Iida Kainu

OPTIMIZATION IN REACT.JS
Methods, Tools, and Techniques to Improve Performance of Modern Web Applications

Faculty of Information Technology and Communication Sciences
Bachelor’s Thesis
May 2022

ABSTRACT

Iida Kainu: Optimization in React.js: Methods, Tools, and Techniques to Improve Performance of Modern Web Applications
Bachelor's Thesis
Tampere University
Bachelor's Degree Programme in Computer Sciences
May 2022

The complexity of modern web applications leads to increased performance demands to succeed in the competitive market. For example, the complexity and the number of commonly used libraries leads to slower page loading times. Furthermore, the number of slow Document Object Model (DOM) operations increases in single-page applications (SPA), which are currently trending. While the leading JavaScript library, React.js, includes multiple optimization methods, such as virtual DOM, the responsibility of further optimization is left to the developers. This paper examines which techniques, methods, and tools can be utilized in React.js ecosystem to increase the performance of a web application. The used research method is literary analysis.

As a result of the literary analysis, the current best optimization techniques, methods and tools are listed, such as decreasing the number of re-renders through best practices of React.js state management and memoization, decreasing loading times, and handling compute-intensive tasks with multithreading and memoization. Furthermore, the overview of the optimization methods in React.js illustrates different techniques. However, some methods require a more custom implementation and thus general concepts, such as HTML5 Web Workers API and Webpack module bundler, are reviewed.

Since the presented examples are general and the performance benefits of combinations of the reviewed methods is not predictable, performance profiling through browser developer tools and React.js Profiler component is also introduced. Performance profiling can be used to gain an understanding for the need of optimization and to analyse the actual benefits gained from utilizing the optimization techniques, methods, and tools.

Key words and terms: web development, React.js, optimization, application-level caching, multithreading, code bundling

TIIVISTELMÄ

Iida Kainu: Optimization in React.js: Methods, Tools, and Techniques to Improve Performance of Modern Web Applications (suom. Optimointi React.js-sovelluskehyksessä: modernien web-sovellusten suorituskyvyn parantavia metodeja, työkaluja ja tekniikoita)
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2022

Modernien web-sovellusten monimutkaisuus johtaa kasvaviin suorituskykyvaatimuksiin, jotta sovellukset olisivat käytettäviä ja siten kilpailukykyisiä. Esimerkiksi monimutkaisuus ja vakiintuneiden kirjastojen määrä johtaa hitaampiin sivun latausaikoihin. Lisäksi dokumenttioliomallin (engl. Document Object Model, DOM) hitaiden operaatioiden määrä kasvaa yhden sivun applikaatioissa (engl. Single-Page Applications, SPA), jotka ovat tällä hetkellä suosittuja. React.js, johtava JavaScript-sovelluskehys, sisältää monia optimointimetodeja, kuten virtuaalisen DOM:n, mutta mahdollisesti tarvittava lisäoptimointi on jätetty kehittäjien vastuuksi. Siksi tämän työn tarkoitus on selvittää, mitä metodeja, työkaluja ja tekniikoita voidaan hyödyntää React.js-sovelluskehyksessä web-sovellusten suorituskyvyn parantamiseksi. Työn tutkimusmenetelmä on kirjallisuuskatsaus.

Työn tulokset osoittavat, että erilaisilla metodeilla, työkaluilla ja tekniikoilla voidaan parantaa modernien web-sovellusten suorituskykyä. Ensimmäinen esitetyistä optimointitekniikoista on uudelleenrenderöintien vähentäminen React.js tilanhallinnan parhaiden käytäntöjen ja välimuistiin tallentamisen avulla. Toiseksi applikaation latausaikoja voidaan lyhentää Webpack-työkalun ja resurssien ennenaikaisen lataamisen kautta. Lisäksi raskaan laskennan tuomia suorituskykyhaittoja voidaan vähentää hyödyntämällä monisäikeistä ohjelmointia ja välimuistiin tallentamista. Tutkielmassa tarkastellaan teorian lisäksi React.js-sovelluskehyksen ohjelmointirajapintaa, joka sisältää metodeja monien optimointiratkaisujen toteuttamiseksi. Tästä huolimatta joidenkin tekniikoiden hyödyntämiseksi tulee käyttää mukautetumpaa toteutusta, jonka vuoksi työssä tarkastellaan myös yleisiä web-ohjelmointikonsepteja, kuten HTML5 Web Workers -ohjelmointirajapintaa ja Webpack-työkalua.
Työssä esitetyt optimointiesimerkit ovat geneerisiä ja erilaisten optimointiratkaisujen yhdistelmien suorituskykyhyötyjen määrä ei ole ennalta arvattavia. Tämän vuoksi työssä esitellään suorituskykyprofilointi selaimen kehittäjätyökalujen ja React.js-sovelluskehyksen Profiler-komponentin avulla. Työn tulosten perusteella todetaan, että suorituskykyprofiloinnin avulla kehittäjät voivat analysoida, milloin web-sovellukset vaativat optimointia sekä sitä, onko sovelletuilla optimointiratkaisuilla todellisia hyötyjä.

Avainsanat: web-ohjelmointi, React.js, optimointi, välimuistiin tallentaminen sovellustasossa, monisäikeinen ohjelmointi, lähdekoodin moduulien niputus

Table of Contents

1 Introduction

Modern web applications are increasingly complex, leading to a need for adequate performance in order to succeed in the competitive market. Due to the complexity of the web applications and the number of commonly used libraries, the large code bundle sizes can affect the page loading time, which needs to be minimized to decrease user bounce-rate. Moreover, in modern web development, single-page applications (SPA) are trending. In SPAs, content is rendered dynamically in the same page. This leads to frequent Document Object Model (DOM) manipulations, which are slow. Therefore, many modern frameworks include optimization methods. React.js, the leading JavaScript library for developing modern web applications, features multiple optimization methods. For example, it optimizes the amount of DOM tree manipulations with virtual DOM (VDOM) and component state management. Furthermore, the default build tool Create React App includes some optimization methods. However, the methods React.js implements by default are not always enough, and the responsibility of further optimization is left to developers.

This paper examines which techniques, methods, and tools can be utilized in React.js ecosystem to increase web applications' performance. The used research method is literature analysis, which is detailed in chapter 2. As a result of the literary analysis, general web optimization techniques and React.js ecosystem implementations are explored. First, performance profiling is overviewed in chapter 3 to gain an understanding for the need of optimization in web applications and to provide methods for analysing the actual benefits gained from utilizing the optimization techniques. Secondly, the operation methods of VDOM of React.js and the additional measures developers may need to take to aid it are examined in chapter 4. Thirdly, application-level caching, memoization, is explored in chapter 5 with examples of use. Next, in chapter 6, loading times and the optimization tools to decrease them are examined. Furthermore, the chapter 7 analyses the benefits of multithreading and overviews the Web Workers API. Finally, the results are concluded in chapter 8.

2 Methods

The research method of this paper is literature analysis with Andor selected as the database. The initial search statements were similar to listed statements, such as "(react OR react.js) AND perform*", "(react OR react.js) AND optimi*", and "react.js AND (optimization OR optimisation) AND (method* OR technique*)". In many cases, the number of initial results belonged in the thousands, and they were filtered according to the subject, such as "Computer Science", "Application Software Development", "JavaScript", and "Web applications". Furthermore, the year of publication was limited to 2015-2022 for recent information.

The candidate papers were selected in a multistep process. First, the search results in Andor are ordered in relevance, thus papers were selected for the next phase until the titles appeared to become irrelevant to the research question. For the second phase, abstracts were analysed to determine whether the paper examined optimization related to React.js. In fact, the process of literature analysis was iterative: if a paper mentioned a relevant technique, yet did not address React.js directly, searches with the term were made. In such cases, the search statement followed the formula of "(react OR react.js) AND {technique name}". Furthermore, selected papers were complemented with the original documentation of React.js, Google Developers, MDN Web Docs and Webpack.

3 Performance Profiling

In modern software client projects, developers are often under time pressure to implement new features. While fulfilling the needs of the client is important, so is the perceived quality of the product. Thus, performance profiling can be used to decide whether to spend valuable development time on optimization to increase the user experience. First, it is recommended to overview the general performance of an application to determine if a need for optimization exists (Griffiths & Griffiths 2021). In addition, optimization problems could be identified through exploratory testing by querying if any application operations feel slow. After that, the origin of found problems should be identified (Griffiths & Griffiths 2021). Furthermore, performance profiling can be utilized to analyse the benefits of optimization techniques (Griffiths & Griffiths 2021). This chapter covers the process of performance profiling through tools such as Chrome DevTools Lighthouse and React Developer Tools as well as the React.js Profiler component.

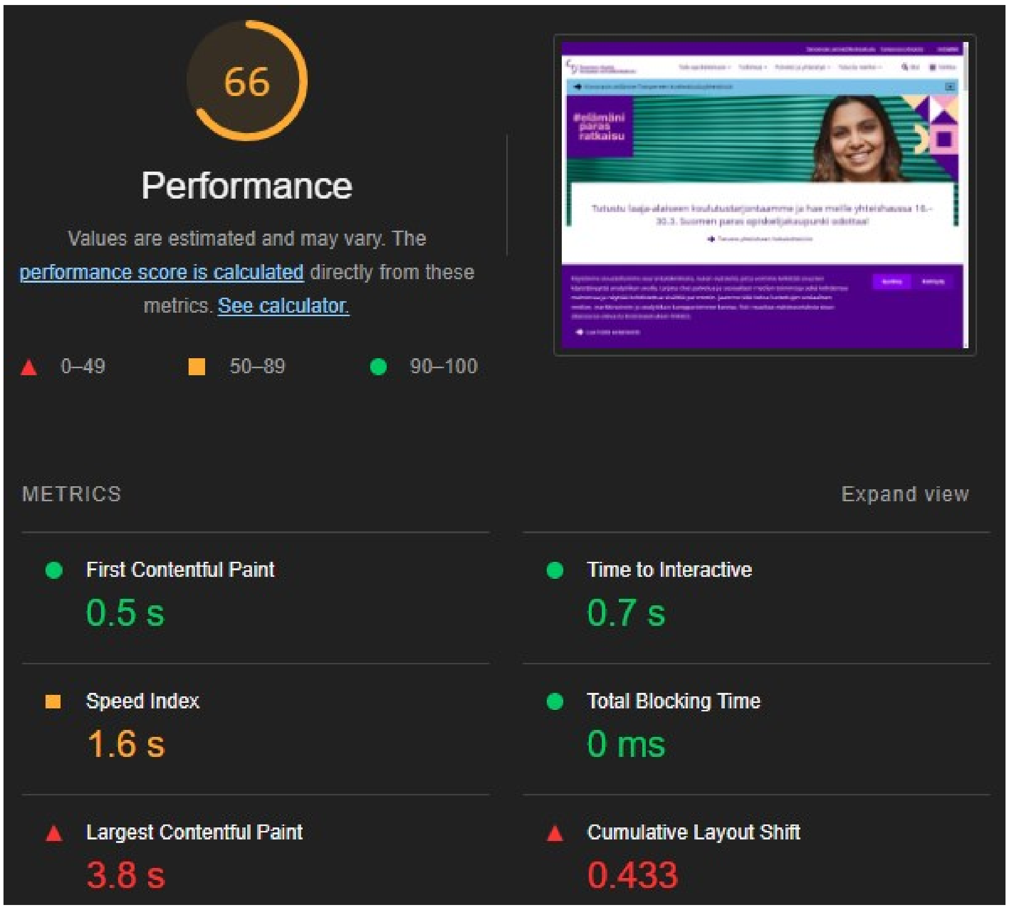
3.1 Chrome DevTools Lighthouse

Lighthouse is one of the Chrome DevTools, a set of developer tools in Google Chrome, which can be utilized to audit the general performance of a web application. For most accurate results, the optimized production build (chapter 6.1) of the application should be used for the audit. Other aspects, such as accessibility, best practices of modern web development, search engine optimization (SEO) and progressive web app standards can also be analysed with Lighthouse, but this paper focuses only on performance.

The performance audit provided by Lighthouse covers various metrics related to page load speed to calculate a weighted performance score ranging from 0-100. The metrics are scored and categorised with colours from red to green, where red and orange scores imply a need for improvement. A total score of 90-100, categorized green, is ideal for good user experience. To determine the scores, the metric values are compared to performance data gathered from the most visited websites of the Internet. (Google Developers 2021)

The first of the Lighthouse performance metrics is *First Contentful Paint* (FCP), the time taken to render the first piece of DOM content after navigating to the page. The second, *Time to Interactive* (TTI), is the time for the page to fully become interactive, regardless of when visual content appears to be ready. The third, *Total Blocking Time* (TBT), is a sum of the time of all long tasks (>50ms) between the phases of FCP and TTI. Next, *Speed index* (SI) represents how quickly content is visually displayed on the page, measured by capturing a video of the page load. Moreover, *Largest Contentful Paint* (LCP) measures the render time of the largest image or text block in the viewport. Finally, *Cumulative Layout Shift* (CLS) is a measure of the largest burst of unexpected layout shifts occurring during the lifespan of a page. An unexpected layout shift means any visible element changing its position without user interaction. Thus, a burst of layout shifts consists of multiple shifts within a short time, maximum of 5 seconds to be exact. For a good user experience, a high CLS score is to be avoided. (Google Developers 2021)

The following audited application (Figure 1) has an overall score of 66 with a need for improvement, although the FCP, TTI and TBT scores are good. In fact, the application features multiple large images, which may have affected the low LCP score. Furthermore, the bursts of layout shifts are noticeable on load of the audited page. However, as emphasized by Lighthouse (Figure 1), performance score can vary between audits of the same page. As a consequence, the results of a later audit were higher, with an overall score of 82 and SI in the green category.



Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.

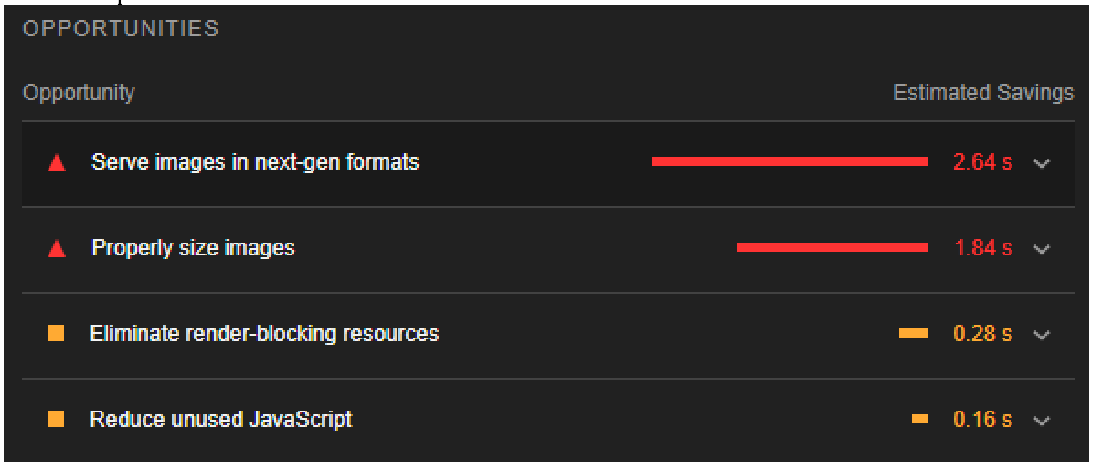
▲ 0–49 ■ 50–89 ● 90–100

METRICS Expand view

● First Contentful Paint 0.5 s	● Time to Interactive 0.7 s
■ Speed Index 1.6 s	● Total Blocking Time 0 ms
▲ Largest Contentful Paint 3.8 s	▲ Cumulative Layout Shift 0.433

Figure 1. Lighthouse performance metrics of tuni.fi.

While it can be beneficial to understand the definitions of the metrics, the recommendation is to focus on improving the total performance score (Google Developers 2021). For this purpose, the Lighthouse report includes Opportunities and Diagnostics sections, which can aid in identifying problem areas: the most significant opportunities or diagnosis are colour coded red and displayed higher in the list (Google Developers 2021). For example, the analysed web application in Figure 2 could benefit from image optimization. Improvement in the area could especially affect the low LCP score (Figure 1) and thus the total performance score.


OPPORTUNITIES

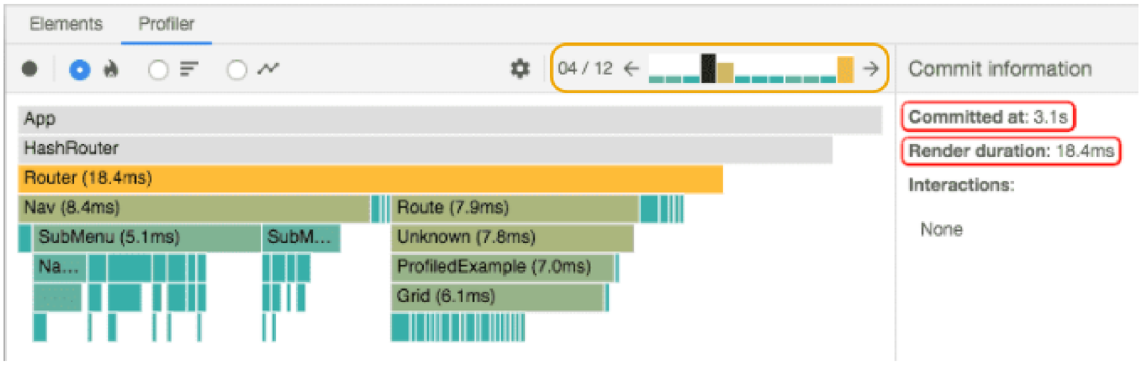
Opportunity	Estimated Savings
▲ Serve images in next-gen formats	2.64 s
▲ Properly size images	1.84 s
■ Eliminate render-blocking resources	0.28 s
■ Reduce unused JavaScript	0.16 s

Figure 2. Lighthouse performance improvement suggestions for tuni.fi.

3.2 React Developer Tools Profiler

The Profiler of React Developer Tools provides an interactive method to identify why a certain performance issue exists. The tool can be downloaded as a browser extension, and it can be located in the "Profiler" tab of DevTools. To profile an application, a browser session is recorded. Performance data is gathered on each render of the application during the session. The data is grouped by each *commit* (Figure 3), meaning the phase when React.js applies changes such as DOM updates. The height of the commits in the commit bar corresponds to the time taken to render. Moreover, commits with fast render time can be hidden from the settings to filter the amount of data shown and thus, simplify the analysis process. (Vaughn 2018)

Once a commit has been selected, its flame chart can be analysed below the commit bar (Figure 3). In the chart, bars of components are presented in the same tree-structure they are created in. In addition to the written render times, the width and colour indicate the length of the component render time. For example, in the selected commit of Figure 3, the Router component took most time to render at 18.4ms, which includes the render durations of Nav and Route components. The chart is interactive: clicking the bars provides a zoomed-in view of the chart and detailed component data, its props and state. Furthermore, selecting a component and looking for changes in its data while browsing commits can reveal why the component re-rendered. (Vaughn 2018).



Committed at: 3.1s
Render duration: 18.4ms
Interactions:
None

Figure 3. Part of React Developer Tools Profiler performance results, where the commit bar is circled in orange and commit information in red. In the lower left corner is the flame chart. (Vaughn 2018)

Overall, the Profiler helps to identify performance bottlenecks in the application. The flame chart can be an efficient way to recognize problematic components since they occupy more of the overall visual space and as such can be easier to notice. In addition, analysing the component data in groups of commits can reveal unnecessary state or prop changes, which could be affecting the application's performance.

3.3 React.js Profiler Component

Utilizing optimization methods, tools, and techniques may take some exploration, especially as many are too general for specific tasks. As such, when implementing optimization solutions, performance profiling should be used to determine whether performance increased. To gather performance data of a component and its descendants, a Profiler component can be utilized (Code example 1). As a note, the component should not be used in production since each commit uses some CPU and memory resources (Meta Platforms 2022f).

Profiler's callback function (Code example 1) is run each time a component within the subtree commits (Meta Platforms 2022f). First parameter of the callback function is the ID of the Profiler, which is useful in case multiple components are utilized at the same time. However, in terms of measuring performance, the most important parameters are actualDuration and baseDuration. ActualDuration measures the time spent rendering the Profiler and its descendants (Meta Platforms 2022f). If memoization is benefitted from, the parameter's value should decrease significantly in update phases, since descendants should only re-render when their props change (Meta Platforms 2022f). On the other hand, baseDuration estimates a worst-case render time without memoization (Meta Platforms 2022f). Comparing the two parameters can be useful to determine the impact of used optimization methods.

```
  const onRenderCallback = (

    id, // the "id" prop of the Profiler tree that has just committed
    phase, // either "mount" (if the tree just mounted) or "update" (if
it re-rendered)
    actualDuration, // time spent rendering the committed update
    baseDuration, // estimated time to render the entire subtree without
memoization
    startTime, // when React began rendering this update
    commitTime, // when React committed this update
    interactions // the Set of interactions belonging to this update
  ) => {
    // Aggregate or log render timings...
  };

  return (
    <App>
      <Profiler id="Navigation" onRender={onRenderCallback}>
        <Navigation />
      </Profiler>
      <Main />
    </App>
  );
```
Code example 1. Profiling Navigation component in React.js (Meta Platforms 2022f).

4 Decreasing Re-renders

Virtual Document Object Model (VDOM) is a general concept to reduce the amount of DOM operations through virtual representations of the UI, which has been studied to benefit performance in complex Single-Page Applications (Chęć & Nowak 2018). React.js includes an implementation of the VDOM, where two representations of the DOM tree are stored to memory. The implementation includes a heuristic algorithm for the detecting differences in the VDOM's, which is efficient with a complexity of O(n) (Meta Platforms 2022d). Nevertheless, actions can be taken to aid the process further and decrease the number of unnecessary re-renders, and as such, the amount of DOM operations.

For example, React.js keeps track of component's state and property variables. In case of any changes, the "diffing" algorithm is used to deduce whether to re-render the component. If re-rendering is needed the component's children are also checked. However, it is argued that this type of iterative comparison may cause performance issues in complex applications with large numbers of components and thus additional measures are suggested. (Javeed 2019)

Firstly, the amount of state and prop variables should be reduced whenever possible. Combined with avoiding frequent and unnecessary changes to the variables, this leads to fewer comparisons and re-renders. Secondly, the main component should be split into independent components, which handle their own state. Thus, smaller parts of the DOM tree need to be re-rendered when the state of the main component is updated. (Javeed 2019).

Furthermore, a linter such as ESLint may assist to follow the best practices of state management in React.js. For example, using unstable keys while mapping a list of content in JSX leads to unnecessary recreation of the components in each re-render. Whether an error occurs due to a human error or a gap in knowledge, code linting can automatically alert developers. The recommended ESLint plugins are eslint-plugin-react and eslint-plugin-react-hooks, which include React.js specific linting rules.

5 Memoization

The form of caching discussed in this chapter is referred to as *application-level caching* or *memoization*. The central principle of memoization is to store the results of computations in the application level for later use. The results are stored by the computation parameters. If a result is found by given parameters, it is returned; otherwise, the computation is performed, and it is cached. (Mertz, Nunes, Della Toffola, Selakovic, & Pradel 2021). Research shows that 13% of JavaScript performance issues were caused by repeated operations (Mertz et al. 2021 [Selakovic & Pradel 2016]). Recomputing is avoided by memoization, leading to significant performance gains (Mertz et al. 2021).

Memoization is discussed in relation to database queries in Mertz et al. (2021) research. The technique can also be applied in front-end in the form of storing computation results in the browser memory. More specifically, memoization can be utilized to avoid recomputations of compute-intensive tasks and re-renders occurring due to changes in object references. React.js includes a few *Hooks*, functions that provide React.js state and lifecycle methods for function components, for implementing application-level caching. The memoization Hooks include memo, useCallback, and useMemo, which are overviewed in this chapter. As a general guideline, caching expensive and frequently requested computations lead to higher improvements (Mertz et al. 2021).

Memo, one of React.js Hooks, can be used to memoize component result and thus should only be used if the component renders the same results given same props. With memo, the component is only re-rendered when its props or state change. Furthermore, shallow comparison of the props is used to determine changes. If objects are used as props, a custom comparison function should be provided, as in Code example 2. (Meta Platforms 2022c).

```
const MyComponent = (props) => {
  /* render using props */
};
const areEqual = (prevProps, nextProps) => {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
};
export default React.memo(MyComponent, areEqual);
```

Code example 2. Memoized component with a custom comparison function (Meta Platforms 2022c).

UseCallback memoizes a callback function to avoid function redefinitions on each component re-render (Meta Platforms 2022e). UseCallback is recommended to be used whenever a function is passed to a memoized component, or a function is passed as an effect argument. For example, useCallback is often used with useEffect. UseEffect is run only whenever the dependencies change and useCallback can prevent infinite loops resulting from the re-creation of a function used in useEffect. Moreover, useCallback should be used when components are defined inside a component (Code example 3) (Perkkiö 2021). Without useCallback, the reference of the component changes on each re-render, leading to performance losses and possible bugs relating to unstable component state (Perkkiö 2021).

```
const Component = () => {
  const MemoizedNestedComponent = React.useCallback(() => <div />, []);

  return (
    <div>
      <MemoizedNestedComponent />
    </div>
  );
};
```

Code example 3. Memoized nested component, with an empty dependency array (Perkkiö 2021).

Instead of memoizing a callback function, UseMemo memoizes a return value of a function (Code example 4) (Meta Platforms 2022e). UseMemo is recommended to be used with performance-wise expensive computations with a large data size, and thus, one use case could be caching the results of a search.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Code example 4. Memoization of the result value of a function (Meta Platforms 2022e).

Deciding what to cache can be time consuming, which is why automation of caching has been proposed (Mertz & Nunes 2018). However, no popular implementations exist for React.js yet. Furthermore, the way memoization is implemented in React.js is not described in depth in the documentation for developers. For example, it is unclear what is the time to live (TTL), meaning expiration time, of the different memoization techniques. Thus, it is not apparent when exactly memoization techniques should be used for the highest performance gains.

6 Decreasing Load Times

Load times are important for user experience. Especially in non-mobile browsing context, long *page loading time* (PLT) has a significant negative effect on user's Quality of Experience (QoE) (Barakovic & Skorin-Kapov 2015). For clarification, PLT in SPAs is considered as the initial loading time when first visiting the application, not any subsequent re-renders on the page. In other words, routing to a different page view would be considered as loading time within the application, not PLT. The following subchapters examine tools and techniques to decrease loading times. Webpack can be utilized to decrease PLT, while resource preloading is a technique to decrease loading times within the application.

6.1 Webpack

The PLT of a web application is affected by the amount and size of the loaded files (Pavic & Brkic 2021). The files include the front-end code and its dependencies, meaning any imported JavaScript libraries. While Webpack, a module bundler, has multiple other use cases, one of its benefits is reduced PLT. This chapter overviews the Webpack's methods to achieve faster loading through code bundling, code splitting, tree shaking, and image compression.

In the basic use case, Webpack is used to compile all the JavaScript code, CSS files, fonts, and images into one file, called a *bundle*. Code bundling reduces the number of loaded files, in turn requiring less requests to display the application. Nevertheless, extensive use of large JavaScript libraries can still contribute to long loading times due to the large file size. (Bouzid 2020, Pavic & Brkic 2021). In complex web applications, React.js' ecosystem often includes multiple JavaScript libraries, for example to handle request sending, application state management and routing. The heavy architecture leads to a longer PLT if all of the code is bundled into one file.

Code-splitting refers to generating multiple smaller bundles to be loaded on demand, reducing the amount of code to be loaded at a time and as such the PLT. Code-splitting is supported by Create React App's default Webpack configuration. Additionally, code-splitting can be achieved by a custom configuration. In React.js, modules can be loaded on demand by dynamic imports, which return a promise. Furthermore, React.lazy-function enables rendering dynamic imports as regular components. Usually fallback content, such as a loading animation, is provided while the component bundle is being fetched as in Code example 5. When React.lazy or dynamic import is used, Webpack code-splits the application into multiple bundles by default. (Meta Platforms 2022a)

```
import React, { Suspense } from "react";

const OtherComponent = React.lazy(() => import("./OtherComponent"));

const Component = () => (
  // Show a spinner while the profile is loading
  <Suspense fallback={<Spinner />}>
    <OtherComponent />
  </Suspense>
);
```

Code example 5. Dynamically loaded component, which renders a spinner while loading. (Meta Platforms 2022a)

Tree shaking reduces the bundle sizes by eliminating *dead code*, meaning unused code. Dead code is determined through unused function exports, if ES2015 module syntax and the production mode of the Webpack is used. However, "sideEffects" property needs to be defined in the package.json in order for Webpack to actually drop unused code from the created bundles. The value can be set to false if the codebase does not include *side effects*, defined as code performing a special behaviour when imported. Otherwise, an array of file names should be provided. For example, if css-loader-plugin is used to import

CSS-files, “*.css” should be added to the array to avoid unintentional code elimination. Furthermore, Webpack’s production mode will *minify* source code, that is, convert it to a smaller form, by default. (Webpack 2022)

Lastly, unnecessary large image sizes or resolutions have a considerably negative impact on the loading times of a web application. Image size is an important factor to consider, since even images taken with modern mobile phones can exceed 10 megabytes. (Pavic & Brkic 2021). Create React App does not include image optimization by default. If a web application uses images, the process of compressing large images can be automated by an image optimizer library or a custom Webpack configuration with an image optimizer plugin.

6.2 Resource Preloading

Resource preloading is a technique, where resources are fetched before use, leading to lower loading times. Resource preloading does not decrease PLT, instead affecting the loading times within the application. (Yeo, Rim, Shin & Moon 2020). For example, decreasing an application’s waiting time to see a detailed graph about previous data will create a more fluid and enjoyable user experience.

There are no obvious built-in methods for resource preloading in React.js. However, it is planned that Suspense, a component for displaying different content while loading, will be enhanced to handle asynchronous operations, such as preloading resources (Meta Platforms 2022b). Nonetheless, the technique could already be applied with some consideration: fetching could be started as part of an event handler, instead of loading resources in useEffect Hook once the component has started to render. For example, consider that an application has a link to another view loading new data. The link could be changed to a button with onClick() event handler starting the fetching of the required data and navigating the user to the correct view. However, this would require global state management, such as React Redux, for the server request results to be used in the navigated component.

7 Multithreading

In this chapter multithreading is examined from different points of view. First, the operation of HTML5 multithreading is overviewed. Secondly, different types of workers are examined to gain understanding of different use cases. Next, the problem of estimating the optimal number of workers is discussed. Lastly, code examples of multithreading in React.js are introduced.

HTML5 standard introduced multithreading to JavaScript web applications with the Web Workers API. With the API, code can be run concurrently in multiple threads, called workers. Communication with the worker and the main thread, responsible for modifying the user interface, operates through passed messages. Two types of web workers exist: dedicated workers and shared workers. Dedicated workers are accessible to only the script that created it, whereas shared workers can be accessed from any script from the same domain. (Verdu & Pajuelo 2016).

Verdu and Pajuelo (2016) introduce further Web Workers classifications based on the intended use: single worker, multiple asynchronous workers, and multiple synchronous workers. A single worker can be used to conduct all compute-intensive tasks. For example, in HTML5 videogames single worker is used for physics and artificial intelligence. When multiple workers are needed, the synchronicity needs to be considered. Multiple asynchronous workers should be used for large or continuous workloads, where tasks are not related to each other. For example, spell checking consists of independent tasks, and it can be implemented with multiple asynchronous workers. Lastly, multiple synchronous workers can be used for highly parallel code, where tasks can be split to multiple jobs. The key aspect with synchronous workers is that tasks are dependent from each other, and workers may be idle at times. An example of a synchronous task is video processing, where workers need to wait for completion of other jobs until moving to a new frame.

Javeed (2019) emphasizes that creating and stopping Web Workers takes resources. Thus, they should be utilized for computationally intensive operations with large data sizes. In fact, running computationally intensive operations on the main thread can block the UI leaving it unresponsive to the user. In these cases, performance gains of concurrency and the UX benefits of responsive UI outweigh the disadvantages of creating Web Workers.

Regarding the number of workers, Web Workers API does not include dynamic scaling according to any performance metrics. Unlike other approaches, creating threads is left to the developer. Furthermore, estimating the number of needed workers is difficult even with knowledge about the number of logical cores of the computer, since users may execute other applications in the background. An overestimate could overload the system, degrading the performance of all applications. Thread pool management could be considered in cases where a large number of threads is needed to dynamically adapt the number of web workers to the available resources. (Verdú, Costa & Pajuelo 2016) However, the implementation of this as well is left to the developers as React.js does not include such features.

Lastly, simple code examples, without details about calculations, are presented to understand the Web Workers API in React.js environment. The API is quite straightforward: communication between the main thread and the web worker is handled with postMessage() and onmessage() [sic] methods. In the main thread the methods are called through the Worker interface whereas in the worker script the worker is in the global scope (MDN Web Docs 2022). Furthermore, web worker files are recommended to be stored in the “public” folder, to avoid bundling them with other code in the building process.

First, a web worker is created (Code example 6) in the main thread and the file name of the worker is specified. Then a message is passed to the worker through postMessage(). The message parameter accepts multiple different types, such as booleans, strings, dates, objects, arrays, sets and maps. However, if it is required, null or undefined can be passed. The worker script should contain a function for the onmessage handler (Code example 7). It handles the calculations and passes the result to the main thread through postMessage(). As a note, the sent messages are available in the message event’s data property, which is accessed through ES6 object destructuring in the examples. (MDN Web Docs 2022)

```
  const calculateIntensiveTask = () => {
    const worker = new Worker("filename.js");
    worker.postMessage(message);

    worker.onmessage = ({ data }) => {
      setSomeState(data);
    };
  };
```

Code example 6. Use of Web Workers API in a component’s function.

```
onmessage = ({ data }) => {
  const result = calculate(data);
  postMessage(result);
};
```

Code example 7. Part of web worker script handling Web Worker API message passing.

8 Summary

Good performance can benefit the perceived user experience of a web application. The aim of this paper was to examine the techniques, methods, and tools that can be utilized in React.js to increase the performance of web applications and thus, their market competitiveness.

The results indicated that decreasing the number of re-renders and loading times as well as utilizing memoization and multithreading can benefit the performance of web applications. Since React.js provides a simple API for many of these techniques, most are easily accessible for developers. Additionally, it is important to acknowledge to which extent the optimization methods are analysed in studies measuring the performance of web frameworks. For example, studies comparing React.js and other front-end frameworks could be affected by how much effort has been invested in their performance optimization. Despite this, the performance benefits of presented solutions in real-world use cases are not predictable; therefore, performance profiling through React.js Developer Tools Profiler and Profiler component should be considered to ensure that optimization benefits truly exist. Further research on the topic could explore measuring the extent of performance benefits attained through a combination of optimization techniques, methods, and tools, instead of employing only one.

References

Barakovic, S. & Skorin-Kapov, L. (2015). *Multidimensional modelling of quality of experience for mobile Web browsing*. Computers in Human Behavior, 50, 314–332. https://doi.org/10.1016/j.chb.2015.03.071

Chęć, D. & Nowak, Z. (2018). *The Performance Analysis of Web Applications Based on Virtual DOM and Reactive User Interfaces*. In Advances in intelligent systems and computing (Vol. 830, pp. 119–134). Springer International Publishing. https://doi.org/10.1007/978-3-319-99617-2_8

Vaughn, B. (2018). *Introducing the React Profiler*. Accessed 25.3.2022. https://reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html

Bouzid, M. (2020). *Webpack for Beginners: Your Step-By-Step Guide to Learning Webpack 4*. Apress L. P.

Google Developers (2021). *Lighthouse performance scoring*. Accessed 25.3.2022. https://web.dev/performance-scoring/

Griffiths, D. & Griffiths, D. (2021). *React Cookbook (1st edition)*. O'Reilly Media, Inc.

Javeed, A. (2019). *Performance Optimization Techniques for ReactJS*. IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), 1–5. https://doi.org/10.1109/ICECCT.2019.8869134

MDN Web Docs (2022). *Worker*. Accessed 19.3.2022. https://developer.mozilla.org/en-US/docs/Web/API/Worker

Meta Platforms (2022a). *Code-Splitting*. Accessed 15.3.2022. https://reactjs.org/docs/code-splitting.html

Meta Platforms (2022b). *React v18.0.* Accessed 19.4.2022. https://reactjs.org/blog/2022/03/29/react-v18.html

Meta Platforms (2022c). *React.memo*. Accessed 22.2.2022. https://reactjs.org/docs/react-api.html#reactmemo

Meta Platforms (2022d). *Reconciliation*. Accessed 19.4.2022. https://reactjs.org/docs/reconciliation.html

Meta Platforms (2022e). *Hooks API reference.* Accessed 22.2.2022. https://reactjs.org/docs/hooks-reference.html

Meta Platforms (2022f). *Profiler API.* Accessed 31.3.2022. https://reactjs.org/docs/profiler.html#gatsby-focus-wrapper

Mertz, J. & Nunes, I. (2018). Automation of application-level caching in a seamless way. Software, Practice & Experience, 48(6), 1218–1237. https://doi.org/10.1002/spe.2571

Mertz, J., Nunes, I., Della Toffola, L., Selakovic, M. & Pradel, M. (2021). *Satisfying Increasing Performance Requirements With Caching at the Application Level*. IEEE Software, 38(3), 87–95. https://doi.org/10.1109/MS.2020.3033508

Park, H., Cha, M. & Moon, S.-M. (2016). *Concurrent JavaScript Parsing for Faster Loading of Web Apps*. ACM Transactions on Architecture and Code Optimization, 13(4), 1–24. https://doi.org/10.1145/3004281

Pavic, F. & Brkic, L. (2021). *Methods of Improving and Optimizing React Web-applications*. 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), 1753–1758. https://doi.org/10.23919/MIPRO52101.2021.9596762

Perkkiö, A. (2021). *Prevent creating unstable components inside components*. Accessed 12.4.2022. https://github.com/yannickcr/eslint-plugin-react/blob/master/docs/rules/no-unstable-nested-components.md

Selakovic, M. & Pradel, M. (2016). *Performance issues and optimizations in JavaScript*. Proc. 38th Int. Conf. Softw. Eng. (ICSE '16), New York:ACM Press, pp. 61-72.

Verdú, Costa, J. J., & Pajuelo, A. (2016). *Dynamic web worker pool management for highly parallel javascript web applications*. Concurrency and Computation, 28(13), 3525–3539. https://doi.org/10.1002/cpe.3739

Verdu, J. & Pajuelo, A. (2016). *Performance Scalability Analysis of JavaScript Applications with Web Workers*. IEEE Computer Architecture Letters, 15(2), 105–108. https://doi.org/10.1109/LCA.2015.2494585

Webpack (2022). *Tree Shaking*. Accessed 14.3.2022. https://webpack.js.org/guides/tree-shaking/

Yeo, J., Rim, J.-H., Shin, C., & Moon, S.-M. (2020). *Accelerating Web Start-up with Resource Preloading*. In Web Engineering (Vol. 12128, pp. 37–52). Springer International Publishing. https://doi.org/10.1007/978-3-030-50578-3_4