

# **A Detailed Look at the Internal Architecture and Profiling of React.js**



Bachelor thesis

Degree Programme in Computer Applications  
spring, 2024

Cihan Erenler

## ABSTRACT

Many developers use a well-known front-end library called React.js to construct online applications that are both dynamic and responsive. This thesis provides a comprehensive overview of React.js and its internal workings, emphasizing the problems it solves and the best practices for profiling React.js applications.

The first chapter delves into how React.js handles common front-end development problems, such as state management, rendering performance, and component reusability. The inner workings of React.js is investigated in the second portion of this thesis. Topics covered include the framework's architecture, component model, virtual DOM, event handling, state management, and rendering. In the concluding chapter of the thesis, the significance of profiling web applications is covered. This chapter also offers an overview of some profiling tools for React.js, such as Chrome DevTools and React Profiler. This section also explains how to identify performance problems in React applications. The case studies and instances of real-world applications presented throughout the thesis illustrate how the concepts can be applied in real life.

The thesis is practice based and offers helpful information for developers, businesses, and educators interested in front-end development who want to gain a deeper understanding of React.js and how to use it effectively to create high-performing web applications. The thesis is helpful because it offers insightful information that these people can use.

## Glossary

HTML	HyperText Markup Language for web pages
DOM	Document Object Model
VDOM	Virtual DOM
JSX	JavaScript XML
UI	User Interface
URL	Uniform Resource Locator

## Contents

1	Introduction .....	1
2	Methodology.....	2
3	Knowledge base of performance testing.....	3
3.1	Code duplication in conventional web development .....	3
3.2	Performance issues with conventional web development.....	4
3.3	Identifying errors in conventional web development.....	5
3.4	A brief overview of the Document Object Model .....	5
3.5	Virtual DOM in React.js .....	6
3.6	Difficulties in managing states .....	6
3.6.1	State management in conventional web development.....	7
3.6.2	Managing states in React.js .....	7
4	Inner workings of React.js.....	8
4.1	Virtual DOM.....	8
4.2	Creating React elements with JavaScript .....	9
4.3	JSX.....	10
4.4	React components and props.....	11
4.5	React state .....	12
4.6	Unidirectional data flow .....	12
4.7	React.js lifecycle methods .....	12
4.7.1	useEffect .....	13
4.7.2	Update .....	14
4.7.3	Unmount.....	15
5	Profiling React.js application.....	17
5.1	React Developer Tools.....	17
5.2	Reactotron .....	17
5.3	Chrome DevTools .....	17
5.4	Profiler API.....	18
6	Case study .....	19
6.1	Demo applications.....	19
6.2	Component render time.....	21
6.3	Component performance .....	24

6.4	Profiling memory usage.....	27
6.5	Network performance .....	33
7	Summary .....	37
8	References .....	39
9	Material management plan .....	41

## Figures, program codes, commands

Figure 1.	DOM diagram.....	6
Figure 2.	Virtual DOM diagram. ....	9
Figure 3.	React lifecycle methods. ....	13
Figure 4.	Demo application's home page. ....	19
Figure 5.	Demo application's products page. ....	20
Figure 6.	Demo application single product page. ....	20
Figure 7.	Demo application people list. ....	21
Figure 8.	Render times on browser's console.....	22
Figure 9.	Demo application render time.....	23
Figure 10.	Demo application render time after updating the state. ....	24
Figure 11.	React Developer Tools Profiler. ....	25
Figure 12.	Recording actions in React Developer Tools Profiler tab. ....	25
Figure 13.	Results in React Developer Tools Profiler tab.....	26
Figure 14.	Heap snapshot 1. ....	29
Figure 15.	Heap snapshot 2. ....	30
Figure 16.	Heap snapshot 3. ....	30
Figure 17.	Heap snapshot comparison. ....	31
Figure 18.	Heap snapshot objects.....	31
Figure 19.	Heap snapshot problem code.....	32
Figure 20.	Network profiling of the demo application. ....	34
Figure 21.	Network tab recording requests.....	35
Figure 22.	Network request recording details.....	35

Program code 1. React.js solution to the code duplication.....	4
Program code 2. Virtual DOM demo code example. ....	8
Program code 3. Creating element in React.js without JSX.....	10
Program code 4. Creating nested elements in React.js without JSX. ....	10
Program code 5. Creating nested elements in React.js using JSX.....	11
Program code 6. React.js useState code example. ....	14
Program code 7. Code example that demonstrates update lifecycle method.....	15
Program code 8. Code example that demonstrates unmount lifecycle method. ....	15
Program code 9. Importing Profiler API.....	21
Program code 10. Measuring component render time in Profiler API.....	22
Program code 11. Component code that causing the memory leak. ....	29
Program code 12. Updated version of the component that causing memory leak. ....	33

## 1 Introduction

Multiple options are available in the modern world when creating a web application. Technological advancements have enabled programmers to create more sophisticated and user-friendly applications. On the other hand, the excessive increase in complexity throughout development led to brand-new performance and development problems. Developers have created new frameworks and libraries to address those issues, and React.js is among them. The purpose of frameworks and libraries is to simplify and speed up the development process by preventing us from wasting significant time and code. In addition to this, they provide developer-friendly APIs that can manage very complex tasks and operate in an optimized manner. Most of the time, a website is no longer considered a collection of online pages. Instead, web apps are created that may have only one page, and that page serves as a container for the web application rather than the layout for the content. Unsurprisingly, a single-page application of this kind is called a single-page application.

The purpose of this thesis is to provide an overview of what problems React.js solves, how it does so, how it operates internally, and how someone can profile React.js applications to identify underlying performance issues. After identification, optimizing the application is beyond the scope of this thesis. This thesis can serve as a foundation, and some other person can follow up with additional research on optimization. Developers, businesses, and educators who want to gain a deeper understanding of React.js and how to create high-performing web applications will find this research useful.

Research questions are:

- What problems does React.js solve?
- How does React.js internally function?
- How can a React application be profiled?

## 2 Methodology

This section describes the methodology used to collect and analyze information for the profiling of React applications. Profiling React applications involve examining and analyzing their performance, code structure, and different metrics. The approach used here is divided into various steps, each of which is meant to assure the gathering of accurate and useful data.

- **Literature Review:** The research method began with an extensive review of the literature. A thorough search of several scholarly databases has been carried out. The purpose of this literature research was to provide an overview of React, web application profiling, and related concepts.
- **Identification of Key Frameworks and Documentation:** The documentation and resources offered by the React team were chosen as main sources for learning about React and its profiling methods. To fully understand the suggested methods and tools, the official React documentation, as well as any extra content, was reviewed and analyzed.
- **Data Collecting and Paraphrasing:** Collected information from the literature and online resources was carefully analyzed. Key findings, concepts, and methods were paraphrased to ensure that the terminology and context aligned with the research objectives. This step was conducted using partly natural language processing tools to produce clear content.
- **Content Validation:** The paraphrased content was validated through cross-referencing with the original sources. It was essential to maintain accuracy and context consistency throughout the paraphrased material.
- **Content Validation:** Depending on the research scope and objectives, case studies involving real-world React applications may be conducted to demonstrate the practical application of the profiling methodology.



### 3 Knowledge base of performance testing

This chapter explores the solutions React.js offers to the challenges developers encounter in traditional web development.

#### 3.1 Code duplication in conventional web development

Code duplication is a main problem in traditional web development. It occurs when developers must write identical code for various components or pages. The approach above has the potential to result in expanded codebases, elevated maintenance expenses, and reduced efficiency. Moreover, multiple instances of the same code can pose a challenge when attempting to implement modifications or fix errors.

Code duplication is common in traditional web development because of the characteristics of HTML, CSS, and JavaScript. The development of HTML and CSS require manually creating markups for individual elements and stylings. Similarly, using JavaScript requires the creation of different scripts by developers for every function or interaction. As a result, developers often end up writing similar code across multiple pages or components.

Modern web development frameworks like React.js provide a component-based structure to solve this problem. This allows developers to generate reusable components that can be used across various pages or applications. Implementing this methodology results in a decrease in the repetition of code and an increase in the potential for code reuse, ultimately resulting in more streamlined and sustainable codebases. (React, n.b-d)

When the most basic case considered, where two pages have the same layout (including the header and footer). In traditional web development, individual HTML and CSS files are created for each page. In React.js, reusable components can be created to prevent code redundancy. Demonstration of this approach can be seen in Program code 1.

Program code 1. React.js solution to the code duplication.

```
import React from "react";
import Header from "./Header";
import Footer from "./Footer";

function Homepage() {
  return (
    <>
      <Header />
      <main>{/* Main content goes here */}</main>
      <Footer />
    </>
  );
}
export default Homepage;
```

It is possible to create a Header and a Footer component that can be used repeatedly on various pages.

### 3.2 Performance issues with conventional web development

The manipulation of the DOM for UI updates is a common cause of performance problems in traditional web development. The traditional methodology demands a refresh of the entire UI for minor changes, leading to slower rendering and unresponsive user interfaces. (MDN, 2023-b)

React.js addresses performance concerns by utilizing its Virtual DOM(The author will elaborate further on this topic in the subsequent chapter), representing the real DOM stored in memory. React.js updates the Virtual DOM initially when a component's state or props changes and later determines the minimum number of changes necessary to update the DOM. The methodology above minimizes the frequency of DOM manipulation, leading to quicker rendering and heightened user interface responsiveness. (React, n.d.-g)

### 3.3 Identifying errors in conventional web development

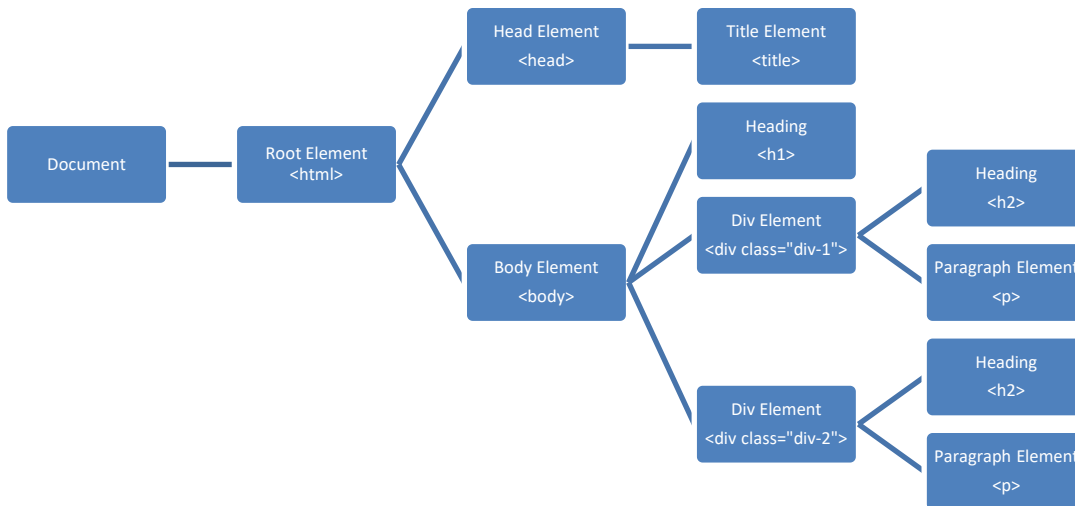
Identifying and debugging errors in traditional web development can pose a challenge due to the code's tightly coupled nature. Identifying the root cause of an error can be challenging, as modifications made to a specific segment of the code may result in unintentional effects in other areas. React.js addresses this problem via its component-based nature. React applications use self-contained components, each with a clearly defined purpose. This allows identifying and resolving errors as modifications to a particular component do not impact the other remaining components. (React, n.d.-b)

Consider a standard web application that consists of a form with multiple input fields. The data is sent to the server for processing once the form is submitted. If an error occurs throughout the procedure, identifying the cause of the issue may be difficult. In a component-based application, the form may be divided into separate parts for specific input fields. This makes it easier to identify the exact component that produced the error and resolve the problem.

### 3.4 A brief overview of the Document Object Model

The Document Object Model is a programming interface that allows web developers to interact with the structure, style, and content of web documents such as HTML, XML, and SVG. The document is represented as a hierarchical tree-like structure consisting of nodes and objects. Each node in the structure corresponds to a document element, property, or text. The Document Object Model provides a set of characteristics, methods, and events that programmers may use to access and modify the document. Appending or deleting nodes, altering properties or styles, and controlling user interactions are all examples of this. Figure 1 illustrates the tree-like structure of nodes that comprise the DOM. (MDN, 2023-a)

Figure 1. DOM diagram.



This tree-like structure is likely to be more deeply layered and complex in real-world applications.

### 3.5 Virtual DOM in React.js

React provides a lightweight DOM. The code in question does not interact with DOM generated by the browser, but rather with DOM stored in the system's memory. This results in a very efficient and robust application performance. Most other web development frameworks interact directly with the browser's DOM, resulting in a change of the whole DOM tree on each page event trigger.(React, n.d.-g)

### 3.6 Difficulties in managing states

User input, session data, server responses, and other comparable data types can all be considered dynamic information that changes over time. The traditional approach to state management involves the use of server-side technologies such as PHP and ASP.NET to store and retrieve state data as required.

### 3.6.1 State management in conventional web development

The downsides of server-side state management have become more obvious as web application complexity and interaction have increased. The significant overhead associated with server inquiries, which lowers application performance and responsiveness. Additionally, server-side state management may need to be updated over time to provide a consistent user experience across all devices and platforms.

To address these issues, some recent web frameworks, such as React, use a client-side approach to state management. Currently, state data is kept and processed inside the boundaries of the user's browser, utilizing technologies such as JavaScript and local storage. This method provides for quicker response times and better scalability by dividing work between the server and the client. The server just provides static assets and data, while the client is responsible for all state-related actions.

### 3.6.2 Managing states in React.js

Managing the state is one of the most critical parts of developing React application. React's built-in tools for managing state, like the `useState` and `useEffect` hooks, make it easy to handle state within components. One of the benefits of handling state in React is that it lets us separate and modularize components, which makes it easier to reuse code and makes the program easier to keep up to date. With React's component-based design and state management features, developers can make flexible, scalable, and easy-to-keep apps. Also, how React handles states can improve the speed of a program. State management methods like those in React can make an application run much faster by lowering the number of times components need to be re-rendered. (React, n.d.-d)

## 4 Inner workings of React.js

To develop a single-page web application with improved performance, it is necessary to have a solid understanding of the inner workings of the React.js framework. In the following sections, the inner workings of React.js will be investigated in more detail.

### 4.1 Virtual DOM

React.js maintains the UI's visual representation in memory using the ReactDOM library. When executing a React.js application, it is not rendered immediately in the browser's DOM; instead, it is created in the computer's memory using ReactDOM. ReactDOM is responsible for interacting with the actual DOM, which means it is also responsible for what is seen on the screen. When state data is updated or modified, React.js updates the components that use this data. It determines if these components are rendering anything new. If this is the case, React will notify ReactDOM of these modifications so that ReactDOM can display them on the screen. (React, n.d.-g)

In Program code 2 represents a demo component.

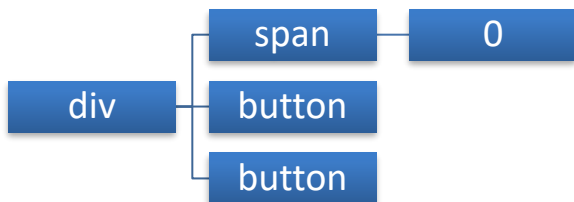
Program code 2. Virtual DOM demo code example.

```
const Demo = () => {
  const [state, setState] = useState(0);
  return (
    <div>
      <span>{state}</span>
      <button onClick={() => setState(state++)}>+</button>
      <button onClick={() => setState(state--)}>-</button>
    </div>
  );
};
```

This sample component displays a div element with two buttons and a span element. There is a state with an initial value of 0. The span element is responsible for displaying the state value, and

the state value is changed via buttons. The demo code would appear like Figure 2 on the browser's DOM.

Figure 2. Virtual DOM diagram.



The following situation would occur if React.js talked directly with the actual DOM: a user clicks the plus button. The status of the Demo component would be changed from 0 to 1. React.js reevaluates the DOM when a state changes. As a result, React.js would go over the Demo component and re-render all the Demo component's elements on the DOM. Only the span element's value has changed in this situation, but as a result, every component on the screen must be re-rendered, which is costly in terms of performance.

React.js provides a virtual version of DOM in the computer's memory rather than talking directly with the actual DOM. When a state changes, the virtual DOM component is the first to be changed. The virtual DOM is then compared to the actual DOM to find changes. Instead of updating the full DOM, this technique updates only the impacted sections. In Demo component, the span element is the only one that has been modified. For that reason, React.js is highly efficient and fast when compared to traditional DOM operations. (React, n.d.-g)

## 4.2 Creating React elements with JavaScript

The virtual DOM of React is a tree of React nodes, similar to how the DOM is a tree of nodes. In Program code 3, construct React nodes as follows:

Program code 3. Creating element in React.js without JSX.

```
React.createElement(type, props, children);
```

The starting point is a React object with the createElement function. This syntax may be used to build components, as seen in Program code 4:

Program code 4. Creating nested elements in React.js without JSX.

```
const element_1 = React.createElement("li", { className: "item-1" }, "Item 1");
const element_2 = React.createElement("li", { className: "item-2" }, "Item 2");
const element_3 = React.createElement("li", { className: "item-3" }, "Item 3");
const fragment = [element_1, element_2, element_3];
const list = React.createElement("ul", { className }, fragment);
```

It is easy to understand how the code might quickly become unmanageable in this manner. React has a more human readable solution for avoiding this confusion which is called JSX. JSX will be explained in more detail in the next section.

### 4.3 JSX

Constructing the virtual DOM by repeatedly invoking the React.createElement() method can make it challenging to visually represent these multiple functions as an HTML tag hierarchy. JSX is an optional syntax like HTML, enabling us to construct a virtual DOM tree without calling React.createElement(). (React, n.d.-c)

The same list component can be created using JSX as shown in Program code 5.



Program code 5. Creating nested elements in React.js using JSX.

```
var listOfItems = (  
  <ul>  
    <li className="item-1">item 1</li>  
    <li className="item-2">item 2</li>  
    <li className="item-3">item 3</li>  
  </ul>  
)  
);
```

Because of its straightforward design, JSX makes it much easier for developers to create React applications. So, when a developer writes a code like in Program code 5 and compiles it, React.js would execute the code like in Program code 4 under the hood.

#### 4.4 React components and props

When using unidirectional data flow and immutable data, component-based architecture is useful. At the beginning of an application building process, it is easy to think of each component as a separate unit. All interface components blend to create a single overall application that is so powerful that separating it down into separate components looks almost impossible. Consider the possibility of needing to build a spaceship. A rocket booster, a few wings, life support, and so on would be required. Consider how one could continue if one of the limitations was that each moving component of the spaceship had to be tested independently. Testing is the key difference between constructing a system as a whole and creating a large collection of little components. The component-based architecture has the advantage of allowing us to test each piece. Using components also allows us to break the user interface into separate, reusable elements and evaluate each separately. (Pitt, 2016)

Components are conceptually like JavaScript functions. They receive arbitrary inputs referred as props and return React.js components that describe what should be displayed on the screen. (React, n.d.-b)

## 4.5 React state

Components in React often need to manage and display data. When updating what's shown on the screen, a component must keep track of both the current data and the updated data. One approach is to use a variable to store this information and React.js does allow for it. However, the problem is that React.js cannot automatically keep track of the changes to this variable. So, the UI won't reflect the updates since React doesn't know when to trigger a re-render.

To address this issue, React.js provides the `useState` hook, which acts as the component's memory. When the data changes, the hook signals to React that a re-render is necessary. This mechanism ensures that the DOM gets updated, allowing us to see the latest version on the screen. (React, n.d.-f)

## 4.6 Unidirectional data flow

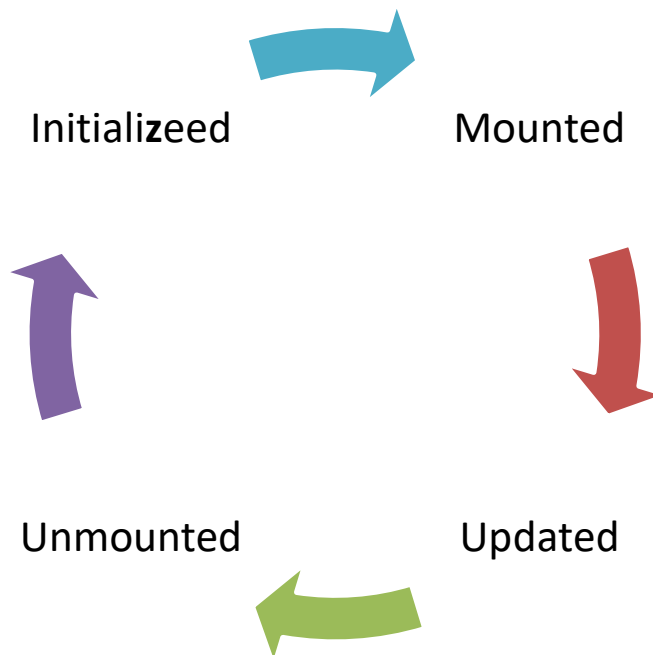
The concept of unidirectional data flow refers to the one-way transfer of data. React.js uses unidirectional data flow, which means the data can be passed only from parent to child components through props but not vice versa. Since React.js uses unidirectional data flow, it ensures that data passed from the parent component cannot be updated or altered from the child component, which provides a clean data flow between components. (Marttila, 2016)

## 4.7 React.js lifecycle methods

React has three main lifecycle categories: mounting, updating, and unmounting. When React renders a component for the first time, it is mounted. At this point, it calls a reliable function. Upon the initial rendering of a component instance, no updates occur. Upon the second rendering, a component undergoes an update process with each subsequent rendering. (React, n.d.-b)

Visual representation of this concept can be seen in Figure 3.

Figure 3. React lifecycle methods.



React has three main lifecycle categories: mounting, updating, and unmounting. When React renders a component for the first time, it is mounted. At this point, it calls a responsible function. Upon the initial rendering of a component instance, no updates occur. Upon the second rendering, a component undergoes an update process with each subsequent rendering. A component's unmounting phase occurs when a component disappears from the DOM. This scenario may occur in the event of DOM re-rendering that excludes the component or when the user navigates to a different page. (React, n.d.-b)

#### 4.7.1 useEffect

Upon the initial rendering of a function component, the useState hook runs. The practice of utilizing this method involves retrieving data and sending requests to a server. (React, n.d.-f)

In Program Code 6, the useEffect hook will run on the initial render.

Program code 6. React.js useState code example.

```
const DemoComponent = () => {  
  useEffect(() => {  
    // after the initial render this part will be called  
  }, []);  
  
  return (  
    <div>  
      <h1>Some title</h1>  
      <p>Some text</p>  
    </div>  
  );  
};
```

For instance, the `useEffect` section may be used to determine whether to fetch data from an external API or to begin a loading state. This section makes use of React's `useEffect` hook to manage side effects such as data fetching.

#### 4.7.2 Update

When the state value of a component is modified or updated, it triggers a re-render. Consequently, the component undergoes an update. The related method runs whenever this happens. In program code 7, the component has an index state. By clicking the following button, the state can be updated. When the state updates, the component gets re-rendered. (React, n.d.-f)

A conditional update also can be implemented by passing a dependency array to `useEffect` hook. For instance, in Program code 7 `index`'s state value is used as a dependency in `useEffect` hook. It means that whenever `index`'s state value is modified it will trigger a re-render and the logic that implemented in `useEffect` will run.

Program code 7. Code example that demonstrates update lifecycle method.

```
const DemoComponent = () => {
  const [index, setIndex] = useState(0)

  useEffect(() => {
    // this part will run every time the state changes
  }, [index]);

  return (
    <div>
      <span>{index}</span>
      <button onClick={() => setIndex(index++)}>next</button>
    </div>
  );
};
```

Now the logic of what would happen after the index updated can be implemented in useEffect section.

### 4.7.3 Unmount

When a component is removed from the DOM, it is unmounted. At this point, React.js calls the responsible method. Implementation can be seen in Program code 8. (React, n.d.-f)

Program code 8. Code example that demonstrates unmount lifecycle method.

```
const DemoComponent = () => {
  useEffect(() => {
    // this part will be called when the component mount
    return () => {
      // this part will be called when the component unmount
    };
  });
  return <div ref={divElement}>{/* Some content */}</div>;
};
```

This functionality is quite useful. For example, an event listener created for a particular component can be removed. This event listener is not needed when the component is unmounted.

## 5 Profiling React.js application

Improving application performance involves using techniques like application profiling. This method analyzes and measures how applications behave, offering insights into their workings and areas for enhancement. An essential part of application profiling is tracing, which tracks the execution flow within an application. By doing so, it helps pinpoint bottlenecks and performance issues, allowing for code optimization and overall performance improvement. The next section delves deeper into the profiling tools applied to React.js applications. (React Blog, n.d.)

### 5.1 React Developer Tools

The React Developer Tools is a browser extension that enables the inspection of component hierarchies and the viewing of component state and props. Additionally, the application features a profiler section designed to detect performance issues. (React, n.d.-e)

### 5.2 Reactotron

Reactotron is a software application used on macOS, Windows, and Linux operating systems to analyze and evaluate React JS and React Native applications. This software feature enables users to access their application state, display API requests and responses, conduct rapid performance evaluations, subscribe to specific segments of their application state, and show messages equivalent to the console. (Reactotron, n.d.)

### 5.3 Chrome DevTools

Google Chrome incorporates a set of tools called DevTools directly into its web browser for developers. DevTools empowers developers to design websites more efficiently by allowing real-time page modifications and quick identification of issues. Additionally, it can be utilized for profiling react applications and identifying performance issues. (Chrome DevTools, n.d.-a)

## 5.4 Profiler API

The Profiler API in React helps developers measure and assess the efficiency of their components. It makes it easier to identify which components are causing performance issues and determine the root cause of the problem. (React, n.d.-a)



## 6 Case study

This section of the thesis aims to develop a React.js application that demonstrates various profiling techniques using two different tools, namely React DevTools and the Profiler API.

### 6.1 Demo applications

The author created an e-commerce React.js application to demonstrate the profiling tools. An external backend API used during the development. This is a website of an imaginary furniture web store. For the state management author used Context API that provided by React.js and for styling author used a CSS library called Tailwind.css. This is a web application that consists of a landing page that showcases featured products. The visual representation of this page can be seen in Figure 4

Figure 4. Demo application's home page.

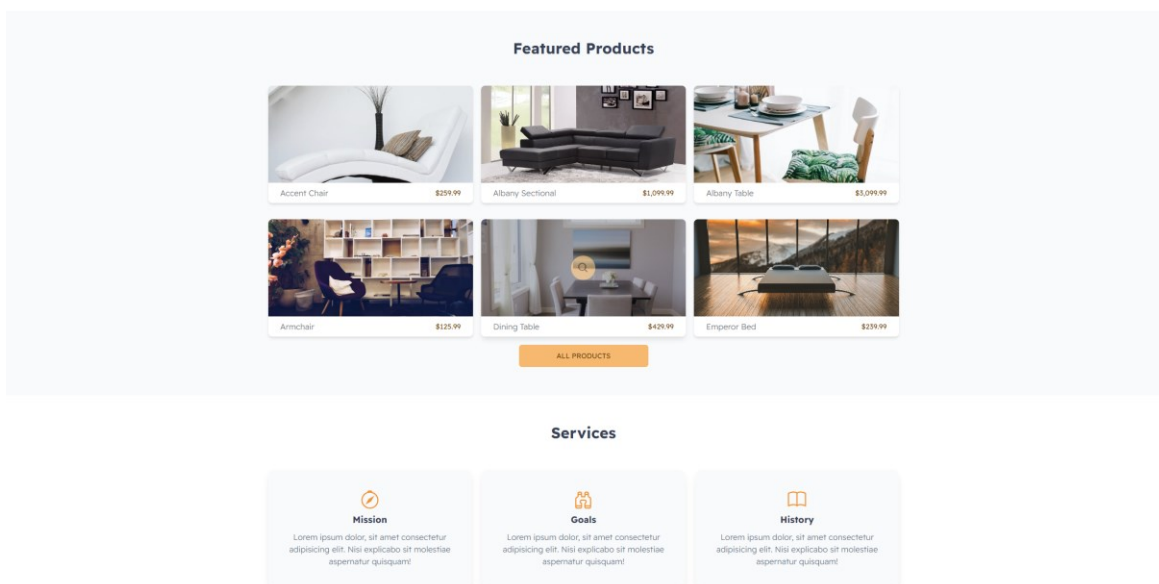


Figure 5 displays all products page that displays all available products including a filtering mechanism that allows filtering by category, company, color and price information.

Figure 5. Demo application's products page.

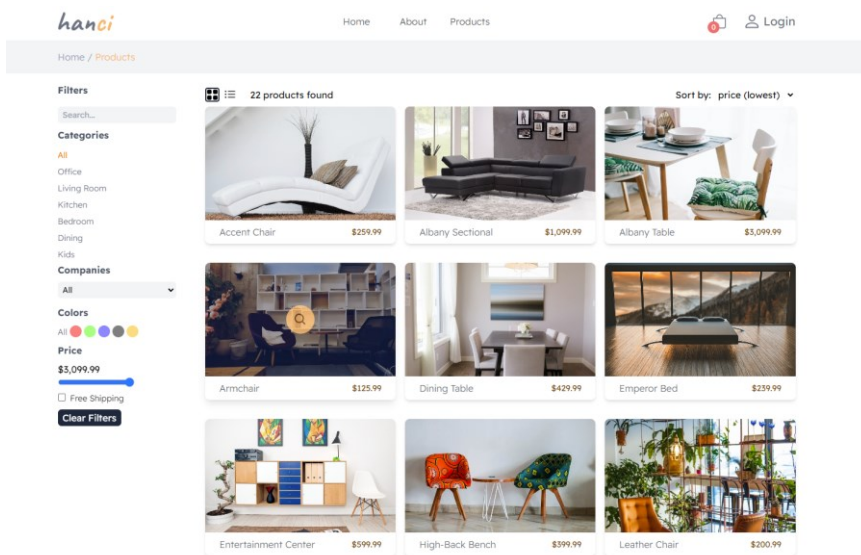
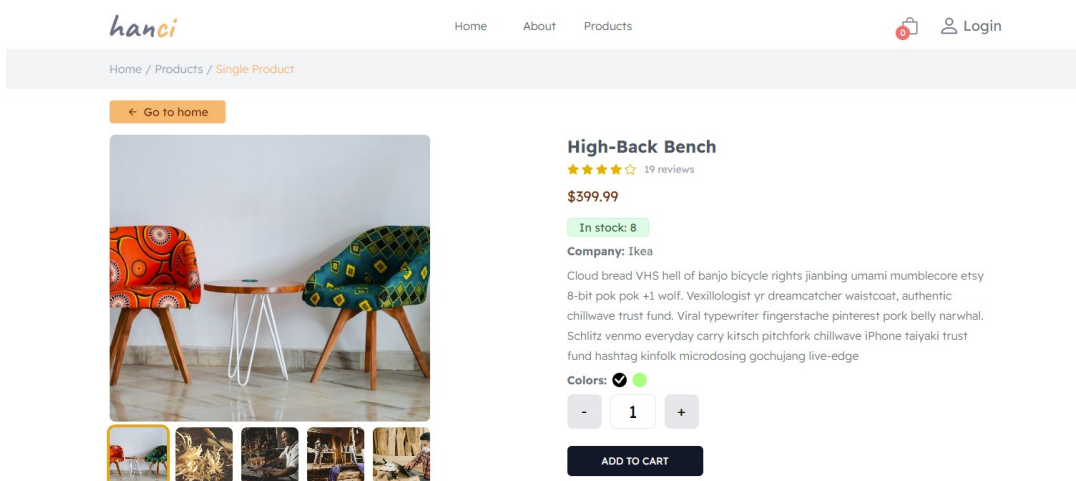


Figure 6 show the individual product page that displays detailed information about the product such as price, stock availability, description, and photo gallery.

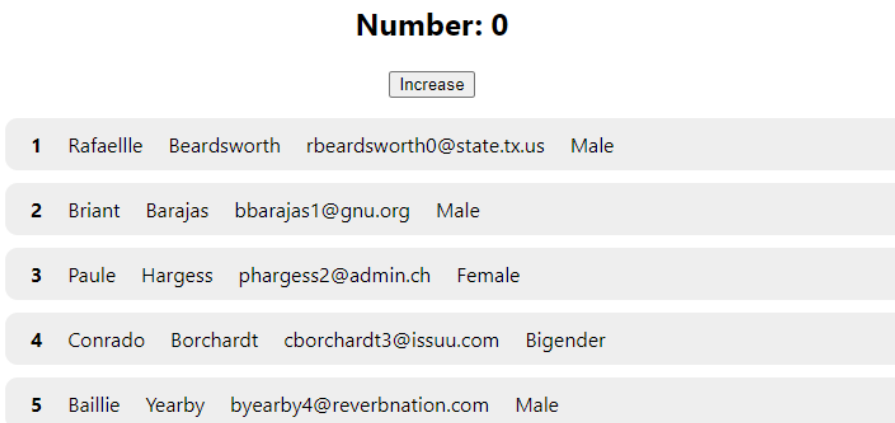
Figure 6. Demo application single product page.



The author has also created a second example application to illustrate potential bottlenecks in React applications. Using this method, performance concerns can be investigated, and various solutions can be considered. The application is a straightforward web page that displays a list of

people containing information about the person. It includes a number and a button to increase it, designed in a simple manner to keep things easy to follow and focus more on profiling. A visual representation of the application can be seen in Figure 7.

Figure 7. Demo application people list.



In this application 3000 list item used intentionally to create some bottleneck for testing purposes.

## 6.2 Component render time

The Profiler API can be utilized to obtain information about the render time of a component. Assuming there is a desire to determine the duration it takes to render the single product page; the Profiler component needs to be imported from the React library as follows as in Program code 9.

Program code 9. Importing Profiler API.

```
import { Profiler } from "react";
```

Then, the single-page component needs to be wrapped with the Profiler component. The Profiler component requires two props: an id and onRender. The id is used to identify the component being profiled, and onRender is a function that takes three arguments: id, phase, and actualTime. The id corresponds to the prop passed to the Profiler component, the phase indicates whether the

component is in an update or mount phase, and `actualTime` represents the time measured during the component's render. The code implementation can be seen in Program code 10.

Program code 10. Measuring component render time in Profiler API.

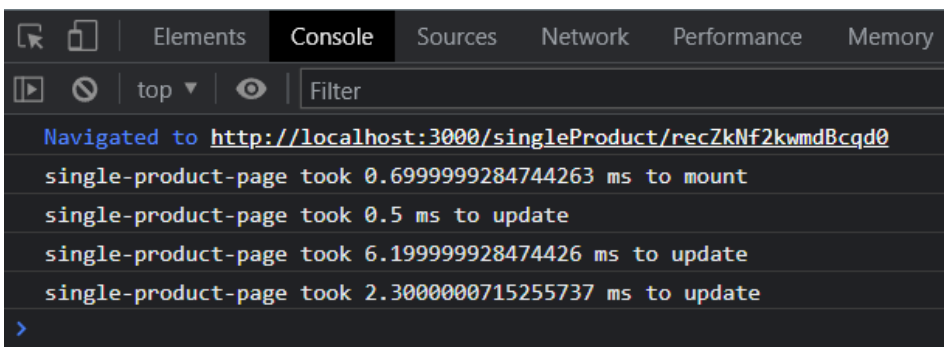
```

<Profiler
  id="single-product-page"
  onRender={({id, phase, actualTime}) => {
    console.log(`${id} took ${actualTime} ms to ${phase}`);
  }}
>
<SingleProduct>
</SingleProduct>
</Profiler>

```

When the page is refreshed the component's render information can be seen on the console as in Figure 8.

Figure 8. Render times on browser's console.



The screenshot shows the browser's developer console with the 'Console' tab selected. The address bar shows the URL `http://localhost:3000/singleProduct/recZkNf2kwmdBcq0`. The console output displays the following log messages:

```

Navigated to http://localhost:3000/singleProduct/recZkNf2kwmdBcq0
single-product-page took 0.6999999284744263 ms to mount
single-product-page took 0.5 ms to update
single-product-page took 6.199999928474426 ms to update
single-product-page took 2.3000000715255737 ms to update

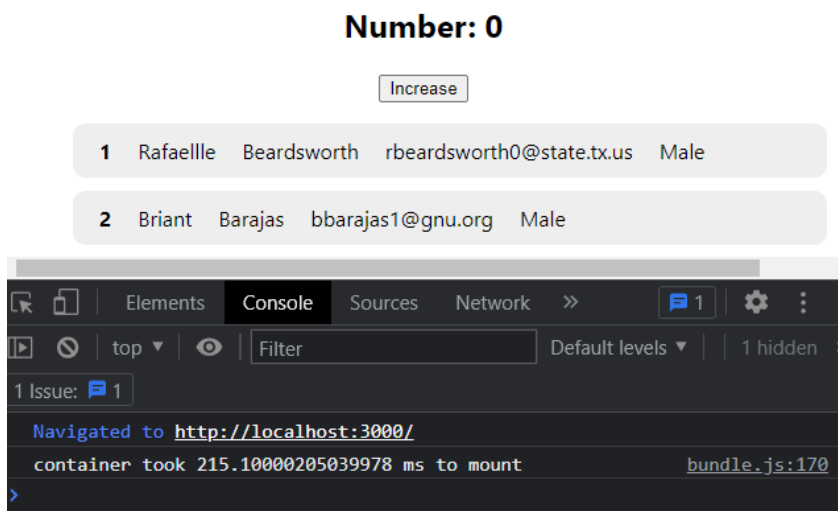
```

The component render time is evident from its id and numbers. In Figure 11, the render time for the specific component is fast and falls within an acceptable range. Typically, render duration should not exceed 16 milliseconds to provide a smooth app experience. (React Native, n.d.) The results appear satisfactory. However, if a slow performance is observed, an investigation into the cause may be necessary. This could signal issues like unnecessary re-rendering of multiple

components. In such cases, an examination of the application's state management method would be necessary.

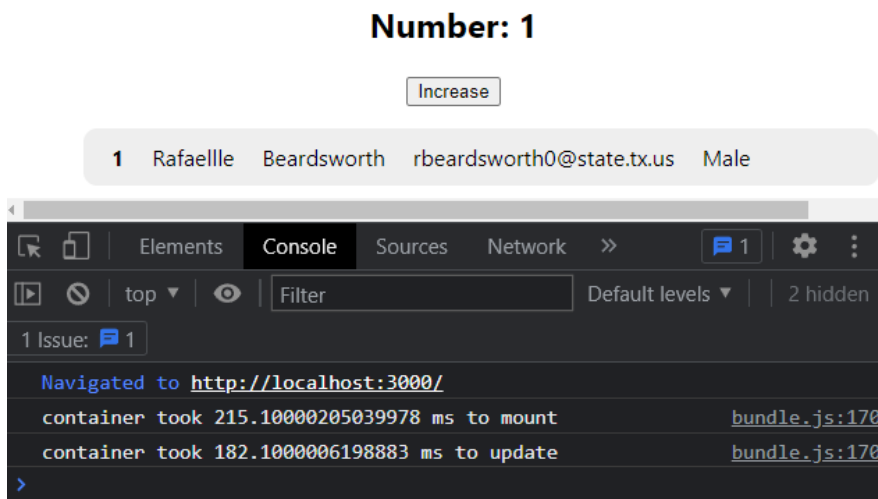
Applying the same steps to the second application, which renders a long list of people's information, yields the following measurements. The profiler from React is imported, and the container application is wrapped. Upon refreshing the page, the obtained result is as in Figure 9.

Figure 9. Demo application render time.



Observing that rendering 3000 distinct persons in a single view takes longer than before highlights a serious performance issue. Now, clicking the button to increase the value of the number will reveal what happens as in Figure 10.

Figure 10. Demo application render time after updating the state.



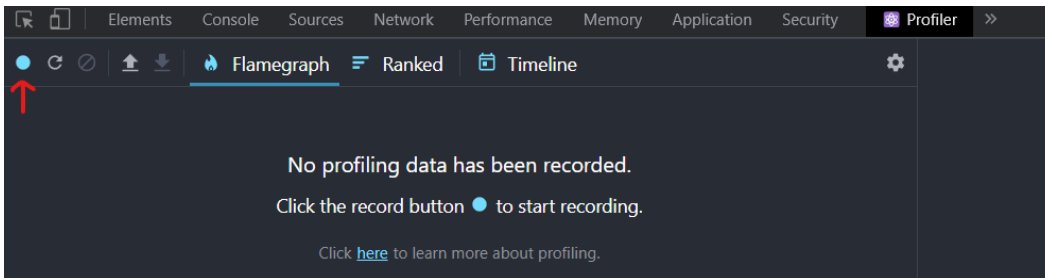
As observed, updating the component takes a long time since the entire container component must be re-rendered each time the number state changes. The container contains the entire list of people, causing a delay due to the extensive list. The delay is even noticeable when clicking the increase button, signaling a significant performance problem in more complex applications.

### 6.3 Component performance

One of the most important characteristics of a user-friendly web application is its efficiency. As a result, evaluating the efficiency of the application is essential. The performance of the application also can be monitored, and any performance related problems can be discovered. Application's performance can be accessed using React Developer Tools.

To begin, the React Developer Tools extension must be added to the web browser which is Chrome in this test case. Once installed, Profile page can be seen as shown in Figure 11.

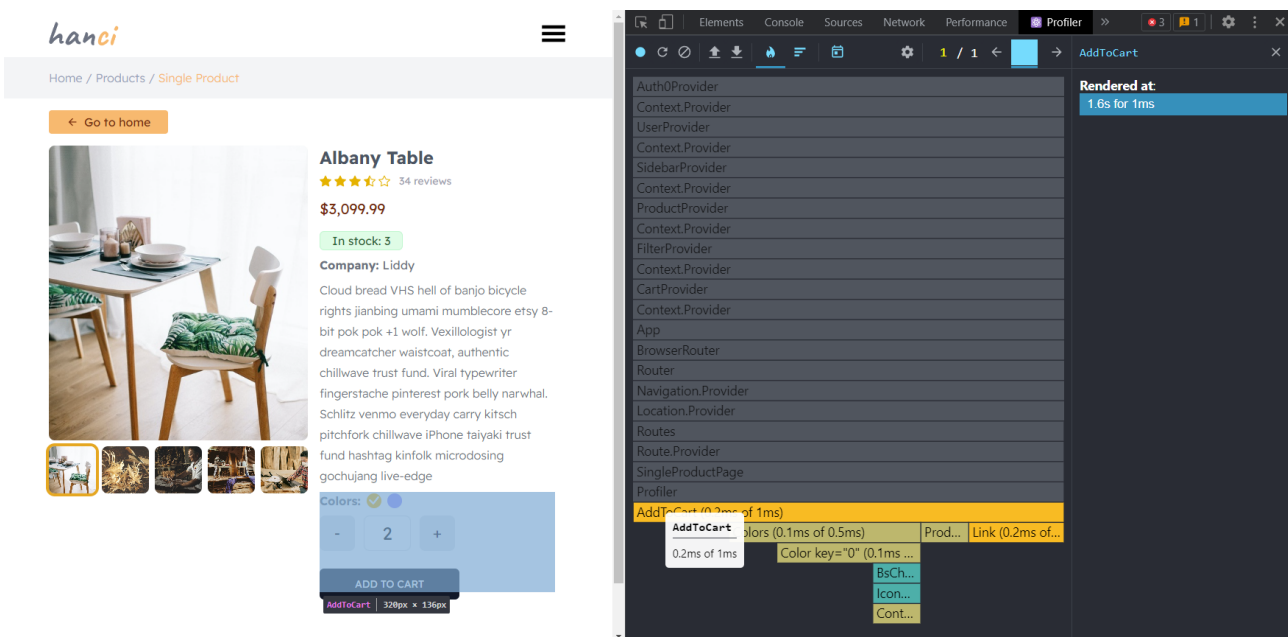
Figure 11. React Developer Tools Profiler.



Navigating to the Profiler tab, users can start interacting with the application by pressing the record button and proceeding. Users can record the entire process and identify any performance issues that may arise.

Assuming being on a single product page in Figure 12, and reading the product's information, deciding to purchase two units. In this case, clicking the plus button to increase the quantity by one becomes the first user interaction. Now, hitting the record button and proceeding with the action will capture the interaction.

Figure 12. Recording actions in React Developer Tools Profiler tab.

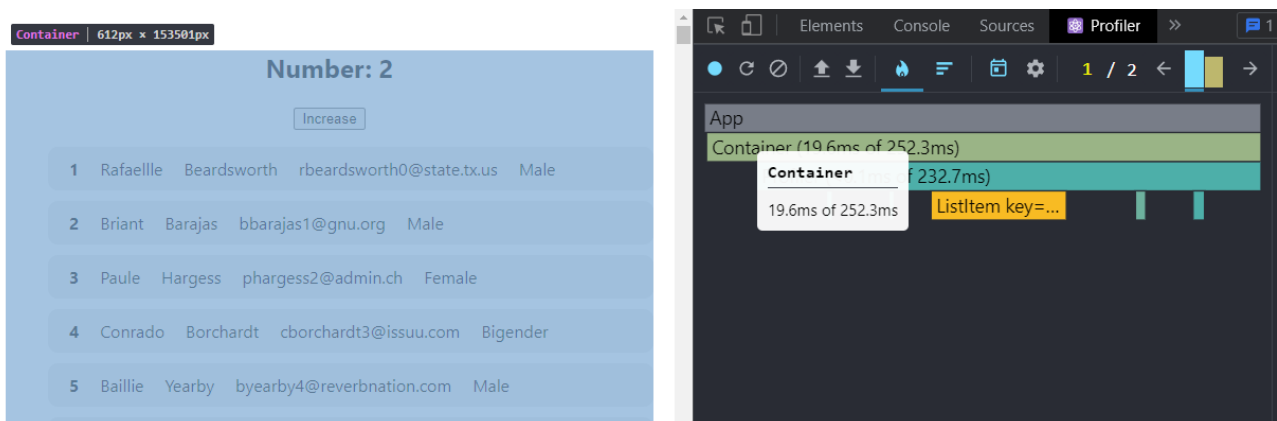


The entire page was not updated when the quantity was increased, which is generally a positive sign because updating components unaffected by state changes negatively impact performance and defeats the purpose of using a front-end framework like React.js for manipulating actual DOM.

Even if the only thing that changes is the amount number, a closer look reveals that React updates the entire container component. The container component comprises not only colors and an add-to-cart button but also other child components. While it may seem inconsequential in this scenario, if the container component includes more than just colors and buttons, it could significantly impact render time. For this reason, as a precautionary measure, preventing this kind of behavior is considered good practice. Now that the potential issue is identified, the focus can shift towards exploring solutions.

Investigating the same scenario in the second demo application, which renders a long list of people's information with a button that increases the number value. The author runs the application and then goes to the profiler tab under the React developer tools. Clicking the record button, the author increases the number two times by clicking the increase button. Result is shown in Figure 13.

Figure 13. Results in React Developer Tools Profiler tab.



The whole container component was re-rendered. It should be remembered that the container component contains a long list. It means that only the number were supposed to be updated. This can be considered as a significant performance issue. Now, possible solutions can be applied to



prevent this performance issue. In this case, using the `useMemo` or `useCallback` react hooks can be considered.

## 6.4 Profiling memory usage

Measuring the memory usage of the application and gathering metrics might provide a hint beforehand about its performance or potential fixes before deployment. Before proceeding with measuring the memory consumption and collecting valuable metrics, a basic understanding of how JavaScript resides in a computer's memory is essential.

**JavaScript Heap:** Heap is a portion of the memory where JavaScript objects live. It is used for dynamic memory allocation during the execution of the JavaScript code. It stores the objects created during the runtime, such as variables, arrays, and functions. A necessary space is allocated on the heap whenever a user creates a new object using the "new" keyword or any other constructor. It is crucial to understand how to optimize memory consumption in any JavaScript application. The author will use Chrome Developer Tools for measuring memory usage. The JavaScript objects on our page and associated DOM nodes are displayed in the Chrome DevTools heap profiler as a breakdown of memory usage. It can be used to discover memory leaks, compare snapshots, analyze memory graphs, and take JS heap snapshots. (Chrome for Developers, 2015)

Before continuing, look at the "Memory" tab under Chrome Developer Tools. Having a basic knowledge of each section under the memory tab is crucial:

- **Heap size:** This section displays both used and unused memory allocated by the JavaScript heap. This value can be monitored to gain an overview of your application's memory usage.
- **Heap snapshot:** This section profiles how memory distributed among JavaScript object and related DOM nodes.

- **Timeline of Allocations:** This section provides a timeline of JavaScript heap allocations. Recording allocations and analyzing memory allocation patterns over time is possible. To begin recording, select the red dot button in the panel's bottom-left corner.

With that basic information, the memory consumption of a JavaScript program can be monitored and analyzed, including React applications, to identify potential memory leaks and optimize overall memory usage.

Now the author will identify a memory leak in one of our demo applications. A memory leak occurs when an application or program fails to release memory that it no longer requires or cannot access. It is a situation in which allocated memory is not correctly released, resulting in a gradual increase in memory usage over time. It may not affect the application's performance immediately, but over time, it decreases the performance and may cause random crashes or unresponsive applications. Therefore, it is crucial to identify and fix memory leaks to ensure optimal performance and avoid unnecessary resource usage in React applications.

In the demo application, there is a button that shows and hides an example component that has an interval in it, and this interval runs whenever the example component mounts. Then interval updates the number value displayed in the example component every second. The example code can be seen in Program code 11.

Program code 11. Component code that causing the memory leak.

```
import { useState, useEffect } from "react";

const Example = () => {
  const [count, setCount] = useState(0);

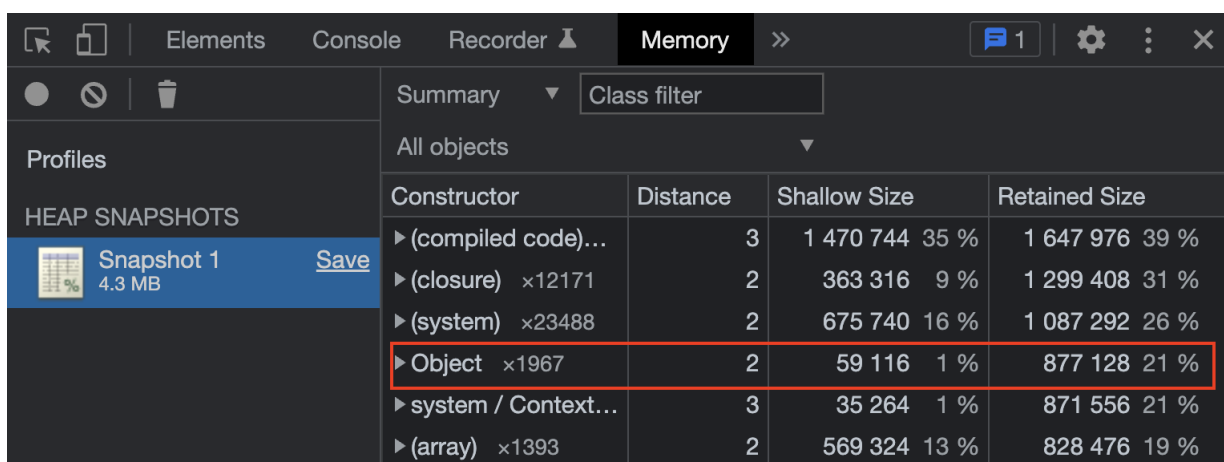
  useEffect(() => {
    const interval = setInterval(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);
  }, []);

  return <div>Count: {count}</div>;
};

export default Example;
```

The author started by clicking the show button, after which the example component and the number value it renders, which increases by one each second. The example component is now unmounted and no longer visible on the screen after the hide button is pressed. Now Chrome Developer Tools can be utilized to take a memory snapshot. The memory snapshot can be seen in Figure 14. (NodeJS, n.d.)

Figure 14. Heap snapshot 1.



The screenshot shows the Chrome DevTools Memory tab with a heap snapshot. The 'Memory' tab is active, and the 'Summary' view is selected. The 'Class filter' is set to 'All objects'. The 'Profiles' panel on the left shows 'HEAP SNAPSHOTS' with 'Snapshot 1' (4.3 MB) selected. The main table displays the following data:

Constructor	Distance	Shallow Size	Retained Size
▶ (compiled code)...	3	1 470 744 35 %	1 647 976 39 %
▶ (closure) ×12171	2	363 316 9 %	1 299 408 31 %
▶ (system) ×23488	2	675 740 16 %	1 087 292 26 %
▶ Object ×1967	2	59 116 1 %	877 128 21 %
▶ system / Context...	3	35 264 1 %	871 556 21 %
▶ (array) ×1393	2	569 324 13 %	828 476 19 %

The author will wait for one minute and then take another heap snapshot. The second memory snapshot shown in Figure 15.

Figure 15. Heap snapshot 2.

Constructor	Distance	Shallow Size	Retained Size
▶ (compiled code)...	3	1 483 212 35 %	1 660 476 39 %
▶ (closure) ×12457	2	371 324 9 %	1 305 772 31 %
▶ (system) ×23488	2	675 740 16 %	1 087 292 25 %
▶ Object ×2254	2	68 300 2 %	894 320 21 %
▶ system / Context...	3	35 264 1 %	871 016 20 %
▶ (array) ×1393	2	569 324 13 %	828 476 19 %
▶ (object shape)...	2	496 792 12 %	513 528 12 %

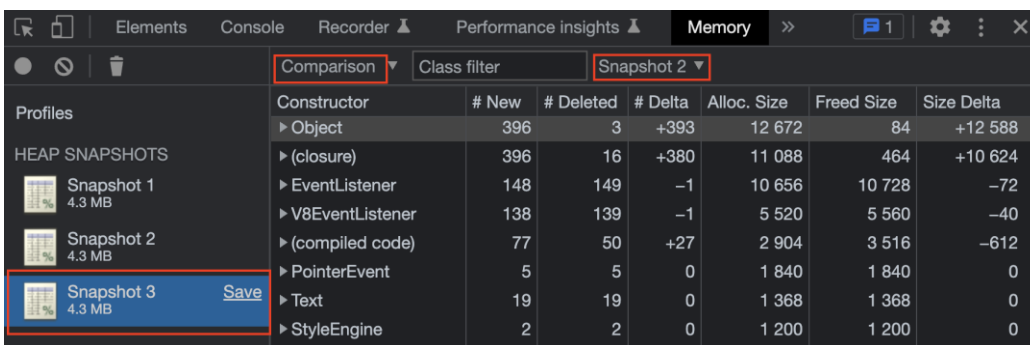
The first thing that stands out is the increased JavaScript object size and number. This means new JavaScript objects were created and took space in memory. That alone tells us that something about the application is wrong. The author will wait for a couple of minutes, and then will take one last heap snapshot to be sure that the heap size is still increasing. The last snapshot is shown in Figure 16.

Figure 16. Heap snapshot 3.

Constructor	Distance	Shallow Size	Retained Size
▶ (compiled code)...	3	1 482 600 34 %	1 659 904 39 %
▶ (closure) ×12837	2	381 948 9 %	1 313 168 31 %
▶ (system) ×23485	2	675 656 16 %	1 088 096 25 %
▶ Object ×2647	2	80 872 2 %	928 672 22 %
▶ system / Context...	3	35 044 1 %	876 232 20 %
▶ (array) ×1387	2	568 836 13 %	836 720 19 %
▶ (object shape)...	2	496 364 12 %	513 100 12 %

Object size and number keep increasing, indicating that the application has a memory leak problem. Now that it is clear there's an issue, it is time to understand what part of the application caused it. The differences between snapshots can be observed by comparing them. The author will select snapshot 3, choose the comparison dropdown item, and then select snapshot 2 to see changes that occurred during the time between snapshot 2 and 3. (Chrome DevTools, n.d.-b)

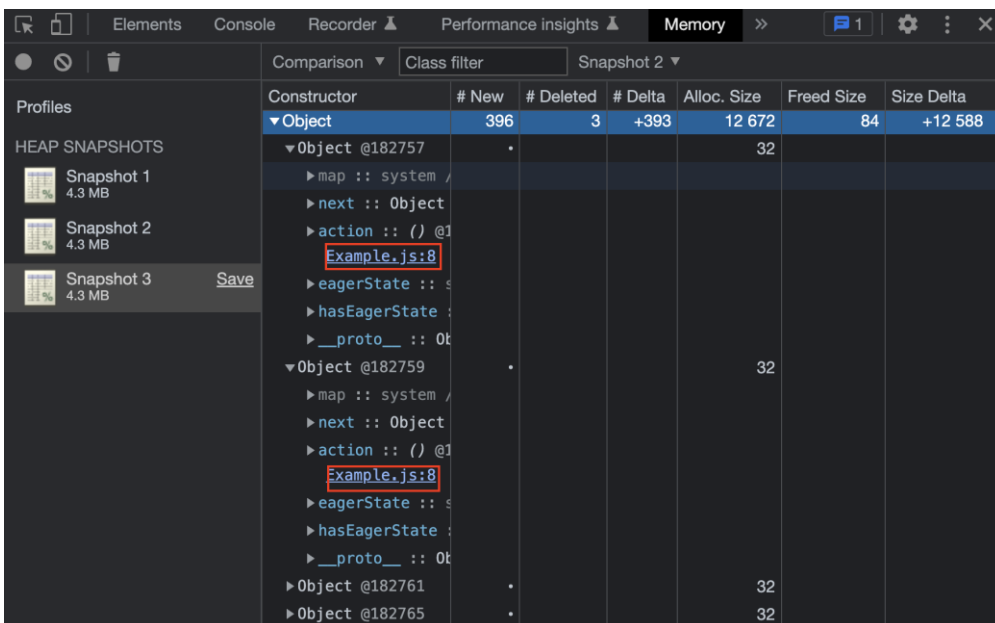
Figure 17. Heap snapshot comparison.



Constructor	# New	# Deleted	# Delta	Alloc. Size	Freed Size	Size Delta
Object	396	3	+393	12 672	84	+12 588
(closure)	396	16	+380	11 088	464	+10 624
EventListener	148	149	-1	10 656	10 728	-72
V8EventListener	138	139	-1	5 520	5 560	-40
(compiled code)	77	50	+27	2 904	3 516	-612
PointerEvent	5	5	0	1 840	1 840	0
Text	19	19	0	1 368	1 368	0
StyleEngine	2	2	0	1 200	1 200	0

Now, the changes that occurred between memory snapshot 2 and 3 can be seen in the memory tab. Next, the author will click the triangle icon next to the Object row to see the created objects as in Figure 18.

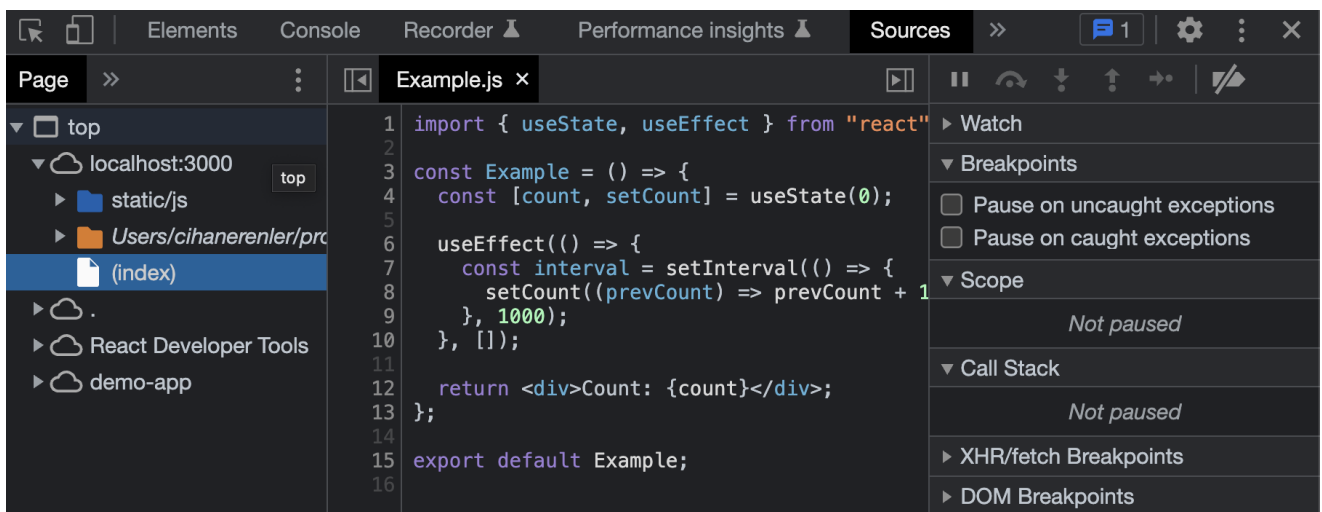
Figure 18. Heap snapshot objects.



Constructor	# New	# Deleted	# Delta	Alloc. Size	Freed Size	Size Delta
Object	396	3	+393	12 672	84	+12 588
Object @182757	•			32		
Object @182759	•			32		
Object @182761	•			32		
Object @182765	•			32		

By analyzing the retained objects, the references preventing the component from being garbage collected can be identified. It is observed that the code causing the memory leak comes from Example.js, the file containing the Example component's code. Fortunately, the code snapshot can be viewed by clicking the file name. Next, the author will click the file name to find out. After clicking the file name, the code can be displayed as in Figure 19.

Figure 19. Heap snapshot problem code.



```

1 import { useState, useEffect } from "react"
2
3 const Example = () => {
4   const [count, setCount] = useState(0);
5
6   useEffect(() => {
7     const interval = setInterval(() => {
8       setCount((prevCount) => prevCount + 1
9     }, 1000);
10  }, []);
11
12  return <div>Count: {count}</div>;
13 };
14
15 export default Example;
16

```

The observed time interval causes memory leaks. Even though this component is unmounted, the timer continues running, leading to a memory leak. The interval is not cleared, and the component's reference is retained, preventing it from being garbage collected. A cleanup function needs to be added to address the memory leak in the useEffect hook, clearing the interval when the component is unmounted. This can be implemented as in Program code 12. Also, the importance of the life cycle hooks can be seen here. The unmount lifecycle hook can be used to clear the interval after the component is unmounted.

Program code 12. Updated version of the component that causing memory leak.

```
import { useState, useEffect } from "react";

const Example = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);

    return () => {
      clearInterval(interval);
    };
  }, []);

  return <div>Count: {count}</div>;
};

export default Example;
```

With this way, the interval will be cleared after the component is unmounted.

## 6.5 Network performance

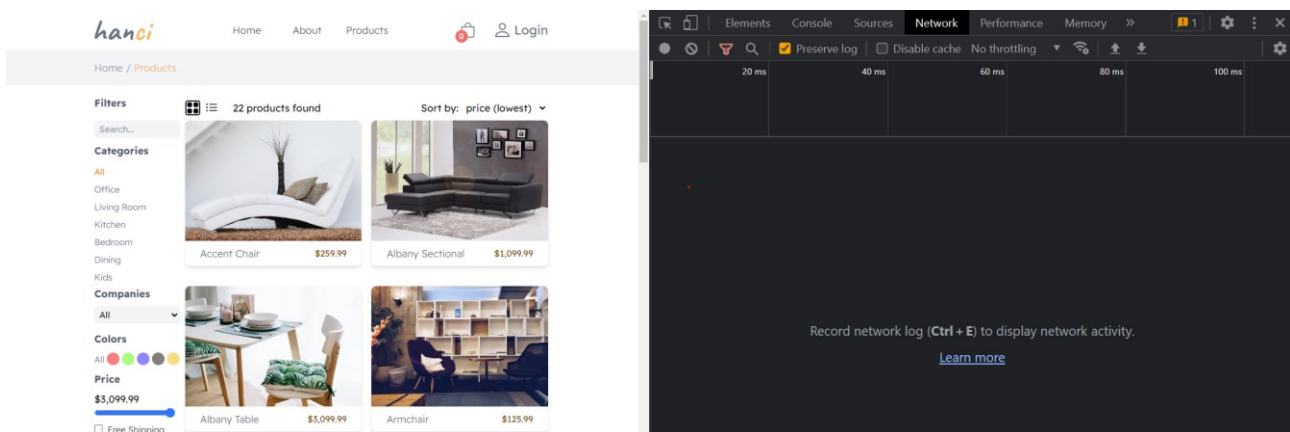
Since it directly impacts the user experience, network performance is considered one of the most important aspects of a React application. It influences the page's loading time, and optimizing network requests and employing different caching techniques enhances user experience and engagement. Improving network speed allows for the use of fewer resources, making the application's system more effective. (Chrome DevTools, n.d.-b)

When profiling React applications and measuring various metrics, several aspects come into play, including network requests made by the application, request/response times, network latency, and data transfer size. This approach helps identify potential bottlenecks and improve the application's performance.

Utilizing Chrome DevTools Network tab to analyze the network performance of a React application is a proficient approach to acquiring comprehension regarding the network requests executed by the application. The tool analyzes metrics such as request timings, data transfer sizes, and HTTP statuses.(Chrome DevTools, n.d.-b)

To measure the metrics, the author will use a demo e-commerce application. The author will run the application and open the Chrome DevTools by right-clicking on the page and selecting inspect. Then, the author will switch to the Network tab on the DevTools panel. The network tab is shown in Figure 20.

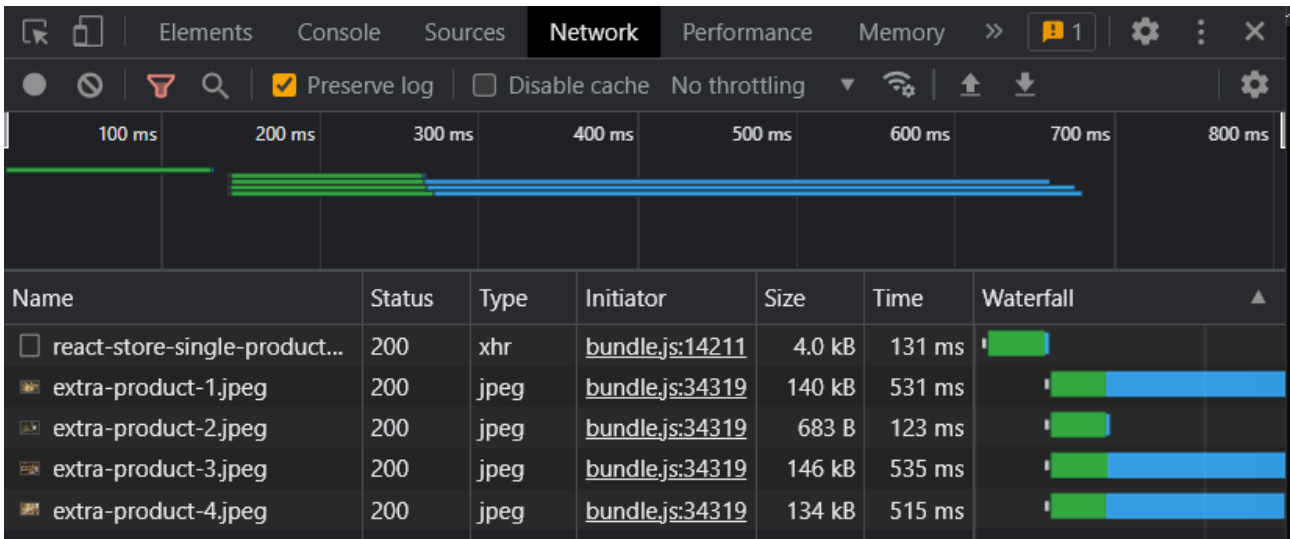
Figure 20. Network profiling of the demo application.



Now the author will click the circle record button on the dev tool's top left corner to start the network recording. Then he will click one of the products displayed on the screen to navigate to a single product page. This will send an HTTP request to API to retrieve specific product data. After that, network request can be analyzed. Results seems as shown in Figure 21.

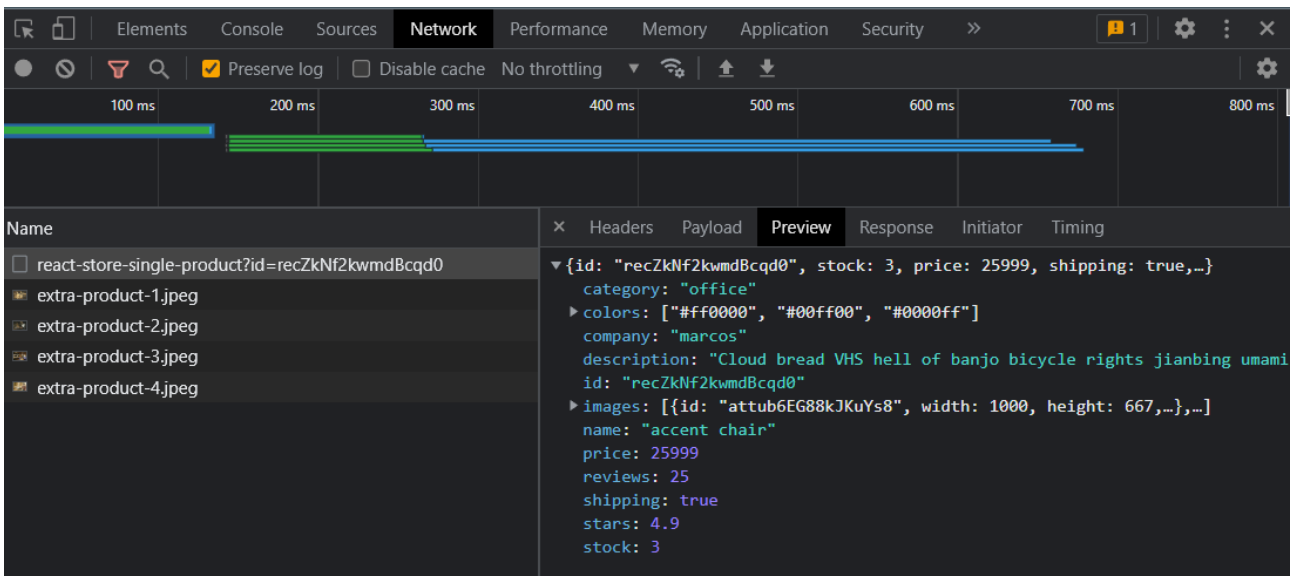


Figure 21. Network tab recording requests.



Chrome DevTools recorded network requests when the author interacted with the application. Every request is presented as a separate row, providing information such as the waterfall timeline, request method, status, size, and timings. From here, an individual row can be clicked and more detailed information about that request can be seen. The author will click the first item in the table to see more information. Different tabs such as headers, payload, preview, response, initiator, and timing can be seen in Figure 22

Figure 22. Network request recording details.



The headers tab displays the detailed information for request and response headers. The payload tab displays the parameters passed to the request URL, such as id. The response tab displays the raw response data. The preview displays the formatted response data. Lastly, the timing tab displays various timings related to the request, such as DNS lookup time, connection time, and data transfer time.

Examine requests with lengthy durations, large data sizes, or high latencies. These can indicate possible performance issues or optimization opportunities. Consider variables such as slow server response times, inefficient data transfer volumes, and mergeable multiple sequential requests.

## 7 Summary

This thesis delves into the world of profiling React apps, focusing on the issues React.js addresses efficiently and offering a thorough insight into its underlying workings. With the rising complexity of modern online applications, optimizing speed and improving user experience is critical. React profiling tools help reach this aim.

The thesis started by looking into the fundamental difficulties that React.js handles, such as maintaining the state of complicated UIs, optimizing re-renders, and updating the DOM effectively. React.js's core workings can be understood by delving into its reconciliation process, virtual DOM, and component lifecycle functions, all of which contribute to React's outstanding speed.

Various profiling tools are available to developers to help them improve React apps. This thesis delves deeply into these technologies, offering insights into their features, capabilities, and best practices. Developers may use React profiling tools to find performance bottlenecks, detect inefficient renderings, and optimize their components for increased efficiency.

This study includes a complete case study on profiling component render time, component performance, memory utilization, and network performance. Real-world examples are examined to demonstrate the practical uses of React profiling. Developers may make educated judgments about optimizations by methodically evaluating the performance characteristics of components, resulting in speedier and more responsive apps.

Memory use analysis also identifies any memory leaks or excessive memory consumption, guaranteeing effective resource utilization. Furthermore, network performance assessment aids in the identification of network bottlenecks, allowing developers to improve data fetching and transmission efficiency.

Finally, this thesis offers a unique and complete investigation of profiling React apps, providing a profound grasp of the challenges React.js addresses, its internal workings, and the use of React

profiling tools. The case studies explain how to use profiling approaches to enhance component render time, speed, memory utilization, and network performance. Developers may unleash the full potential of React by using these profiling approaches, providing highly performant and responsive apps to users.

## 8 References

Chrome DevTools. (n.d.-a). *Chrome for Developers*. <https://developer.chrome.com/docs/devtools/>

Chrome for Developers. (2015, June 8). *Record heap snapshots*.

<https://developer.chrome.com/docs/devtools/memory-problems/heap-snapshots/>

Chrome DevTools. (n.d.-b). *Inspect Network Activity*.

<https://developer.chrome.com/docs/devtools/network>

Marttila, R. (2016). *Handling unidirectional data flow in a React.js application*.

<https://trepo.tuni.fi/handle/123456789/24121>

MDN. (2023, May 21-a). *Introduction to the DOM - Web APIs*. [https://developer.mozilla.org/en-](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

[US/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

MDN. (2023, August 3-b). *Web performance*.

<https://developer.mozilla.org/en-US/docs/Web/Performance>

NodeJS. (n.d.). *Using Heap Snapshot*.

<https://nodejs.org/en/learn/diagnostics/memory/using-heap-snapshot>

Pitt, C. (2016). *React Components*. Packt Publishing Ltd.

React. (n.d.-a). *Profiler*. <https://react.dev/reference/react/Profiler>

React. (n.d.-b). *Components and Props*.

<https://legacy.reactjs.org/docs/components-and-props.html>

React. (n.d.-c). *Introducing JSX*. <https://legacy.reactjs.org/docs/introducing-jsx.html>

React Blog. (n.d.). *Introducing the React Profiler*

<https://legacy.reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html>

React. (n.d.-d). *Managing State*. <https://react.dev/learn/managing-state>

React. (n.d.-e). *React Developer Tools*. <https://react.dev/learn/react-developer-tools>

Reactotron. (n.d.). *Inspect your React and React Native apps*. <https://infinite.red/reactotron>

React. (n.d.-f). *State and Lifecycle* <https://legacy.reactjs.org/docs/state-and-lifecycle.html>

React. (n.d.-g). *Virtual DOM and Internals*. <https://legacy.reactjs.org/docs/faq-internals.html>

React Native. (n.d.). *Performance Overview*. <https://reactnative.dev/docs/performance>

## 9 Material management plan

A thorough material management plan is necessary in the context of this thesis, which is mainly research and use case-based, to ensure the systematic arrangement, preservation, and appropriate treatment of the research materials. The exact tactics and procedures used to handle the research materials throughout the course of this investigation are described in this section.

Academic publications and original documentation pages serve as the bulk of the research material for this thesis. This study did not include any data collected from or interviews with human participants. There are no concerns about the preservation of personal or secret information since the research resources are mostly scholarly publications and publicly available paperwork.

Copyright and intellectual property rights are carefully protected when using third-party information, such as quotes from scholarly journals and other documents. According to accepted academic guidelines, proper citations and references are used.

The original authors and publishers retain ownership of the research materials, which usually come from academic journals and publicly accessible documentation. Only academic study and reference uses are intended for the data gathered for this thesis. Beyond the scope of this thesis, no more sharing, distribution, or usage is intended.

This material management plan makes sure that research materials are handled and preserved ethically and in accordance with this research thesis's specifications.