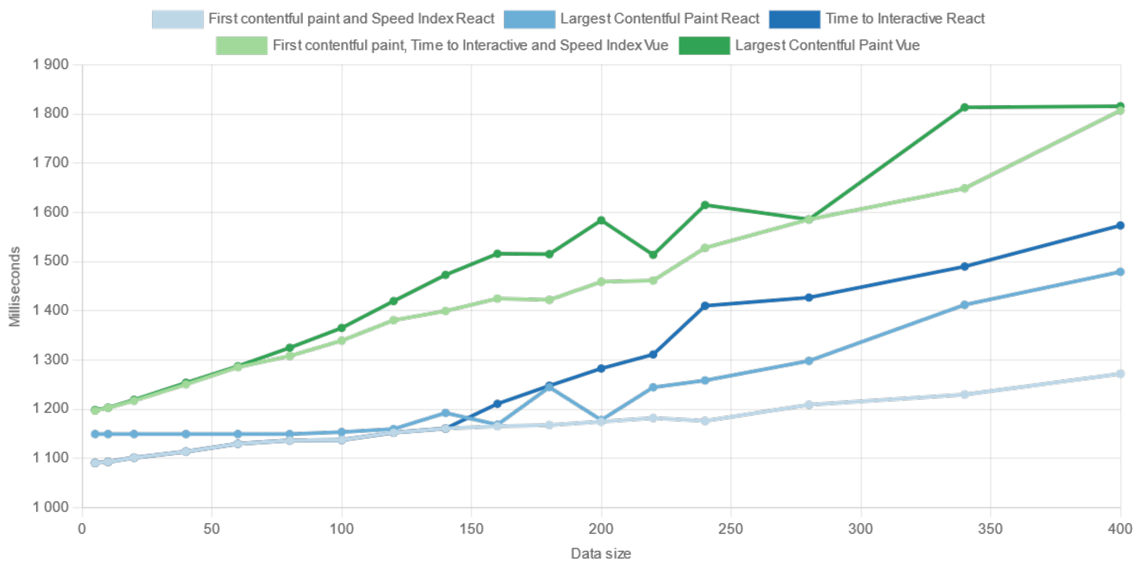


Figure 3. Ranking application second test median values

In Table 4. we can see linear regression slopes of different audits. Most difference is in the FCP, with React having 0.411 ms/data size and Vue having 1.423 ms/data size. That is almost 3.5 times bigger slope than React's. Least difference is in TTI, with React having 1.273 ms/data size and Vue having 1.424 ms/data size. That is roughly 1.12 times bigger than React's. With these trends, it indicates that React would scale better and so work better at more demanding rendering tasks.

Table 4. Ranking application second test slopes

Category	Slope React	Slope Vue
First Contentful Paint	0.411 ms/data size	1.423 ms/data size
Largest Contentful Paint	0.774 ms/data size	1.641 ms/data size
Speed Index	0.411 ms/data size	1.423 ms/data size
Time to Interactive	1.273 ms/data size	1.424 ms/data size

In Figure 4. we can see how relative median absolute deviation (RMAD) changes relative to the data size. Again some of the values were combined under one plot as the values were almost the same, with at most 0.075 difference in RMAD. It looks like the variability rises the same as the results themselves when data size increases. One notable thing is that React's TTI rises rapidly in 150-200 data size range and almost goes above Vue's LCP which has overall highest RMAD. With exception on React's TTI, React has less variability compared to Vue overall. This is the opposite what was seen in first test.

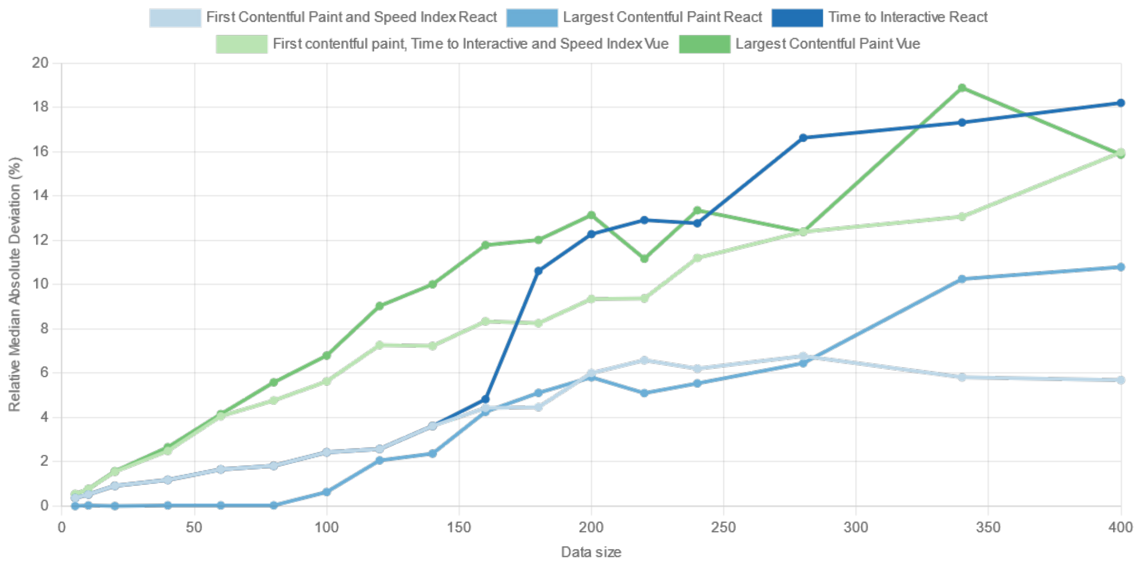


Figure 4. Ranking application categories Relative Median Absolute Deviation

4.5.2 Summary

Second test’s results shows that React continues to be the faster of the two libraries. In the sample range, React starts with better results and also keeps scaling better than Vue on each category.

A swap in which one has the higher variability happened compared to first test. In the first test Vue had less variability in results were React has less on the second test. Possible reason for this sort of change is the difference in the structure of the data used. Where first test has only one topic and many options and rankings. The second test has as many topics as it has options and rankings. As seen in Figure 1. React has noticeably lower values for page one and two. Page one's values are affected by the number of topics in the data because page one lists out all the topics. This difference between pages decreases in second test as there is roughly the same amount of elements needed to be rendered in the first page as there is in the other pages. Also page two was not included in second test which has slightly lower values than the other pages on React version.

Something interesting happens to the React around the 150 data size and after. TTI and LCP start scaling slightly worse after 150 data size. Also the sudden spike in TTI median absolute deviation. One hypothesis for this is that something changes in how react handles the rendering of the content.

4.6 Why some values stay the same

Values for some of the categories stay almost or absolutely the same. For React there is FCP and Speed Index and for Vue there is FCP, TTI and Speed Index. Here is some possible reasons why that is.

Categories that behave the same for both React and Vue, those are FCP and Speed index. FCP as the name states tells the time it takes to render anything that is visible to the user. Speed Index is how fast the end result content is visible. These two might not differ in Ranking application because there is not many stages into rendering complete page. Most if not all page content is requested at initially before even any rendering has started. The speed index would start getting slower if there were e.g. code in React or Vue components that requests remote content. This would mean that first the component is needed to be rendered before the request for more displayable content could be started. This adds then delay before page would be completely rendered. And there would start seeing difference in FCP and Speed Index.

With Vue having FCP and Speed Index same there is also TTI that has also same values than FCP and Speed Index across all measurements. TTI simply is the same as FCP if there is nothing major being processed by browser after FCP [17]. This would indicate that Vue will process all things before rendering anything on the page. Unlike React, which seems to first render content and then process other things done by the library.

5 Smaller tests

Objective of smaller tests is to have more *stripped down* test compared to Ranking tests. Tests were factors outside of tested libraries are minimized even more. Smaller tests also aim to see how React and Vue perform on updating existing page content. Where Ranking tests tested how a *lifelike application* performs, it has things that are not directly part of the tested library and still might have effect on results e.g. CSS and localStorage. Also what was not possible to test on Ranking application is the performance on content modification. This is due to what sort of things Lighthouse measure and the behaviour of Ranking application when it is used.

5.1 Test subjects and test types

Smaller tests have two types of test subjects and two types of tests for both subjects. First test subject is page that lists paragraphs in an ordered list. Second test subject is page that renders paragraphs in a tree-like structure. These test subjects are called "list" and "nested". These two represent the *backbone* of how website content is structured. Both test subjects have their own implementation in React and in Vue and also use pre-generated random word paragraphs as content which they will render. The test types are initial render name “mounting test” and modification render test called “update test”. The mounting test is similar to the Ranking tests. It measures how fast content will be rendered into the web page. The mounting test tests how fast will the libraries be at re-rendering the content if some changes are introduced to it.

The test subjects works as follows. First, a base component is given list of paragraphs to render. In List, this base component then iterates through the list of paragraphs and renders a list item component per paragraph. The list item component then handles the rendering of given paragraph. In nested, the base component starts off a recursive rendering by giving a single component called "child node" the paragraph list and depth variable of zero as properties. This child node component then renders paragraph text from the paragraph list. What paragraph the component renders is determined by using the depth property as index for the paragraph list. After paragraph rendering, the component renders two more "child node" components. To these two components it then gives same paragraph list and it's own depth number incremented by one as properties. This recursive rendering continues until the depth value has reached the length of the paragraph list.

5.2 Implementation

The way how measurements are made in smaller tests base on Performance API. Performance API is found in all modern browsers and is an interface for accessing performance related information of currently open web page. Performance API is used to create and retrieve PerformanceEntries from browser's “performance entry buffer”. These performance entries have identifying properties “name” and “type” and timestamp properties “startTime” and “duration”. These PerformanceEntries are created either indirectly by browser itself on certain events or directly by calling functions “performance.mark()” and “performance.measure()”. “mark()” creates a “mark” type entry into the buffer with “startTime” value being duration from beginning of the document's (web page's) lifetime to time the function was called in milliseconds. “measure()” creates PerformanceEntry with duration as difference in time of two given marks as arguments. Smaller tests make use of these mark and measurement PerformanceEntries to collect performance information from the test subjects. [19]

The way how performance API is implemented to the tests is by adding snippet of code into components inside the test subjects. This snippet creates performance mark in two types of events. First one is when the component has rendered its content the first time. This mark is called "mounted mark". The second one is when the component gets updated. This mark is called "updated mark". Example of these snippets can be seen in Listing 2. for React and in Listing 3. for Vue. This kind of snippet of code is then inserted into whatever component “mounted” and “updated” measurements want to be taken from. For list test subject this sippet is inserted into every list item component and in nested test subject into every child node component.

```
const isNew = useRef(true);

useEffect(() => {
  if (isNew.current) {
    performance.mark(`list-item-mounted`);
    isNew.current = false;
  } else {
    performance.mark(`list-item-updated`);
  }
});
```

Listing 2. Example performance marker code in React components

```
mounted: function() {
  performance.mark(`list-item-mounted`);
},
updated: function() {
  performance.mark(`list-item-updated`);
}
```

Listing 3. Example performance marker code in Vue components

Mounted mark is stored after a component is initially rendered. Updated mark is stored when the component is needed to be re-rendered. Re-render is usually triggered by change in the given component properties. To trigger this re-render for the tests, the page is initially given list of paragraphs with half of the paragraphs shuffled and a sort is triggered which then triggers update on the components. In test page there is sort button which can be clicked to trigger the sort. Right order for the paragraphs is determined by id that is attached to each paragraph. The ids are sequential integers starting from number one. To see how long the framework took to re-render the content a performance mark is stored when the button is clicked.

Measurements from the test subjects are collected in very similar fashion than in the Ranking tests. A script spins up puppeteer browser session, opens the target page that is running on “serve” server and collects measurements. Also different data sizes are used to see how the results scale. For each data size 20 measurements are taken. One major difference compared to Ranking tests is that smaller tests do not use Lighthouse to take measurements. The Performance API is used instead.

How the values are collected for a single measurement is as follows. Mount value is the highest “startTime” value from all the “...-mounted” performance marks. Update value is the difference of button click mark's “startTime” and the highest “...-updated” mark’s “startTime”. Performance API's “measure()” is able calculate these differences by giving it start mark's name and end mark's name. If there is multiple marks with same name, “measure()” takes the mark with highest “startTime” value into consideration.

5.3 List test’s results

Results for the list test subject in mount and update tests can be seen Figure 5 graph. The data size number indicates how many paragraphs the page has to render. This ranges from one to 10,000 in the tests. In mount, React scores median of 452.10 milliseconds and Vue 430.70 milliseconds at data size of one. Both scale very linearly with slopes of 0.125 ms/data size for React and 0.040 ms/data size for Vue. At data size of 10,000 React's mount time is at 1,689.1 milliseconds and Vue's is at 810.45 milliseconds. Update values start very low for both. React from 1.2 millisecond and Vue from 1.1 millisecond at data size one. Also linear increase in update can be seen also. React has so high slope of 0.158 ms/data size on update that its trend line intersects with the mount's trend line at around 9,500 data size and ends up with higher time at 10,000 data size with 1,742.30 milliseconds. Vue has much lower slope of 0.045 ms/data size. That gets it to 447.20 milliseconds at 10 000 data size.

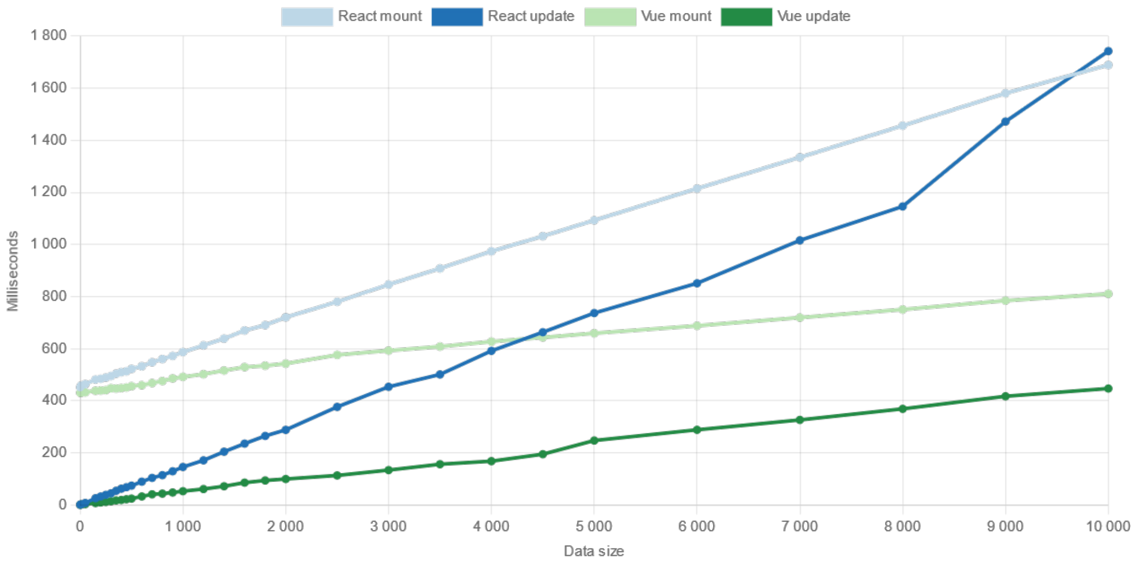


Figure 5. Smaller test list median values

5.4 Nested test's results

The nested test subject's data size behaves differently compared to list test subject's. This is because the data size indicates how many *levels deep* the tree structure is rendered and for each level the amount of paragraphs is doubled compared to the previous level. This is because, as explained previously, if not full depth is reached child node component will render two child node components more for the next level. E.g. with data size three, level one has one paragraph, level two has two paragraphs and level three has four paragraphs. The sum is seven paragraphs for data size of three. This results in amount of paragraphs increasing exponentially with the increase of data size. At data size 14 there's 16 383 paragraphs to render. This increase in amount of paragraphs can be seen in the nested test subject's results in Figure 6.

As seen in Figure 6. values for nested test subject start fairly low but begin to ramp up noticeably at 8-10 data size. For mount, end resulting values, at data size 14 are 1 833.40 milliseconds for React and 759.55 milliseconds for Vue.

Update times grow slightly slower than Vue's mount time. At size 14 React has 729.60 milliseconds and Vue has 624.50 milliseconds. Interestingly React's update time dips below Vue's values at data size 13 where Vue has median of 323.60 and React has 170.65.

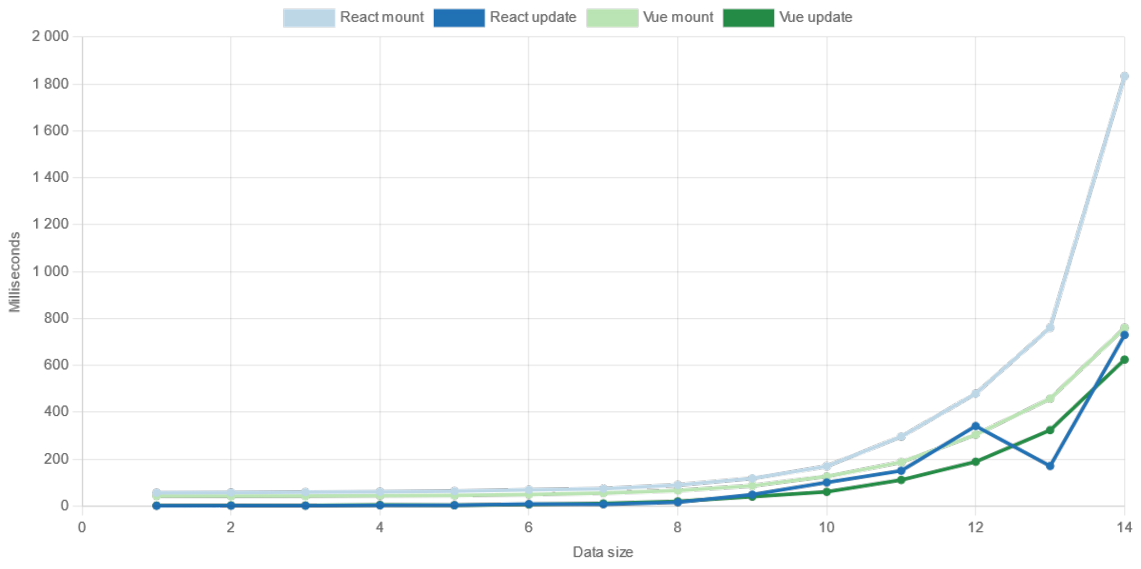


Figure 6. Smaller tests nested median values

5.5 Summary

As results show (Figure 5. and Figure 6.) React is getting more slower than Vue as data size increase. This happens in both tests for both test subjects. Surprisingly React's update on list is starting to have worse results than its mount after around 9,500 data size. It could be thought that rendering everything would be always worse than updating just parts of the already rendered content, but it seems not to be case here. It seems that for React it is more demanding to update list content than create it from scratch after a certain point. One unexpected anomaly can be seen in the nested test subject's React update test result. At data size of 13 it deviates from the trend noticeably. And the results were fairly consistent when median absolute deviation is ~1.35 milliseconds when median is 170.65 milliseconds. Any explanation for this anomaly could not be found. Worst results at the top end were React's update on list with 1,742.30 milliseconds and React's mount on nested with 1,833.40

milliseconds. Overall both frameworks do perform well considering the amount of paragraphs/components they needed to render, for list it is 10,000 at most and for nested it is 16,383 at most.

6 Conclusions

The goal of this thesis was to see if there is performance difference between, JavaScript libraries, Vue and React. The way how this performance comparison was conducted was by taking different kinds of speed measurements from different kinds of test subjects. Speed index, Total blocking time, First Contentful paint and Largest Contentful paint were measured from the Ranking application by using Lighthouse. Mounting and updating speeds were measured from paragraph list and nested paragraphs -pages by using Performance API.

The results were contradictory to each other when comparing Ranking application results to smaller tests results. In Ranking application React is performing noticeably better in every category on higher data sizes. Vue in the other hand is more performant on the smaller tests. Overall both libraries could be considered to be performing well, when looking at the amount of components and content those needed to render in the tests.

There could be a few things that might be the cause of these contradicting results. First one is that there is just too much variability between the libraries. One is more performant on one thing and the other in one thing. The other one is that the method of measuring performance in smaller tests was not reliable enough. As the use of Performance API in the measurement process relied on methods the libraries themselves provide. Unlike in Ranking tests the measurement process was completely independent from the way how the page was created.

To conclude, it can't be said that one library is more performant than the other based on these test. Possibly more testing should be made to get more exhaustive results with more varying test subjects. Also there is indication that relying in the libraries themselves, when taking measurements, could lead to values that are not actually comparable. Better to have tools or methods that do not rely on the test subject at all.

References

1 Sacha Greif and Raphaël Benitte *State of JS 2020: Front-end Frameworks* Available at https://2020.stateofjs.com/en-US/technologies/front-end-frameworks (Accessed 30 May 2021)

2 Facebook Inc. (2021) *React – A JavaScript library for building user interfaces* Available at https://reactjs.org (Accessed 30 May 2021)

3 *Introduction – Vue.js* Available at https://vuejs.org/v2/guide (Accessed 30 May 2021)

4 Facebook Inc. (2021) *Components and Props – React* Available at https://reactjs.org/docs/components-and-props.html (Accessed 30 May 2021)

5 *Props – Vue.js* Available at https://vuejs.org/v2/guide/components-props.html (Accessed 30 May 2021)

6 Facebook Inc. (2021) *Introducing hooks – React* Available at https://reactjs.org/docs/hooks-intro.html#motivation (Accessed 30 May 2021)

7 Facebook Inc. (2021) *Components and Props – React* Available at https://reactjs.org/docs/components-and-props.html#function-and-class-components (Accessed 30 May 2021)

8 *Component basics – Vue.js* Available at https://vuejs.org/v2/guide/components.html (Accessed 30 May 2021)

9 *Puppeteer v9.1.1* Available at https://pptr.dev/#?product=Puppeteer&version=v9.1.1&show=api-overview (Accessed 30 May 2021)

10 Tomas Kalibera, Lubomir Bule & Petr Tuma. 2005. *Benchmark Precision and Random Initial State* International Cherry Hill, New Jersey, USA: Symposium on Performance Evaluation of Computer and Telecommunication Systems

11 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount & Laurence Tratt. (2017) 'Virtual Machine Warmup Blows Hot and Cold' *Proceedings of the ACM on Programming Languages* (52)

12 Google (2021) *Lighthouse | Tools for Web Developers | Google developers* Available at https://developers.google.com/web/tools/Lighthouse (Accessed 30 May 2021)

13 Google (2019) *First Contentful Paint* Available at https://web.dev/first-contentful-paint (Accessed 30 May 2021)

14 Philip Walton (2020) *Largest Contentful Paint (LCP)* Available at https://web.dev/lcp (Accessed 30 May 2021)

15 Google (2019) *Speed Index* Available at https://web.dev/speed-index (Accessed 30 May 2021)

16 (2021) *SpeedIndex | WebPageTest Documentation* Available at https://docs.webpagetest.org/metrics/speedindex (Accessed 30 May 2021)

17 Philip Walton (2020) *Time to Interactive* Available at https://web.dev/tti (Accessed 30 May 2021)

18 Google (2019) *Total Blocking Time* Available at https://web.dev/Lighthouse-total-blocking-time (Accessed 30 May 2021)

19 MDN contributors (2021) *Performance web APIs* | *MDN* Available at https://developer.mozilla.org/en-US/docs/Web/API/Performance (Accessed 30 May 2021)