REACT APPLICATION OPTIMIZATION

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Leng Zhang

2021

SIGNATURE PAGE

THESIS:	REACT APPLICATION OPTIMIZATION
AUTHOR:	Leng Zhang
DATE SUBMITTED:	Fall 2021
	Department of Computer Science
Dr. Yu Sun Thesis Committee Chair Professor of Computer Science	______________________________
Dr. Gilbert S. Young Professor of Computer Science	______________________________
Dr. David L. Johannsen Professor of Computer Science	______________________________

ACKNOWLEDGEMENTS

I am so honored to have so many wonderful professors at Cal Poly Pomona. Especially, Dr. Yu Sun, who is my thesis advisor, encouraged me to move forward with this thesis and gave me a lot of advice for my future career. And I would like to thank my wife for supporting me in pursuing the master’s degree in computer science at Cal Poly Pomona. She worked and took care of our little boy at the same time while I took time to complete my thesis.

ABSTRACT

Optimization is one of the most essential elements for building any application, especially web applications. Modern web applications are more interactive than ever. With poor performance, users may experience disruptions and dissuade the user from continuing using the application. Improved performance can greatly improve the user experience such as lower latency and faster rendering time.

Optimizing web applications takes effort because web applications are not just running on the client-side. A web application is built from a variety of technology stacks: a database to store and manage data, back-end components to serve requests, and front-end components which include HTML, JavaScript, and CSS for visualization. Today, several front-end frameworks allow optimization to be done at ease. In particular, React is one such JavaScript framework. This paper will focus on the optimization strategies for React web applications.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 - Background

In the last decade, JavaScript has explosive growth, and one of the main reasons is that JavaScript is based on Open Web technology which is the biggest and most important intersection and consensus of the whole industry in platform technology. As the native language of Web development, JavaScript's capabilities are constantly evolving together with Web technologies. Under the influence of JavaScript, Web and Web development are also very different than the past.

At the early age of Web development, the development was simple and fast. Usually, 3 to 5 people handle all developments regardless of front-end or back-end. The page is generated on the server-side by JSP and PHP, and the browser is responsible for rendering the page. The browser displays what the server generates, and the display control is in the Web Server layer. The advantages of this model are simple and fast when the business logic is simple, but the business logic will always become complex in the end.

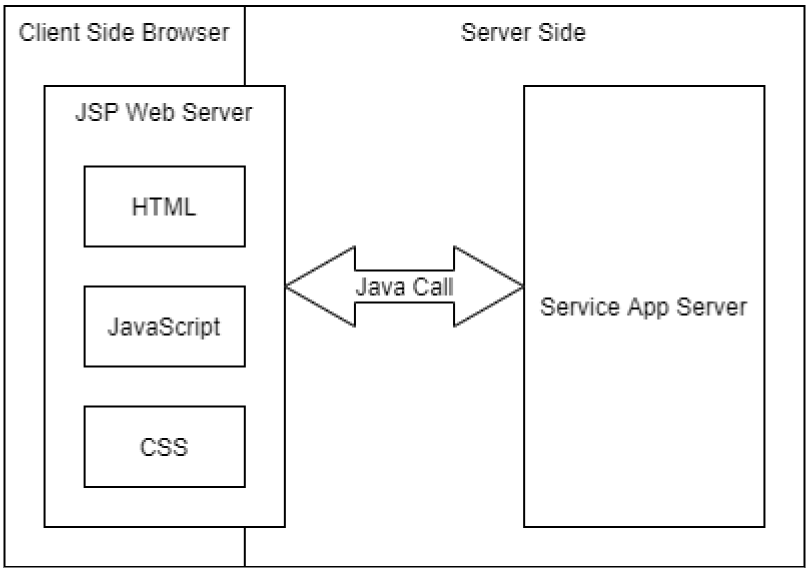


Figure 1: Early Age of Web Development

With more and more servers, relationships between servers become more complicated, and building a local environment on the front end is no longer a simple matter. Considering teamwork, the team will build a centralized development server to solve the problem. This solution may be good for compiled back-end development, but this solution is not friendly for front-end development. For adjusting the style of a button, the developer needs several steps such as developing locally, uploading code, and verification. However, the development server is not always stable and often depends on the back-end team to get works done. The development server solution is difficult to develop the front end locally.

Web technologies have been constantly evolving, and various new web technologies emerge endlessly such as the back-end MVC model, SPA, the front-end MVC model, and Node.js. However, the essence of web technologies never changes, which is to continuously improve productivity, create better products and services, and solve more and more difficult problems. Especially in the field of building user interfaces, various JavaScript frameworks provide various technical solutions such as AngularJS, React, Vue.js, and so on. AngularJS is a fully client-side framework that connects to static HTML with a set of attributes for data binding. React is a component-based JavaScript framework for building user interfaces. Vue.js is an open-source progressive JavaScript framework that provides two-way data binding and server-side rendering for building user interfaces.

Chapter 2 – React

2.1 What is React?

React is the most popular JavaScript framework. According to The State of JavaScript, 2020 survey, React.js is the framework to adopt since 2016. There are 17561 respondents of the survey having positive experiences with React.js. React.js has a usage rate of 80 percent which is 24 percent more than AngularJS and 31 percent more than Vue.js. React.js is the most popular JavaScript framework for building the user interface of a web application.

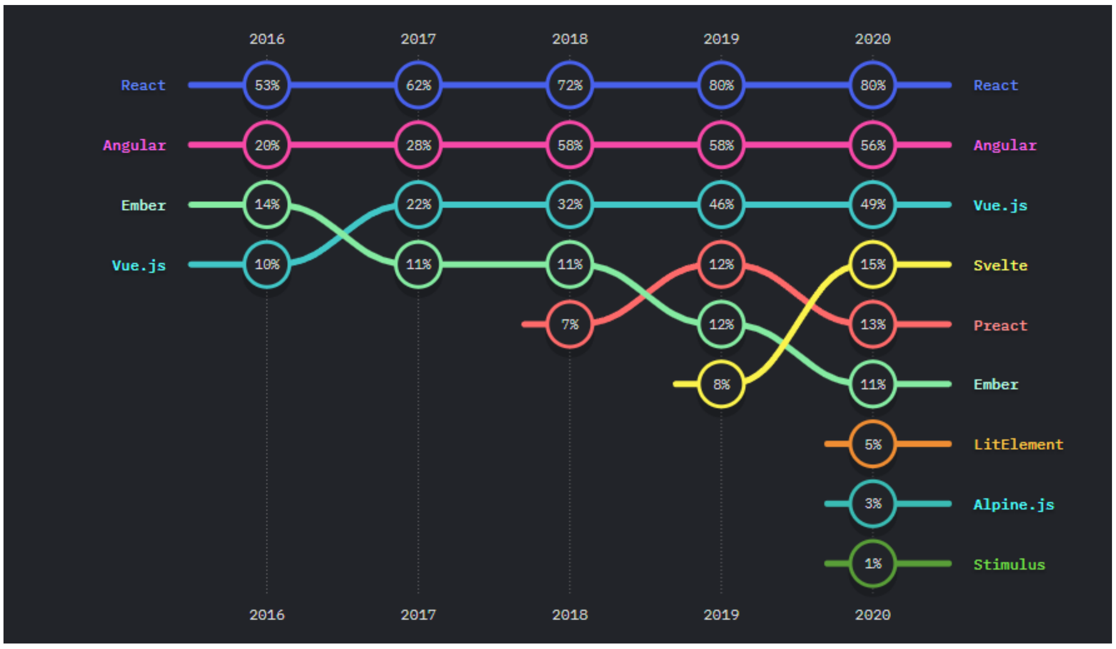


Figure 2: The state of JavaScript Survey 2020 - Front-end Frameworks Usage Ranking

React is used as the design layer of a web application. React can be applied as a foundation while developing mobile or single-page applications, as React is optimal for obtaining swiftly modifying data that must be recorded.[1] There are some core technical features of React such as Components, State and Props, Lifecycle Hooks, Hooks, and virtual DOM.

2.2 Features

2.2.1 React Components

React components are small reusable pieces of code that return a React element to be rendered to the page. The simplest version of React component is a plain JavaScript function as the Function Component of Appendix A, this kind of React component is called the function component. Function component accepts props and returns a React element. And components can also be written by using ES6 class syntax as Class Component of Appendix A. Both function components and class components are equivalent and will render the same output, but the function component is different from the class component. A function component has no state and no lifecycle hooks, function component takes props from the parent component and returns a React element. The class component is a bit more complicated than the function component; the class component has state and lifecycle hooks. React creates instances of class components and initializes them to call lifecycle methods

2.2.2 State and Props

State and props, short for “properties”, are both plain JavaScript objects. Both state and props hold information based on which UI changes, but they have a main difference. The state is managed within the component. For example, a component has a Boolean variable in the state, and the component renders a loader UI when the variable is true. Whereas props are passed into the component from the parent component, props are like function parameters. Props make the component to be reusable, the component can be versatile by getting props from the parent. For example, the component can render different types of loaders based on the variable in props. Therefore, the component can be reused in different situations.

2.2.3 Lifecycle Hooks

Lifecycle hooks are methods provided by React in the class component. By overriding lifecycle hooks, the code can be run at times in the process. For example, overriding shouldComponentUpdate can tell React if a component’s output is not affected by the current change in state or props. The page can fetch data from the server after the page is rendered for the first time by overriding componentDidMount.

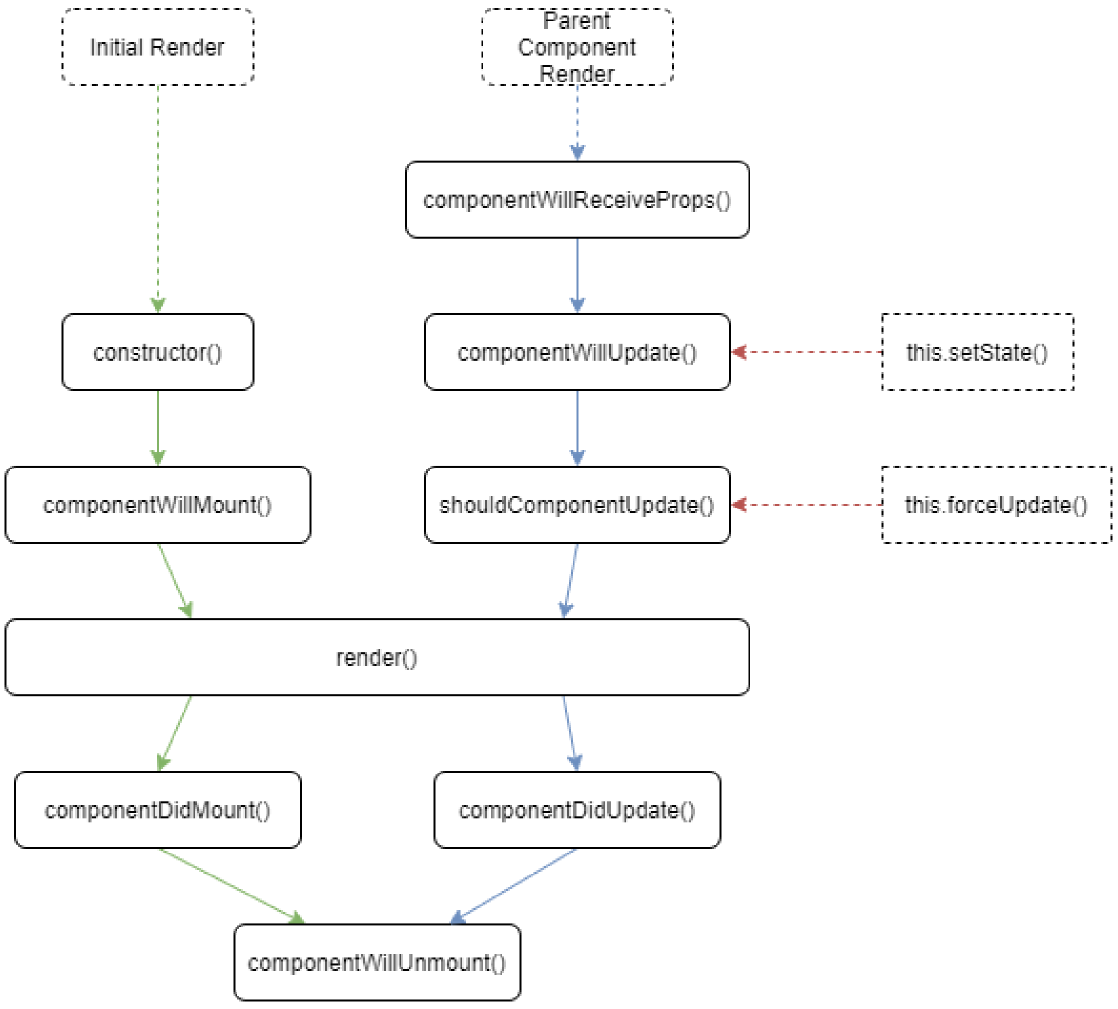


Figure 3: Class Component Lifecycle Hooks

2.2.4 Hook

Hook is a new feature since React 16.8, and Hook lets function components have state and other React features such as lifecycle. Developers can build custom hooks to share reusable stateful logic between components. There are some basic hooks provided by React. The state hook, useState as highlighted lines in Function Component useState of Appendix A, takes an initial state and returns a stateful value and a function to update it, and with useState, function components can become stateful components. The effect hook, useEffect as highlighted lines in Function Component useEffect of Appendix A, adds the ability to perform side effects for a function component. Hook serves the same purpose as lifecycle hooks in the class component.

2.2.5 Virtual DOM

In Web development, the user interface needs to be synced with new data in real-time, so the DOM needs to be manipulated. And manipulating the DOM complexly and frequently is usually the cause of the performance bottleneck. Therefore, React introduced a programming concept, virtual DOM. The ideal representation of a UI is kept in memory and synced with the real DOM tree. React provides a library, ReactDOM, to fulfill this function. Whenever the data changes, React stores the virtual DOM tree in memory and re-renders the entire tree, and then, React calculates the difference between the current virtual DOM tree and the previous virtual DOM tree. This calculation is called reconciliation. In the end, React updates appropriate real DOM nodes which have changed in the current virtual DOM tree.

2.3 React Workflow

2.3.1 Workflow

React is a declarative user interface library. React converts states into a page structure which is the virtual DOM tree. And then, React converts the virtual DOM tree into the real DOM tree. When states have changed, React enters the reconciliation stage first. During the reconciliation stage, React calculates the differences between the new virtual DOM tree and the current virtual DOM tree. After React finishes the reconciliation, React enters the commit stage. During the commit stage, the real DOM tree patches the differences from the reconciliation stage. Finally, the new page with new states is displayed on the screen.

2.3.2 Reconciliation Stage

During the reconciliation stage, React does two calculations. First, React calculates a new virtual DOM structure corresponding to new states. Second, it finds the optimal update plan for modifying the current virtual DOM structure to the new virtual DOM structure. React traverses the virtual DOM by depth-first search. For each virtual DOM node, React does two calculations before React calculates the next virtual DOM node. During the calculation for each virtual DOM node, React calls the render method of the class component or the function component itself first. Some Diff algorithms are implemented inside React, and these Diff algorithms record the virtual DOM node update methods such as update, mount, unmount. In the second step of the reconciliation stage, React uses these Diff algorithms to prepare for the commit stage.

2.3.3 Commit Stage

During the commit stage, React also does two jobs. First, React applies the updated plan which is recorded by the reconciliation stage to the real DOM, and then React calls the hook methods which are exposed to developers such as componentDidUpdate, etc. The executing timing for these two jobs in the commit stage is different from the reconciliation stage. React needs to finish applying the updated plan before calling hook methods. Therefore, the parent component can be found in the componentDidUpdate of the child component, even though the componentDidUpdate of the parent component has not been executed yet.

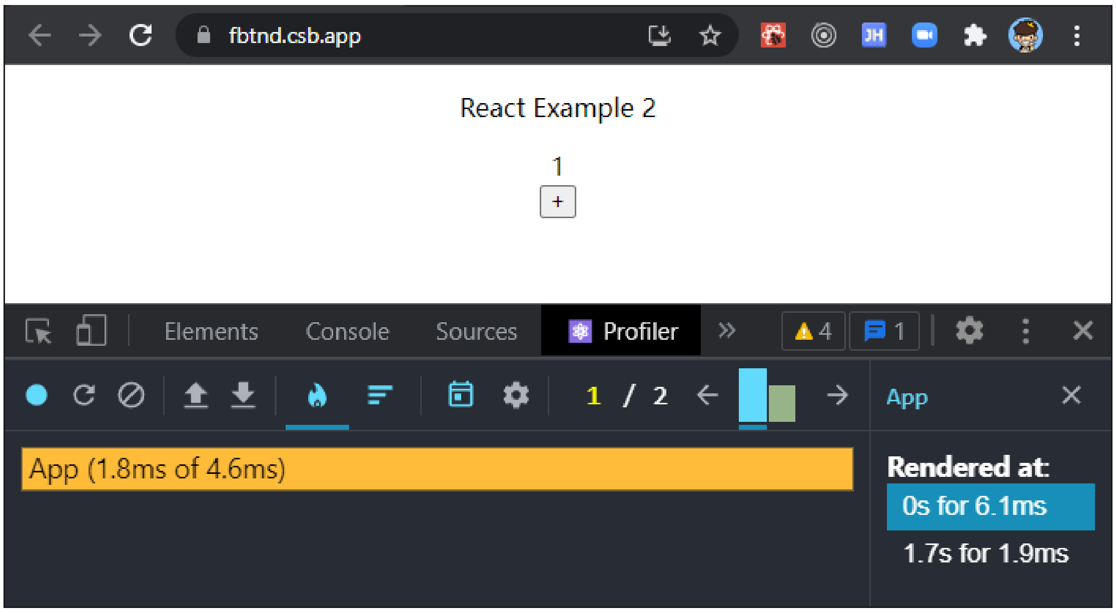
Chapter 3 – Why Optimization?

With virtual DOM, React applications already have high performance, but optimization of React applications is still necessary. There is a case. A component is basically like an excel grid, and the page has a list of this kind of component. If a user is playing with some large data and inserting a row, the page became sluggish and frozen for a few seconds. After the use reloaded the page and made the same actions, the page still doesn’t work. The reason for causing this case is that React generated the virtual DOM by rendering all components and then did reconciliation and updated the real DOM.

The reconciliation is wasting resources in some cases. Ideally, when a child component needs to be updated, React should only reconstruct and compare all components on the path from the root to the child component in the virtual DOM. However, React will reconstruct all components in the virtual DOM tree by default and then compare the generated virtual DOM tree with the previous virtual DOM tree. The application needs extra time to reconstruct components that do not have changes.

In the React Example 1 of Appendix B, the page shows this.state.value on the screen and has a button to trigger the handleClick function which calls setState to increase this.state.value by 1. After the user clicks the button, this.state.value will be increased by 1. The render function will return a new React element with the updated this.state.value as new content which will be re-rendered to the screen.

However, if this.state.value is updated by setState with the same value as the previous value, the render function will also be triggered. In the React Example 2 of Appendix B, after the user clicks the button, the handleClick function will be triggered and call setState to update this.state.value to be 1 which is the default value of this.state.value. Like Figure 4, the profiler result shows that the component App was re-render at 1.7s for 1.9ms after the button is clicked. The state has no changes, but the render function of the component is triggered. For a single component, if the setState function is only called when the state needs to be changed, re-rendering the component can be avoided. Howbeit re-rendering a parent component will also trigger re-rendering child components which have no change.

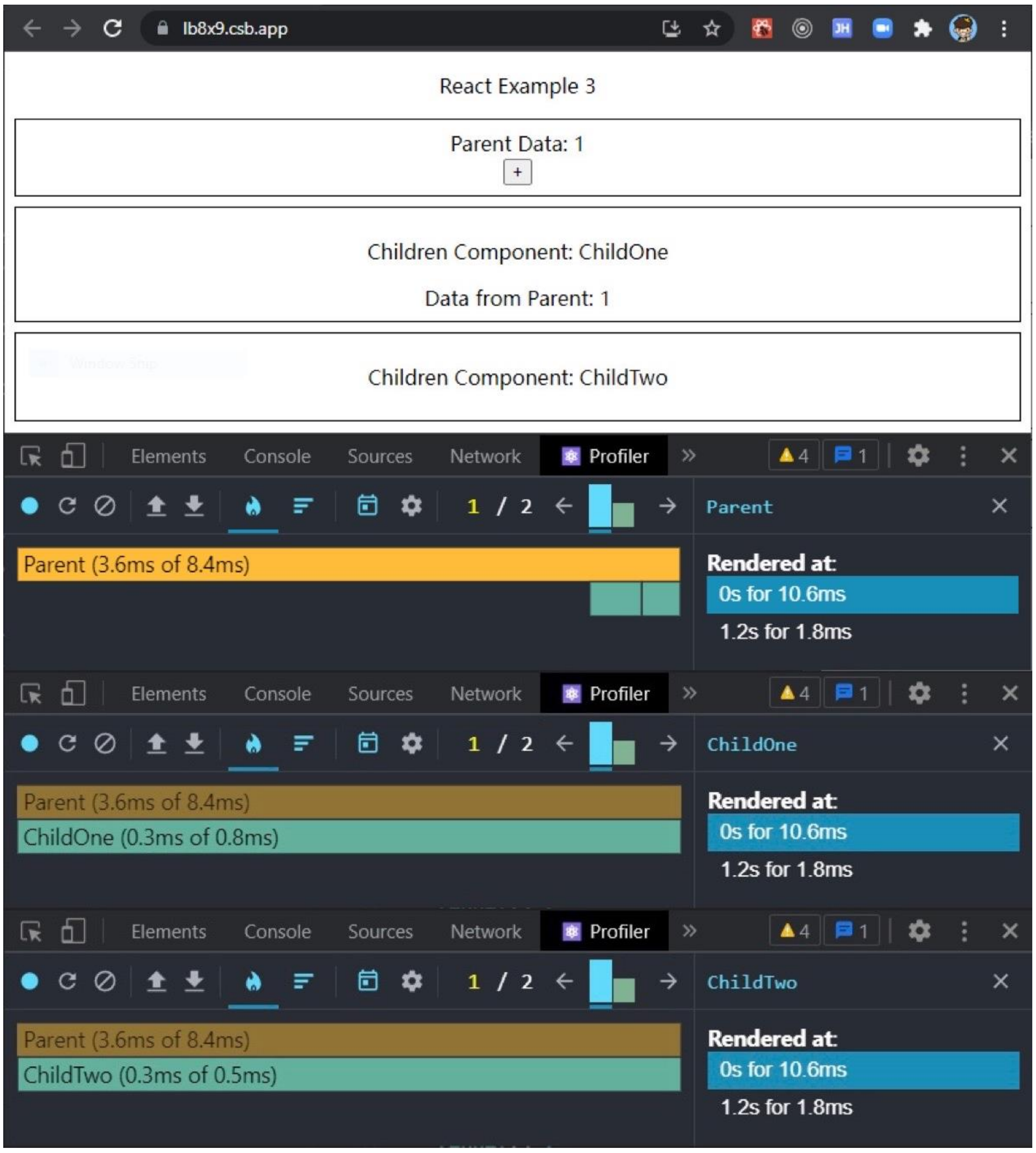


App (1.8ms of 4.6ms)

Rendered at:
0s for 6.1ms
1.7s for 1.9ms

Figure 4: Profiler Result of React Example 2 in Appendix B

In the React Example 3 of Appendix B, the parent component, Parent, has two child components, ChildOne and ChildTwo. ChildOne gets the data from the state of Parent as property and displays the data on the screen. ChildTwo does not get any data from Parent, this component only displays a message on the screen. After the user clicks the button, the handleClick function is triggered, the state of Parent has no changes. However, both ChildOne and ChildTwo are re-rendered. Like Figure 5, the profiler result shows that all components are re-rendered as 1.2s for 1.8ms after the button is clicked. Therefore, whatever the child component gets data from the parent component or not, the child component will be re-rendered if the parent component is rendered.


lb8x9.csb.app

React Example 3

Parent Data: 1

Children Component: ChildOne

Data from Parent: 1

Children Component: ChildTwo

Parent (3.6ms of 8.4ms)

Rendered at:
0s for 10.6ms
1.2s for 1.8ms

Parent (3.6ms of 8.4ms)
ChildOne (0.3ms of 0.8ms)

Rendered at:
0s for 10.6ms
1.2s for 1.8ms

Parent (3.6ms of 8.4ms)
ChildTwo (0.3ms of 0.5ms)

Rendered at:
0s for 10.6ms
1.2s for 1.8ms

Figure 5: Profiler Result of React Example 3 in Appendix B

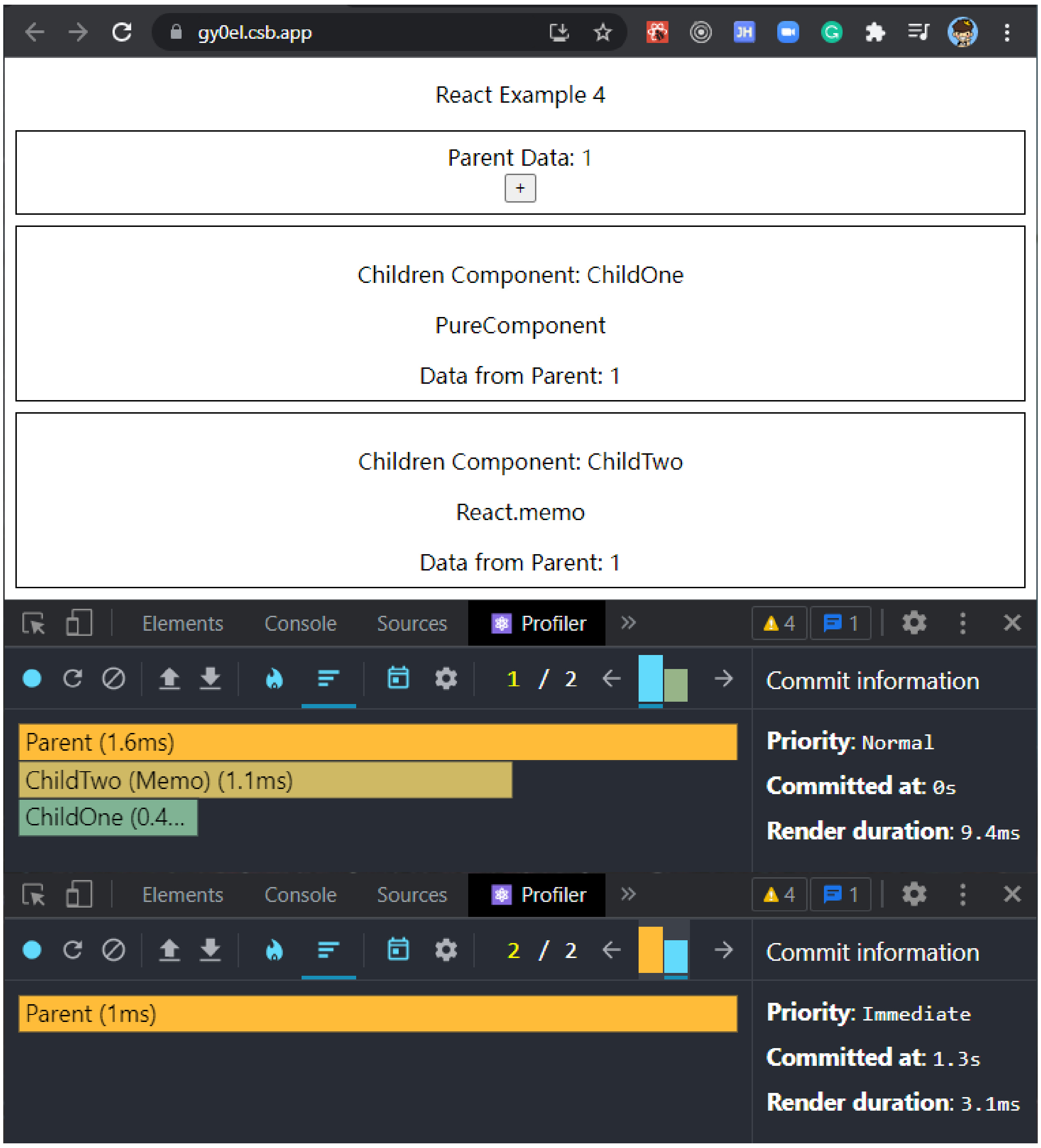
This is an important problem for an application. When the business logic is more complex, and the application becomes huge. The number of components becomes more and more, and the depth of the DOM tree is getting deeper and deeper. Unnecessary rendering brings more performance issues. Therefore, the central concept of React application optimization is to reduce unnecessary component re-rendering after the state of the component changes.

Chapter 4 – Optimization Strategies

4.1 PureComponent, React.memo

As the profiler result shown in Figure 5, only the parent component undergoes a state update, but all props passed from the parent component to the child component are not modified. This state update will cause the child component to re-render. From the perspective of React's declarative design philosophy, if props and the state of the child component are not changed, then the DOM structure and side effects generated should not be changed. When the child component conforms to the declarative design concept, the process of rendering should be skipped at this time. PureComponent and React.memo respond to this scenario. PureComponent does a shallow comparison of props and state for a class component, and React.memo does a shallow comparison of props for a function component.

Figure 6 is the profiler result for implementation of React Example 4 in Appendix B. This example constructs a parent component, Parent, which is inhered regular React component. The parent component has two child components, ChildOne and MemorizedChildTwo. ChildOne is inherited from PureComponent. MemorizedChildTwo is the memorized component of ChildTwo by React.memo. ChildTwo is a function component with the same logic as ChildOne. After the button is clicked, only the Parent will be re-rendered. As Figure 6 shows that the first time rendering for all components, Parent, ChildOne, and ChildTwo, were at 0s for 9.4ms, and only Parent was re-rendered as 1.3s for 3.1ms after the button is clicked. Before each re-rendering, both pure component and memorized component by React.memo do a shallow comparison for props and state to decide re-rendering or not.


gy0el.csb.app

React Example 4

Parent Data: 1

Children Component: ChildOne

PureComponent

Data from Parent: 1

Children Component: ChildTwo

React.memo

Data from Parent: 1

Parent (1.6ms)
ChildTwo (Memo) (1.1ms)
ChildOne (0.4...

Commit information

Priority: `Normal`
Committed at: `0s`
Render duration: `9.4ms`

Parent (1ms)

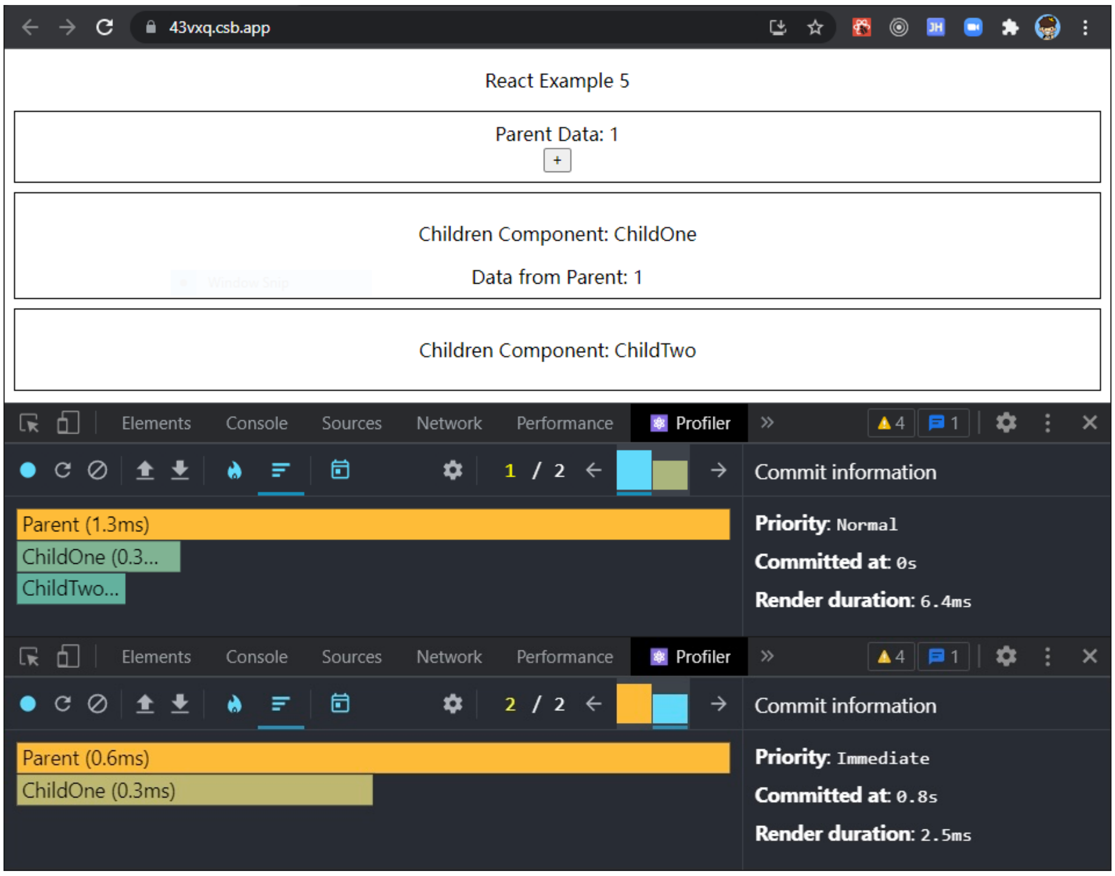
Commit information

Priority: `Immediate`
Committed at: `1.3s`
Render duration: `3.1ms`

Figure 6: Profiler Result of React Example 4 in Appendix B

4.2 shouldComponentUpdate

In actual development, developers are usually passing large objects from the parent component to the child component as props. When a property of the large object is updated but is not used in the child component, this update will trigger the render process of the child component. As the React Example 5 in Appendix B, both child components, ChildOne and ChildTwo, are inherited from PureComponent. this.state.value is an object with a number property called “number”, and Parent passes this.state.value to ChildOne. ChildOne displays props.value.number on the page. Figure 7 shows that only ChildOne and Parent are re-rendered after the button is clicked.


43vxq.csb.app

React Example 5

Parent Data: 1

Children Component: ChildOne

Data from Parent: 1

Children Component: ChildTwo

Parent (1.3ms)
ChildOne (0.3...
ChildTwo...

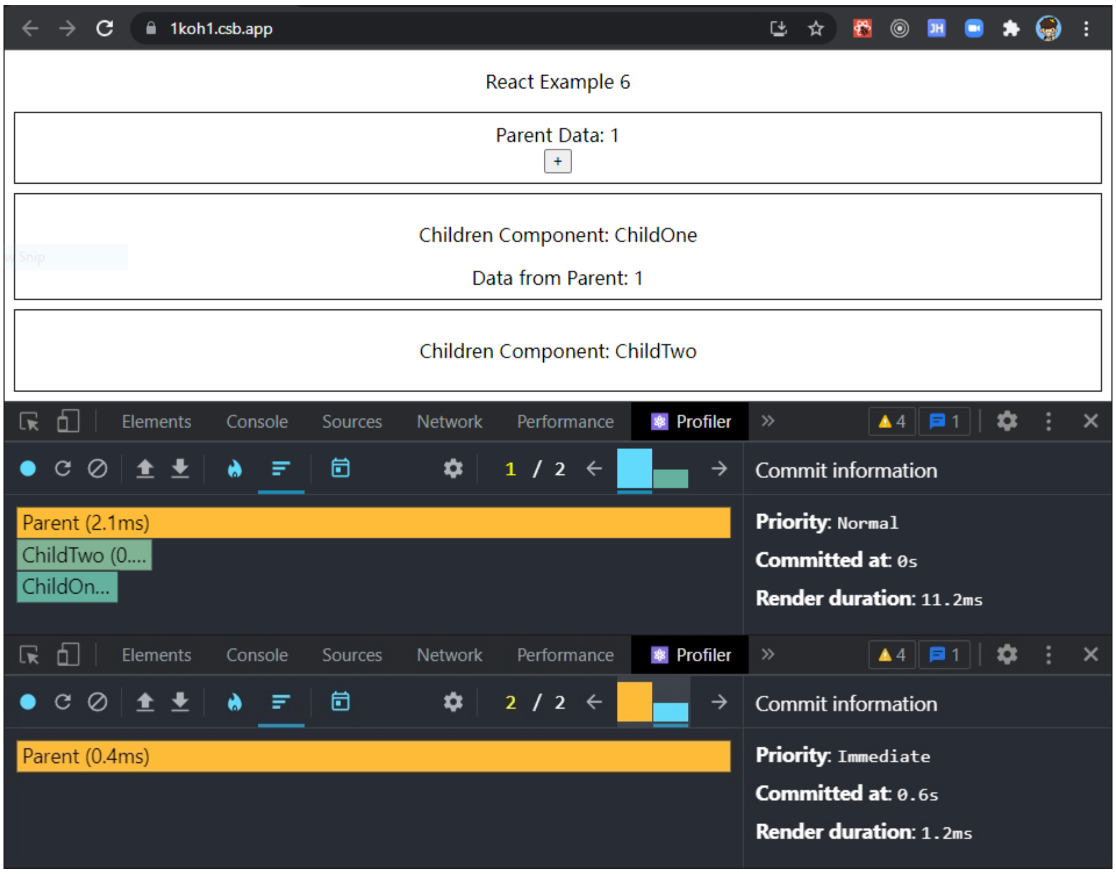
Priority: Normal
Committed at: 0s
Render duration: 6.4ms

Parent (0.6ms)
ChildOne (0.3ms)

Priority: Immediate
Committed at: 0.8s
Render duration: 2.5ms

Figure 7: Profiler Result of React Example 5 in Appendix B

In this scenario, developers can overwrite shouldComponentUpdate with a custom deep comparison algorithm for props and state. Therefore, shouldComponentUpdate can compare those props' properties which the child component is using to control re-rendering the child component or skip. As the React Example 6 in Appendix B, ChildOne is inherited from the regular React component, but ChildOne's shouldComponentUpdate is implemented to compare this.props.value.number and nexProps.value.number. Figure 8 shows that both child components, ChildOne and ChildTwo, are not re-rendered after the button is clicked.


React Example 6

Parent Data: 1

Children Component: ChildOne

Data from Parent: 1

Children Component: ChildTwo

Parent (2.1ms)
ChildTwo (0....
ChildOn...

Commit information

Priority: `Normal`
Committed at: `0s`
Render duration: `11.2ms`

Parent (0.4ms)

Commit information

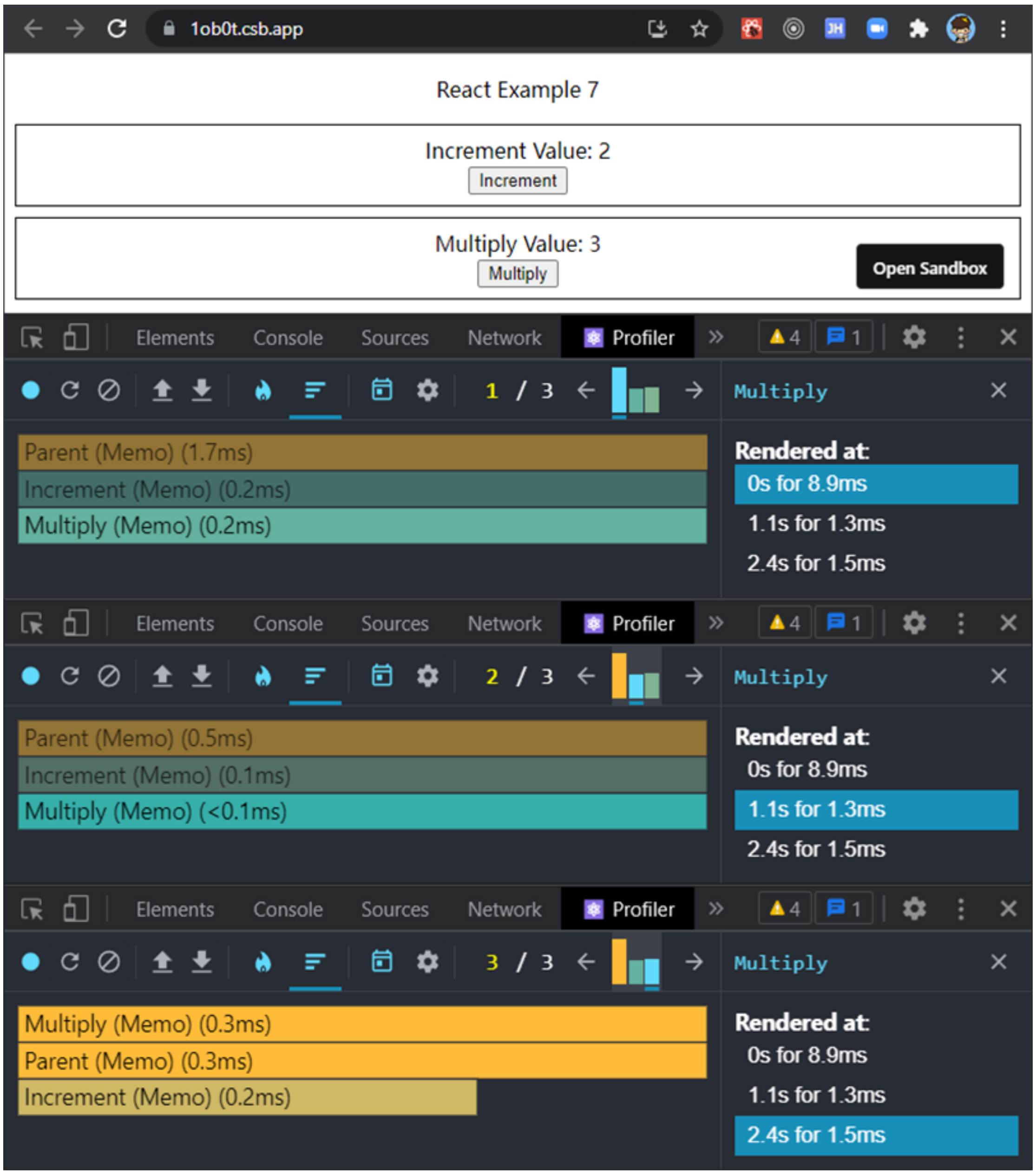
Priority: `Immediate`
Committed at: `0.6s`
Render duration: `1.2ms`

Figure 8: Profiler Result of React Example 6 in Appendix B

Although shouldComponentUpdate helps to skip unnecessary component rendering, shouldComponentUpdate has a drawback. If props or state of the component has large data, shouldComponentUpdate needs to cost time to run the comparison for each time the component tries to re-render. This will also reduce the performance of the application.

4.3 useCallback

Every time the parent component is updated, the derived function will have a new reference. Therefore, the PureComponent and the React.memo strategies will fail. As the React Example 7 in Appendix B, MemoedIncrement and MemoedMultiply are memorized version components of Increment and Multiply. When MemoedIncrement or MemoedMultiply is clicked, handleIncrement or handleMultiply is called to increase incrementValue by 1 or multiple multiplyValue by 3. Figure 9 shows profiler results for three steps. First, the page was loaded, all three components were rendered. Second, the increment button was clicked, all three components were re-rendered. Third, the multiply button was clicked, all three components were re-rendered again. Results show that references of functions are updated every time the parent component is updated. This causes that the child component which gets functions as props from the parent will be re-rendered even if the child component is memorized by React.memo.



React Example 7

Increment Value: 2

Multiply Value: 3

1 / 3 Multiply

Parent (Memo) (1.7ms)
Increment (Memo) (0.2ms)
Multiply (Memo) (0.2ms)

Rendered at:
0s for 8.9ms
1.1s for 1.3ms
2.4s for 1.5ms

2 / 3 Multiply

Parent (Memo) (0.5ms)
Increment (Memo) (0.1ms)
Multiply (Memo) (<0.1ms)

Rendered at:
0s for 8.9ms
1.1s for 1.3ms
2.4s for 1.5ms

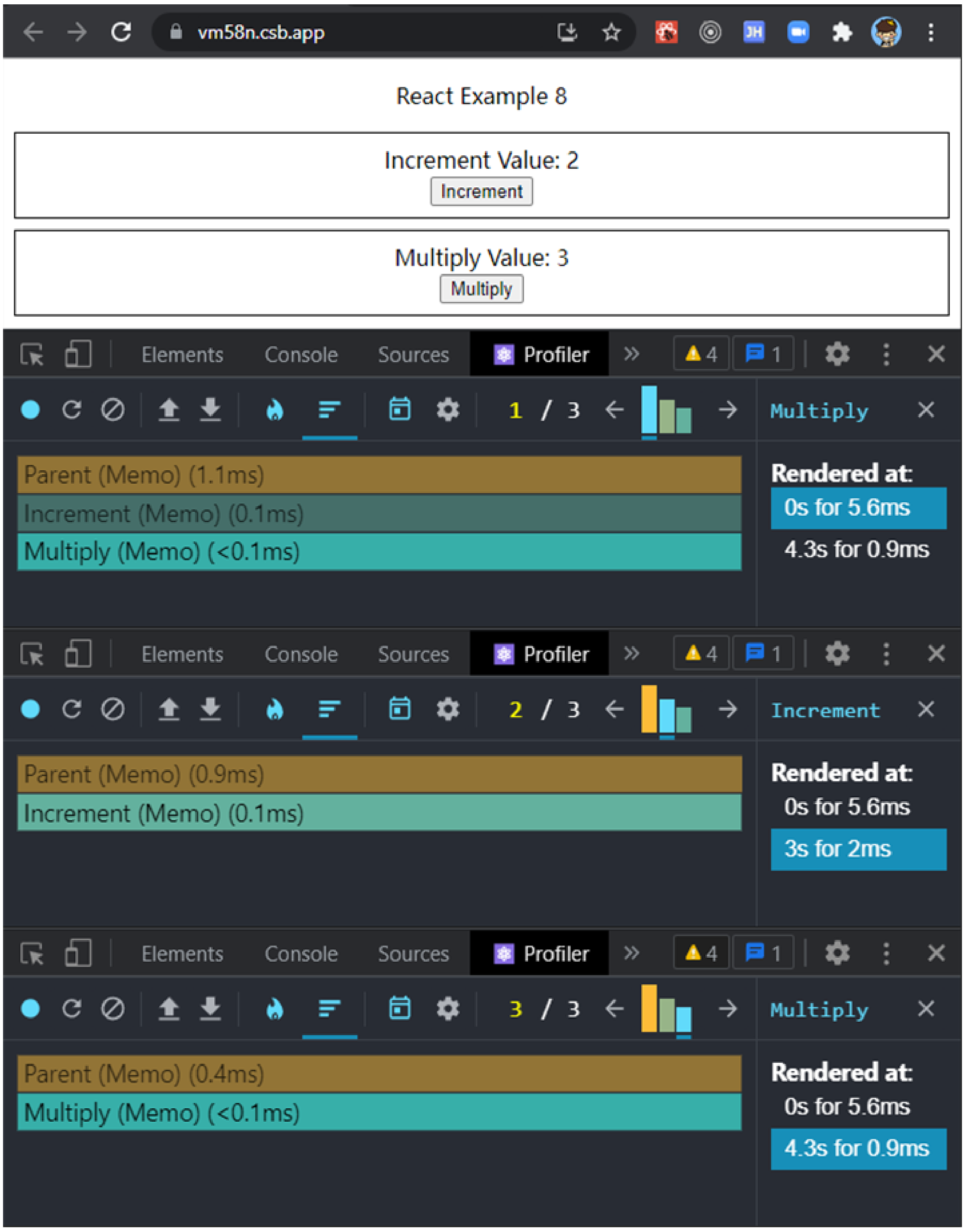
3 / 3 Multiply

Multiply (Memo) (0.3ms)
Parent (Memo) (0.3ms)
Increment (Memo) (0.2ms)

Rendered at:
0s for 8.9ms
1.1s for 1.3ms
2.4s for 1.5ms

Figure 9: Profiler Result of React Example 7 in Appendix B

useCallback is a Hook in React and can help to generate a memorized version of functions. useCallback takes a function and an array of dependencies as parameters. Memorization is a caching mechanism, useCallback remembers and caches the function which is passed as the first parameter of useCallback. And



```
Parent (Memo) (1.1ms)
Increment (Memo) (0.1ms)
Multiply (Memo) (<0.1ms)

Rendered at:
0s for 5.6ms
4.3s for 0.9ms
```

```
Parent (Memo) (0.9ms)
Increment (Memo) (0.1ms)

Rendered at:
0s for 5.6ms
3s for 2ms
```

```
Parent (Memo) (0.4ms)
Multiply (Memo) (<0.1ms)

Rendered at:
0s for 5.6ms
4.3s for 0.9ms
```

Figure 10: Profiler Result of React Example 8 in Appendix B

useCallback returns a memorized version function.[2] The memorized version function is only updated, if and only if one of the dependencies has changed.

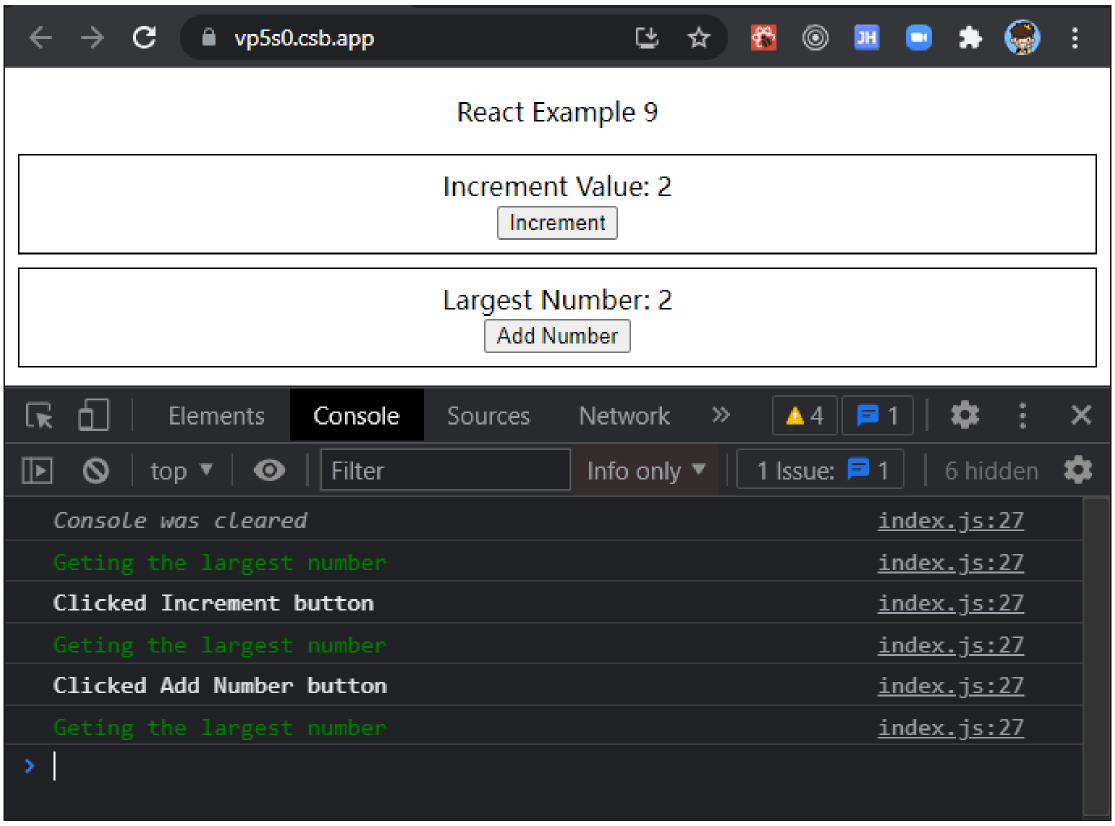
As the React Example 8 in Appendix B, both handleIncrement and handleMultiply are memorized version functions and depend on incrementValue and multiplyValue, so the reference of handleIncrement or handleMultiply will be changed only if incrementValue or multiplyValue is updated. Figure 10 shows profiler results for three steps. First, the page was loaded, all three components were rendered. Second, the increment button was clicked. Parent component and Increment component were re-rendered but not the Multiply. Third, the multiply button was clicked. Parent component and Multiply component were re-rendered but not the Increment. Therefore, useCallback can be used to generate a stable callback function to prevent unnecessary re-rendering child components in the parent component.

4.4 useMemo

useMemo is a built-in Hook in React to memorize a calculation result between a function's calls and between renders. useMemo takes a function that returns the calculation result as the first parameter and an array of dependent variables as the second parameter. The calculation result will be only recomputed if and only if one of the dependencies has changed. This optimization is intended to skip expensive calculations on every render.

As the React Example 9 in Appendix B, there are two buttons "Increment" and "Add Number". If the increment button is clicked, the incrementValue will be increased by 1. If the add number button is clicked, the pushNumber function will add a number to nums and largestNum will be recomputed by the getLargestNum function.

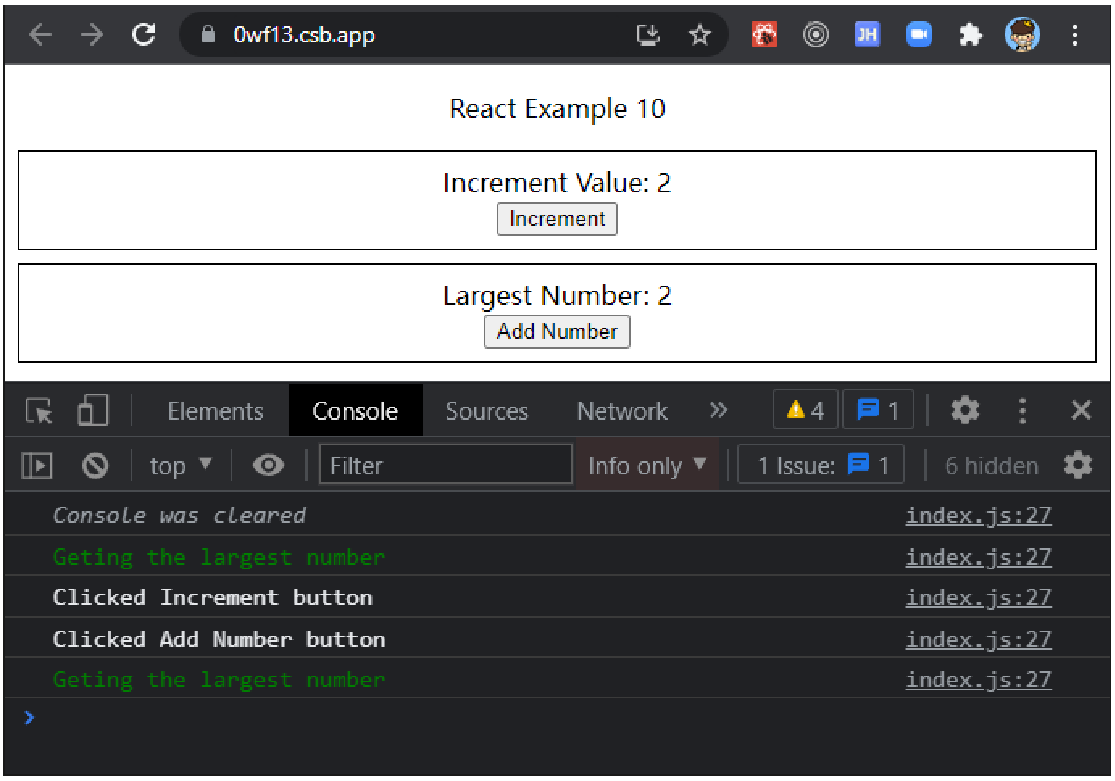
Figure 11 shows the logs of the React Example 9 in Appendix B. After the page was loaded, the getLargestNum function was triggered and returned the largest number of the array nums to largestNum. And then the Increment button is clicked, and the getLargestNum function was triggered to recompute largestNum. After that, the Add Number button is clicked, a number was pushed to the array nums. The getLargestNum was triggered again to compute the new largestNum for the new nums. The React Example 9 has a performance issue that the largestNum was recomputed when the array was not updated.



```
Console was cleared            index.js:27
Geting the largest number      index.js:27
Clicked Increment button       index.js:27
Geting the largest number      index.js:27
Clicked Add Number button      index.js:27
Geting the largest number      index.js:27
```

Figure 11: Logs of React Example 9 in Appendix B

For preventing the recomputing issue, the example can be improved by useMemo as the React Example 10 in Appendix B. The difference between React Example 9 and React Example 10 is that the largestNum is a memorized value that is returned from useMemo. useMemo takes a function that returns the result from the getLargestNum function as the first parameter and an array that contains the array nums as the second parameter. The largestNum will be updated depending on the array nums. Figure 12 which is the logs of the React Example 10 shows that the getLargestNum function was not triggered after the increment button is clicked. However, after the add number button is clicked, the array nums was updated. The getLargestNum function was triggered. In another word, the largestNum was only recomputed when the array was updated. Therefore, useMemo can help to prevent unnecessary recomputations.

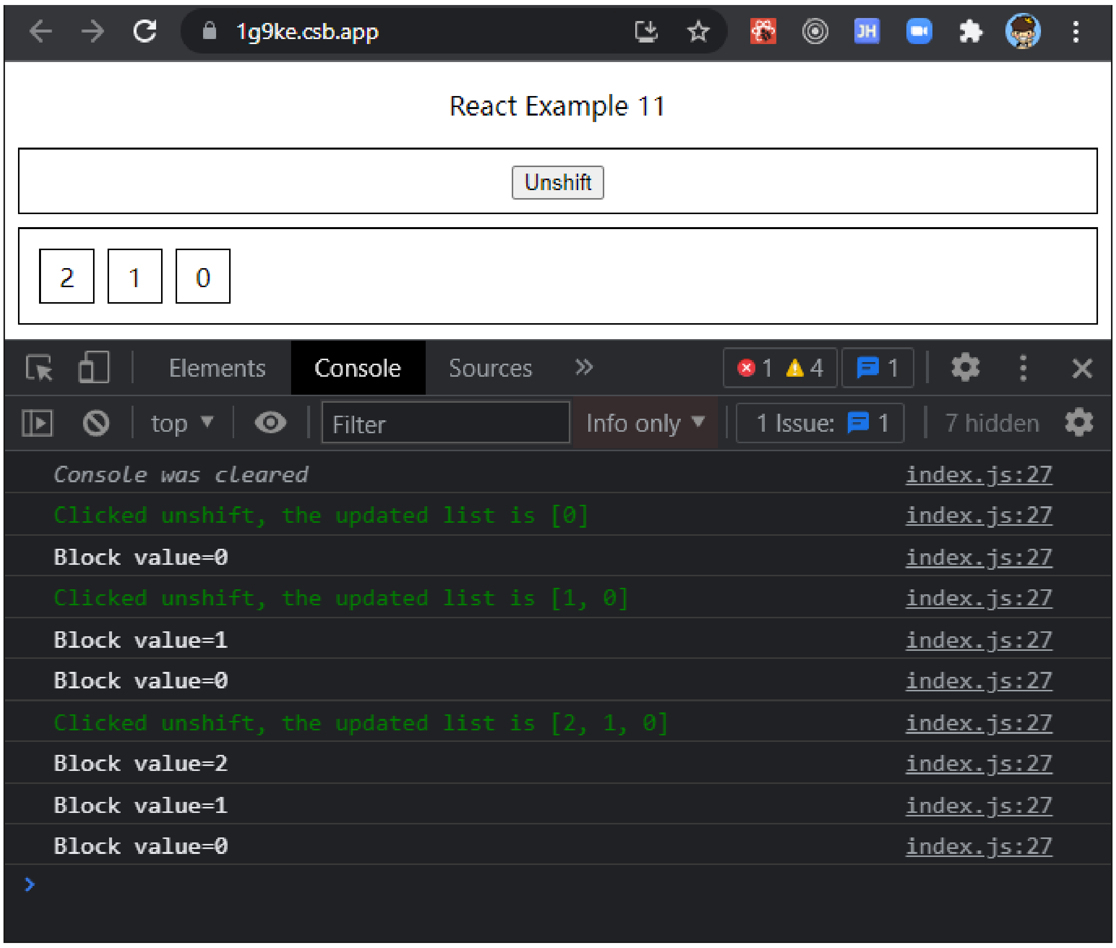


```
Console was cleared                         index.js:27
Geting the largest number                   index.js:27
Clicked Increment button                    index.js:27
Clicked Add Number button                   index.js:27
Geting the largest number                   index.js:27
>
```

Figure 12: Logs of React Example 10 in Appendix B

4.5 Key Property of List

In real development, developers usually need to render a list of components on the page. For example, some data needs to be displayed in a table, we can use the map method of Array to create an array of row components and render them into the page. As the React Example 11, each time the unshift button is clicked, the application will add the length of the current list to the beginning of the list. And list.map will create an array of MemorizedBlock which displays a number block to

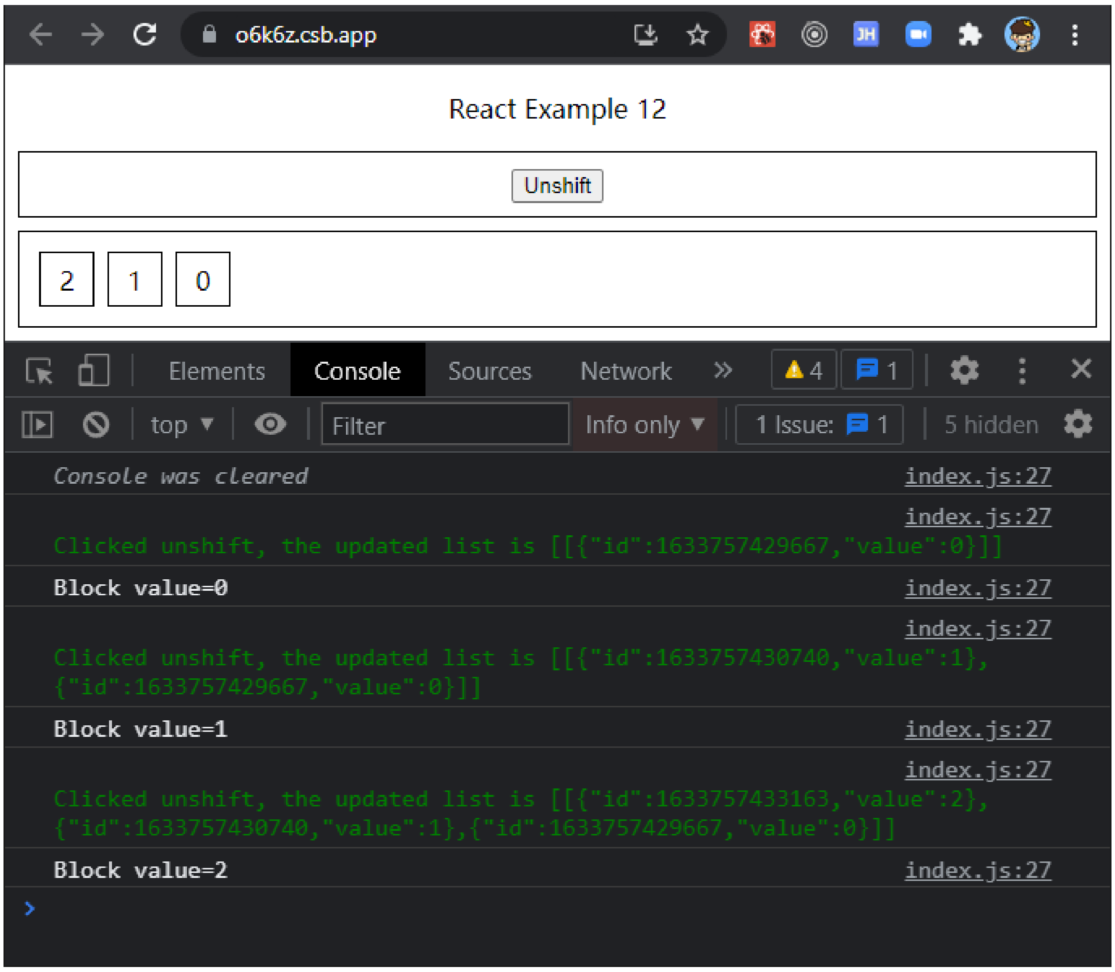


```
Console was cleared                                      index.js:27
Clicked unshift, the updated list is [0]                 index.js:27
Block value=0                                            index.js:27
Clicked unshift, the updated list is [1, 0]              index.js:27
Block value=1                                            index.js:27
Block value=0                                            index.js:27
Clicked unshift, the updated list is [2, 1, 0]           index.js:27
Block value=2                                            index.js:27
Block value=1                                            index.js:27
Block value=0                                            index.js:27
```

Figure 13: Logs of React Example 11 in Appendix B

the page. Figure 13 shows that each time the unshift button is clicked, the whole list of MemorizedBlock components would be re-rendered. This is a serious performance issue. After the last time, the unshift was clicked, the list was added a new item. React executed two times DOM update and one time DOM creation. Assuming that rendering an item needs to take 5ms, and the number of items in the list is 200. After the unshift button is clicked, React needs to execute 200 times DOM update and one time DOM creation. Therefore, the application needs to cost an extra $200 \times 5ms = 1{,}000ms = 1s$ to add the new item to the page.

The optimization of these performance issues is using the key property. For each item in the list, we can generate a unique id and use the id as the key property of each item component. As the React Example 12, each item in the list has a unique id which is generated by Date.now() which returns the number of milliseconds elapsed since the UTC of January 1, 1970. The unique ID is assigned to the key property of each MemorizedBlock component. Figure 14 shows that every time the unshift button



```
Console was cleared                                              index.js:27
                                                                 index.js:27
Clicked unshift, the updated list is [[{"id":1633757429667,"value":0}]]
Block value=0                                                    index.js:27
                                                                 index.js:27
Clicked unshift, the updated list is [[{"id":1633757430740,"value":1},
{"id":1633757429667,"value":0}]]
Block value=1                                                    index.js:27
                                                                 index.js:27
Clicked unshift, the updated list is [[{"id":1633757433163,"value":2},
{"id":1633757430740,"value":1},{"id":1633757429667,"value":0}]]
Block value=2                                                    index.js:27
```

Figure 14: Logs of React Example 12 in Appendix B

was clicked, only the component of the new item was rendered.

The key property helps React to identify which element is changed. After the last time the unshift button was clicked, only the new item with the value of 2 was rendered. During the reconciliation stage, React run the Diff algorithm to compare the virtual DOM. React found that there were two virtual DOM nodes with key values of 1633757429667 and 1633757430740 that had not been modified and did not need to be updated. And React also found that the virtual DOM node with the key value of 1633757433163 did not exist, so React only needed to create a virtual DOM node for the key value of 1633757433163. Compared to React Example 11 which does not use the key property, using the key property can save those update operations for existing items and only cost creating operations for new items.

React officially recommends using the ID of each item as the key property of the component to achieve the above optimization purpose. And React does not recommend using the index of each item as the key. For using the index as the key, if we push a new item to the beginning of the list, the whole list of components will be re-rendered because the index of the new item is 0, and indexes of other items are increased by 1. Therefore, using the index as the key cannot achieve the optimization purpose.

However, it is not always better to use the ID than the index in all scenarios. In a common paging list, the list item IDs on the first page are different than on the second page. Suppose each page displays three items of the list as Figure 15, and each item in the list has a key with a unique ID. When the second page is switched, React

```html
<!-- First Page List Items in Virtual DOM -->
<li key="a">dataA</li>
<li key="b">dataB</li>
<li key="c">dataC</li>

<!-- Second Page List Items in Virtual DOM -->
<li key="d">dataD</li>
<li key="e">dataE</li>
<li key="f">dataF</li>
```

Figure 15: Virtual DOM of Paging List Items with ID as Key

will delete all DOM nodes for the first page and create nodes for the second page because all li tags have different key values. Therefore, using a unique ID as the key property for each item in the list cannot achieve the optimization purpose. In this scenario, using the index as the key is better as Figure 16. When switching to the second page, React only needs to update all DOM nodes for the paging list because all li tags have the same key values. Compared to the example in Figure 15, the example in Figure 16 costs 3 operations less.

```html
<!-- First Page List Items in Virtual DOM -->
<li key="0">dataA</li>
<li key="1">dataB</li>
<li key="2">dataC</li>

<!-- Second Page List Items in Virtual DOM -->
<li key="0">dataD</li>
<li key="1">dataE</li>
<li key="2">dataF</li>
```

Figure 16: Virtual DOM of Paging List Items with Index as Key

Despite the above scenario, React officially recommends using ID as the key value of each item. There are two reasons as follows:

1. When deleting, inserting, or sorting items into the list, it is more efficient to use the key property with a unique ID. Page-turning operations are usually accompanied by API requests, and DOM operations cost much less

time than API requests. Therefore, whether to use ID or index as the key property has little impact on user experience in this scenario.

2. The key property with a unique ID can maintain the list item component state corresponding to the ID. For example, each row in the table has two states, ideal, and edit. At first, all rows are in ideal status. The user clicks on the edit button of the first row to enter the edit state. And then, the user drags the second row and moves it to the first row of the table. If this table uses the index as the key, the state of the first row is still in edit state. However, the user wants to edit the second row which does not meet the expectation from the user’s point of view. Although this problem can be solved by adding a property to identify which state the item is in into the data object of the item and passing this property to the component as props, it is simpler to be solved by using a unique ID as the key.

Chapter 5 - Conclusion

In this thesis, contributing the study of React Application Optimization is the main goal. Before considering optimization, it is worth understanding how React components work, the diffing algorithms, and how rendering works in React. These are all important concepts to take into consideration when optimizing React applications. For optimized strategies, this thesis focuses on how to reduce unnecessary renders such as PureComponent and React.memo, shouldComponentUpdate, useCallback and useMemo, and optimizing list with the key property. These strategies are just the tip of the iceberg of potential performance improvements and conceptually solving performance issues. There are a few other areas to implement rendering improvements that we can study in the future, for example, lazy loading components, cache application state by Service Workers, considering Service-Side-Rendering, etc.

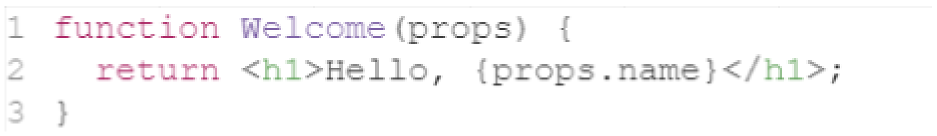
References

[1] Singh, Yashvardhan. React Framework: The best choice to build modern web apps. Retrieved from https://www.greycampus.com/blog/programming/react-for-web-development

[2] Danthasinghe, Wathsala. Performance optimization with React Hooks — useCallback & useMemo. Retrieved from https://blog.devgenius.io/performance-optimization-with-react-hooks-usecallback-usememo-f2e527651b79

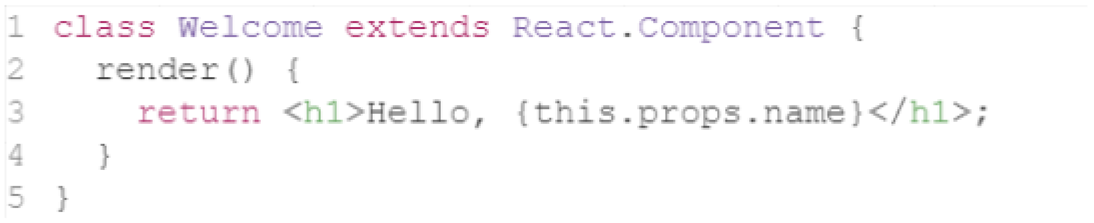
Appendix A

Function Component



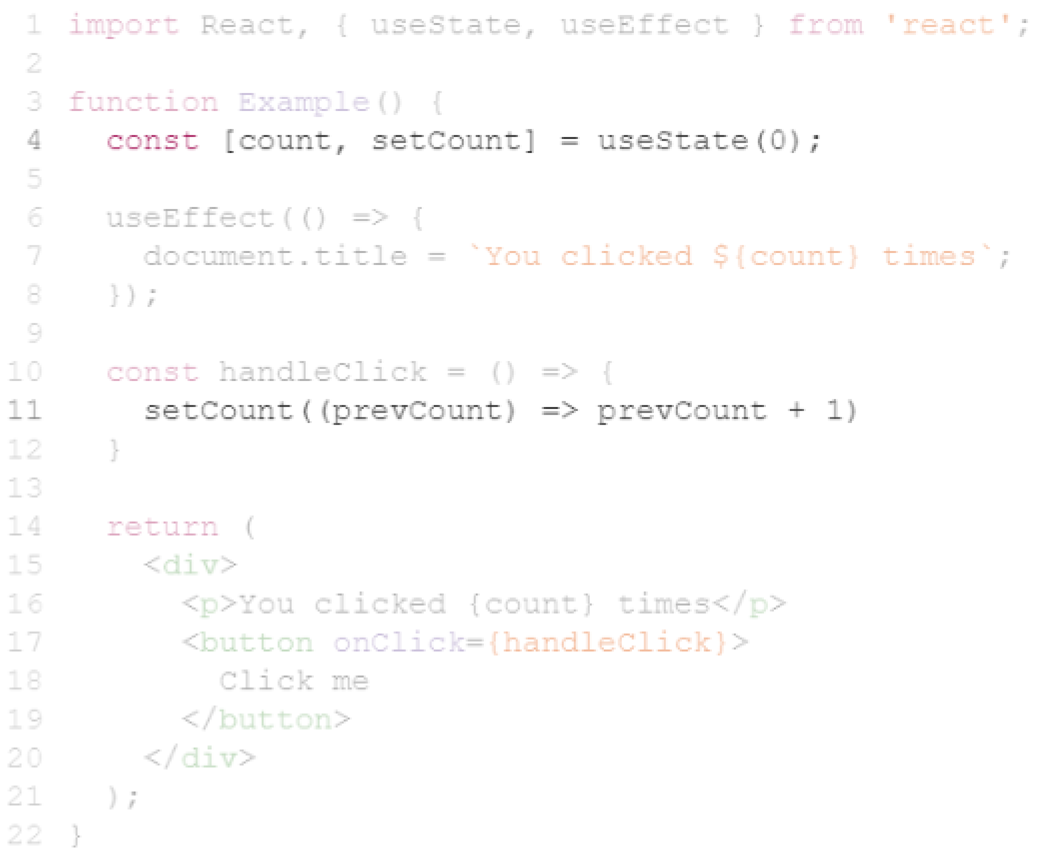
```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Class Component



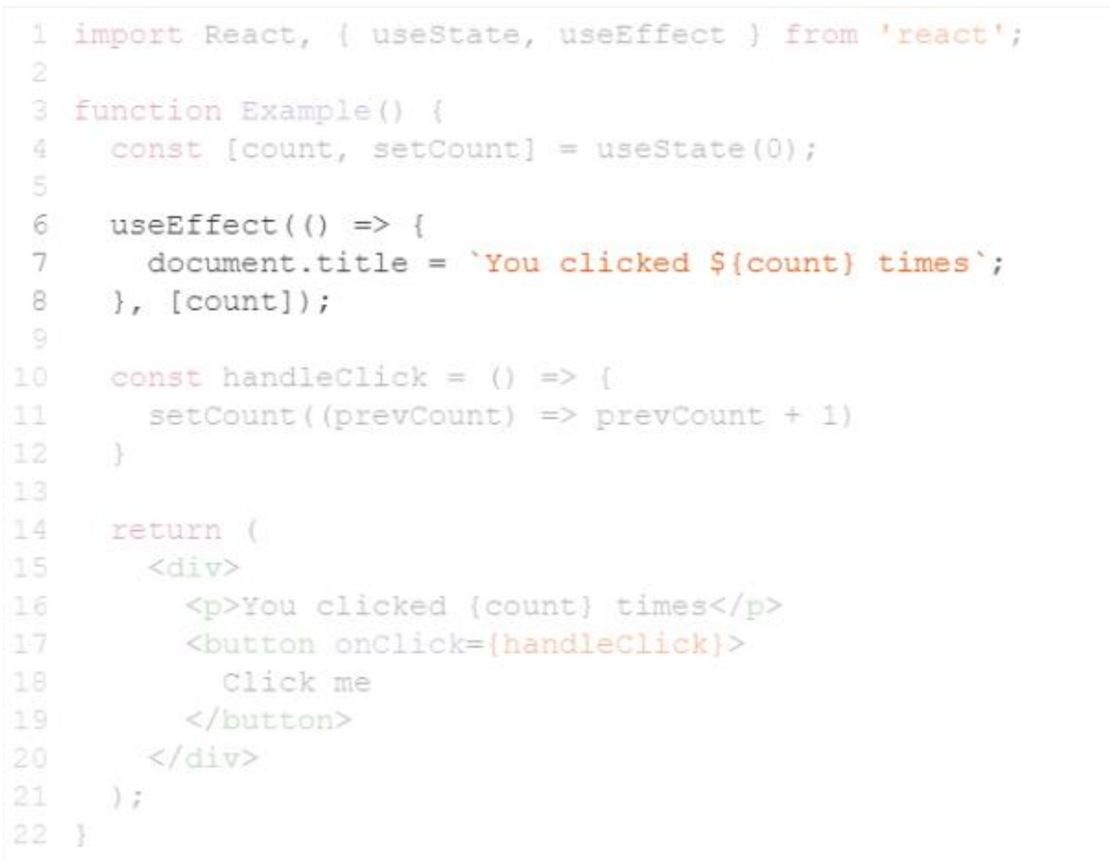
```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Function Component useState



```
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const handleClick = () => {
    setCount((prevCount) => prevCount + 1)
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
      </button>
    </div>
  );
}
```

Function Component useEffect



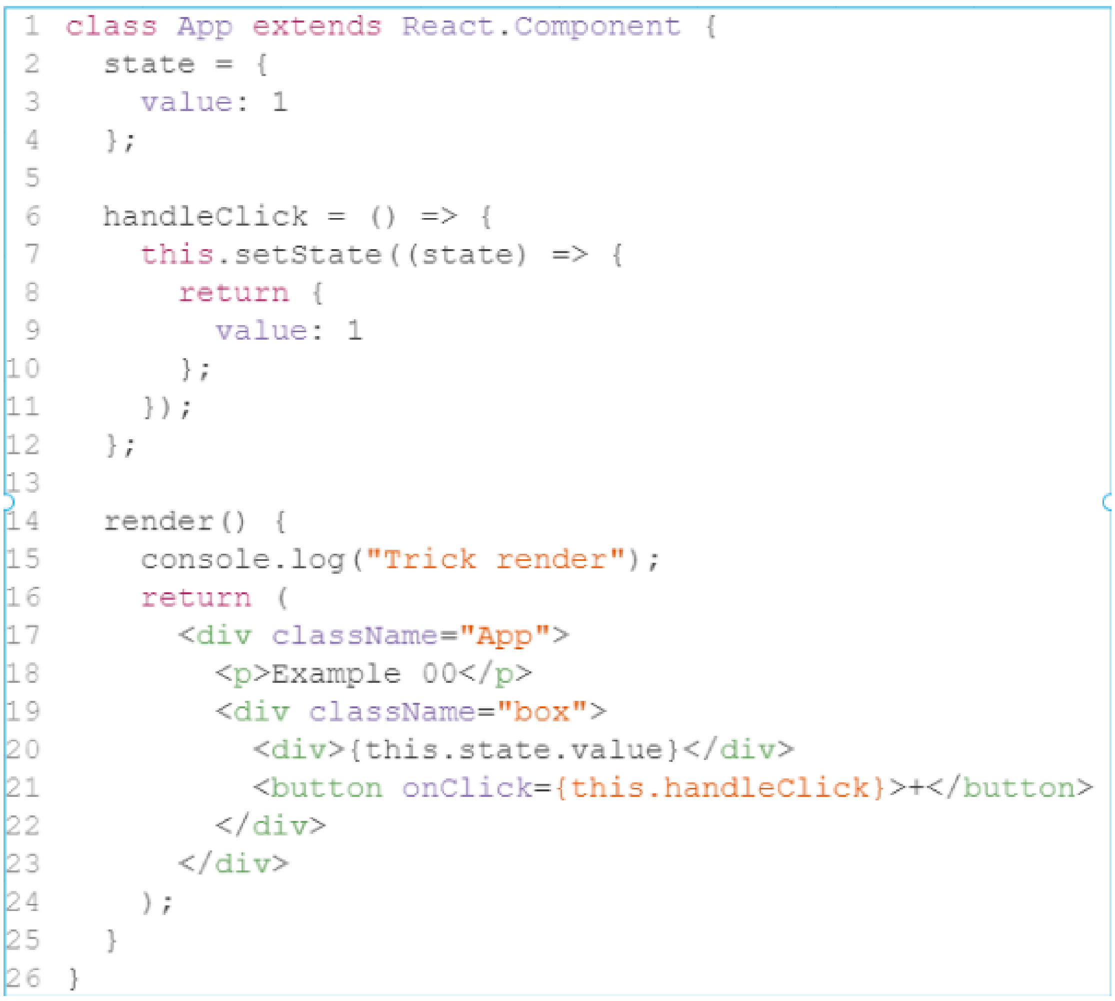
```
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]);

  const handleClick = () => {
    setCount((prevCount) => prevCount + 1)
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
      </button>
    </div>
  );
}
```

Appendix B

React Example 1

```
class App extends React.Component {
  state = {
    value: 1
  };

  handleClick = () => {
    this.setState((state) => {
      return {
        value: state.value + 1
      };
    });
  };

  render() {
    console.log("Trick render");
    return (
      <div className="App">
        <p>Example 00</p>
        <div className="box">
          <div>{this.state.value}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
      </div>
    );
  }
}
```

React Example 2

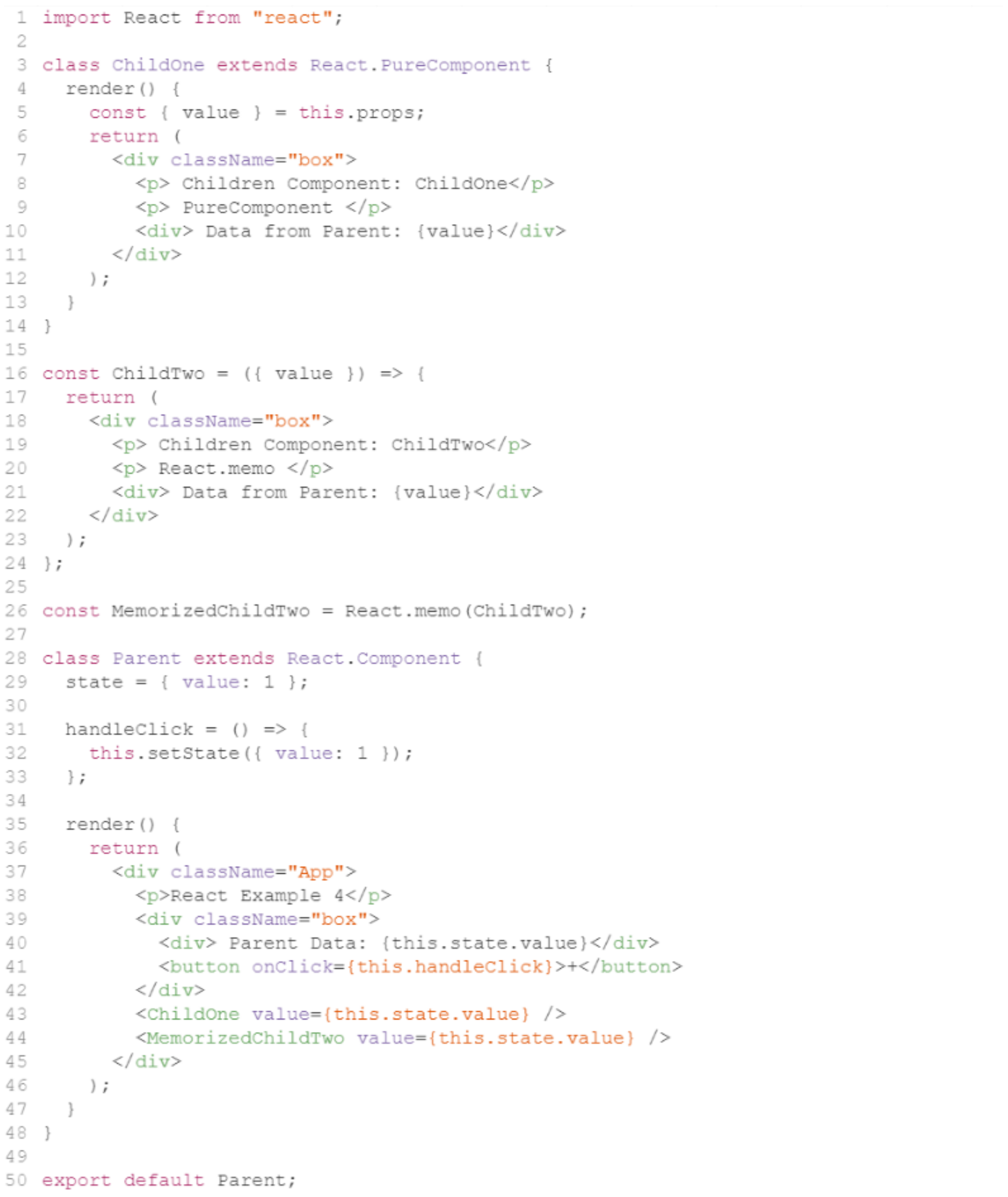


```
class App extends React.Component {
  state = {
    value: 1
  };

  handleClick = () => {
    this.setState((state) => {
      return {
        value: 1
      };
    });
  };

  render() {
    console.log("Trick render");
    return (
      <div className="App">
        <p>Example 00</p>
        <div className="box">
          <div>{this.state.value}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
      </div>
    );
  }
}
```

React Example 3

```
import React from "react";

class ChildOne extends React.Component {
  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p> Children Component: ChildOne</p>
        <div> Data from Parent: {value}</div>
      </div>
    );
  }
}

class ChildTwo extends React.Component {
  render() {
    return (
      <div className="box">
        <p> Children Component: ChildTwo</p>
      </div>
    );
  }
}

class Parent extends React.Component {
  state = { value: 1 };

  handleClick = () => {
    this.setState({ value: 1 });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 3</p>
        <div className="box">
          <div> Parent Data: {this.state.value}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
        <ChildOne value={this.state.value} />
        <ChildTwo />
      </div>
    );
  }
}

export default Parent;
```

React Example 4



```
import React from "react";

class ChildOne extends React.PureComponent {
  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p> Children Component: ChildOne</p>
        <p> PureComponent </p>
        <div> Data from Parent: {value}</div>
      </div>
    );
  }
}

const ChildTwo = ({ value }) => {
  return (
    <div className="box">
      <p> Children Component: ChildTwo</p>
      <p> React.memo </p>
      <div> Data from Parent: {value}</div>
    </div>
  );
};

const MemorizedChildTwo = React.memo(ChildTwo);

class Parent extends React.Component {
  state = { value: 1 };

  handleClick = () => {
    this.setState({ value: 1 });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 4</p>
        <div className="box">
          <div> Parent Data: {this.state.value}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
        <ChildOne value={this.state.value} />
        <MemorizedChildTwo value={this.state.value} />
      </div>
    );
  }
}

export default Parent;
```

React Example 5

```
import React from "react";

class ChildOne extends React.PureComponent {
  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p> Children Component: ChildOne</p>
        <div> Data from Parent: {value.number}</div>
      </div>
    );
  }
}

class ChildTwo extends React.PureComponent {
  render() {
    return (
      <div className="box">
        <p> Children Component: ChildTwo</p>
      </div>
    );
  }
}

class Parent extends React.Component {
  state = { value: { number: 1 } };

  handleClick = () => {
    this.setState({ value: { number: 1 } });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 5</p>
        <div className="box">
          <div> Parent Data: {this.state.value.number}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
        <ChildOne value={this.state.value} />
        <ChildTwo />
      </div>
    );
  }
}

export default Parent;
```

React Example 6

```
class Parent extends React.Component {
  state = { value: { number: 1 } };

  handleClick = () => {
    this.setState({ value: { number: 1 } });
  };

  render() {
    return (
      <div className="App">
        <p>React Example 6</p>
        <div className="box">
          <div> Parent Data: {this.state.value.number}</div>
          <button onClick={this.handleClick}>+</button>
        </div>
        <ChildOne value={this.state.value} />
        <ChildTwo />
      </div>
    );
  }
}
```

```
class ChildOne extends React.Component {
  shouldComponentUpdate(nextProps) {
    return this.props.value.number !== nextProps.value.number;
  }

  render() {
    const { value } = this.props;
    return (
      <div className="box">
        <p> Children Component: ChildOne</p>
        <div> Data from Parent: {value.number}</div>
      </div>
    );
  }
}

class ChildTwo extends React.PureComponent {
  render() {
    return (
      <div className="box">
        <p> Children Component: ChildTwo</p>
      </div>
    );
  }
}
```

React Example 7

```
import React from "react";

const Increment = (props) => (
  <button onClick={props.handleClick}>Increment</button>
);

const MemoedIncrement = React.memo(Increment);

const Multiply = (props) => (
  <button onClick={props.handleClick}>Multiply</button>
);

const MemoedMultiply = React.memo(Multiply);

const Parent = () => {
  const [incrementValue, setIncrementValue] = React.useState(1);
  const [multiplyValue, setMultiplyValue] = React.useState(1);

  const handleIncrement = () => {
    setIncrementValue(incrementValue + 1);
  };

  const handleMultiply = () => {
    setMultiplyValue(multiplyValue * 3);
  };

  return (
    <div className="App">
      <p>React Example 7</p>
      <div className="box">
        <div> Increment Value: {incrementValue}</div>
        <MemoedIncrement handleClick={handleIncrement} />
      </div>
      <div className="box">
        <div> Multiply Value: {multiplyValue}</div>
        <MemoedMultiply handleClick={handleMultiply} />
      </div>
    </div>
  );
};

export default React.memo(Parent);
```

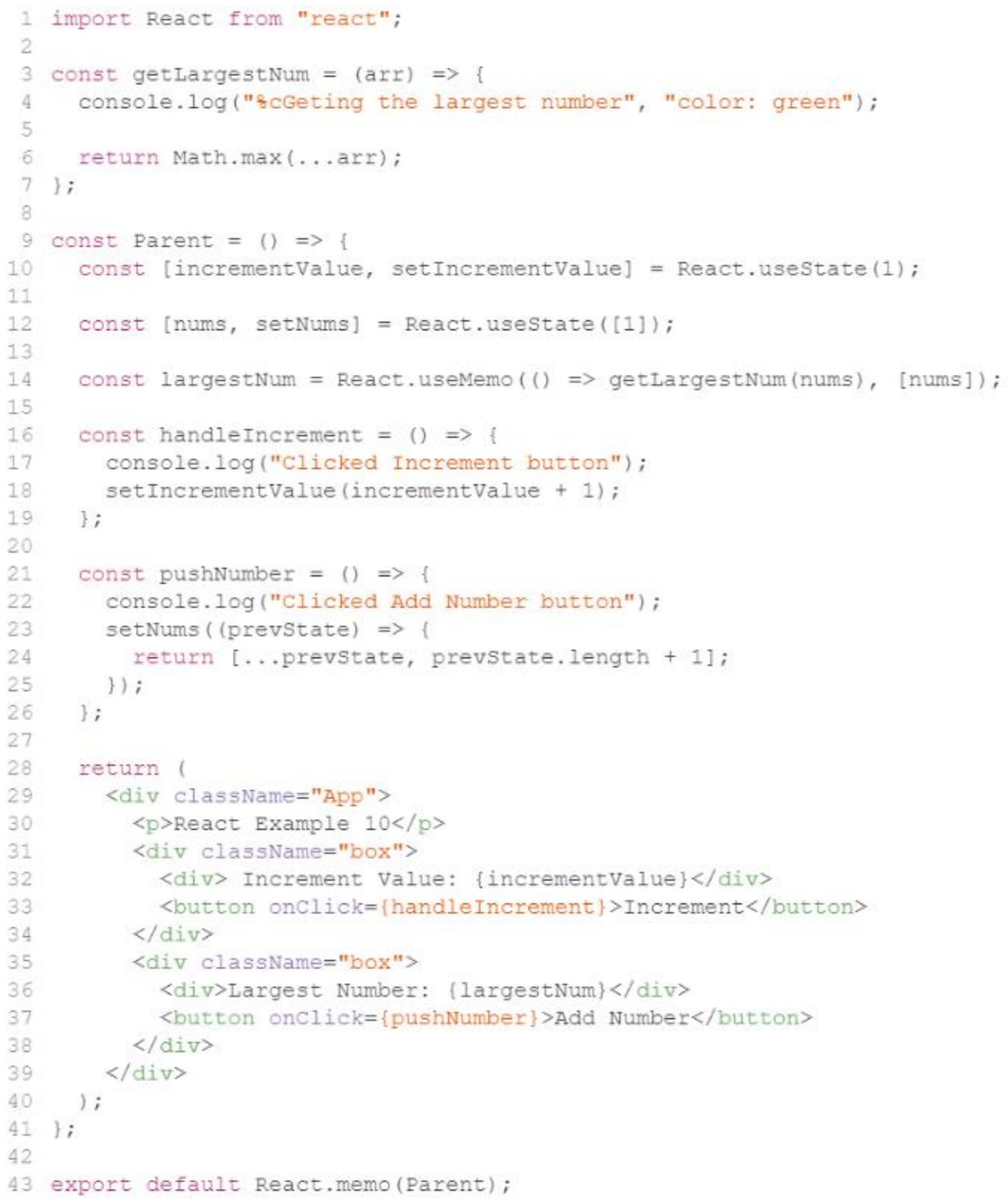
React Example 8

```
import React from "react";

const Increment = (props) => (
  <button onClick={props.handleClick}>Increment</button>
);

const MemoedIncrement = React.memo(Increment);

const Multiply = (props) => (
  <button onClick={props.handleClick}>Multiply</button>
);

const MemoedMultiply = React.memo(Multiply);

const Parent = () => {
  const [incrementValue, setIncrementValue] = React.useState(1);
  const [multiplyValue, setMultiplyValue] = React.useState(1);

  const handleIncrement = React.useCallback(() => {
    setIncrementValue(incrementValue + 1);
  }, [incrementValue]);

  const handleMultiply = React.useCallback(() => {
    setMultiplyValue(multiplyValue * 3);
  }, [multiplyValue]);

  return (
    <div className="App">
      <p>React Example 8</p>
      <div className="box">
        <div> Increment Value: {incrementValue}</div>
        <MemoedIncrement handleClick={handleIncrement} />
      </div>
      <div className="box">
        <div> Multiply Value: {multiplyValue}</div>
        <MemoedMultiply handleClick={handleMultiply} />
      </div>
    </div>
  );
};

export default React.memo(Parent);
```

React Example 9

```
import React from "react";

const getLargestNum = (arr) => {
  console.log("%cGeting the largest number", "color: green");
  return Math.max(...arr);
};

const Parent = () => {
  const [incrementValue, setIncrementValue] = React.useState(1);

  const [nums, setNums] = React.useState([1]);

  const largestNum = getLargestNum(nums);

  const handleIncrement = () => {
    console.log("Clicked Increment button");
    setIncrementValue(incrementValue + 1);
  };

  const pushNumber = () => {
    console.log("Clicked Add Number button");
    setNums((prevState) => {
      return [...prevState, prevState.length + 1];
    });
  };

  return (
    <div className="App">
      <p>React Example 9</p>
      <div className="box">
        <div> Increment Value: {incrementValue}</div>
        <button onClick={handleIncrement}>Increment</button>
      </div>
      <div className="box">
        <div>Largest Number: {largestNum}</div>
        <button onClick={pushNumber}>Add Number</button>
      </div>
    </div>
  );
};

export default React.memo(Parent);
```

React Example 10



```jsx
import React from "react";

const getLargestNum = (arr) => {
  console.log("%cGeting the largest number", "color: green");

  return Math.max(...arr);
};

const Parent = () => {
  const [incrementValue, setIncrementValue] = React.useState(1);

  const [nums, setNums] = React.useState([1]);

  const largestNum = React.useMemo(() => getLargestNum(nums), [nums]);

  const handleIncrement = () => {
    console.log("Clicked Increment button");
    setIncrementValue(incrementValue + 1);
  };

  const pushNumber = () => {
    console.log("Clicked Add Number button");
    setNums((prevState) => {
      return [...prevState, prevState.length + 1];
    });
  };

  return (
    <div className="App">
      <p>React Example 10</p>
      <div className="box">
        <div> Increment Value: {incrementValue}</div>
        <button onClick={handleIncrement}>Increment</button>
      </div>
      <div className="box">
        <div>Largest Number: {largestNum}</div>
        <button onClick={pushNumber}>Add Number</button>
      </div>
    </div>
  );
};

export default React.memo(Parent);
```

React Example 11

```
import React from "react";

function Block(props) {
  console.log(`Block value=${props.value}`);
  return <div className="block">{props.value}</div>;
}

const MemorizedBlock = React.memo(Block);

const Parent = () => {
  const [list, setList] = React.useState([]);

  const handleUnshift = () => {
    setList((prevList) => {
      const newList = [prevList.length, ...prevList];
      console.log(
        `%cClicked unshift, the updated list is [${newList.join(", ")}]`,
        "color: green;"
      );
      return newList;
    });
  };

  return (
    <div className="App">
      <p>React Example 11</p>
      <div className="box">
        <button onClick={handleUnshift}>Unshift</button>
      </div>
      <div className="container">
        {list.map((num, i) => (
          <MemorizedBlock value={num} />
        ))}
      </div>
    </div>
  );
};

export default React.memo(Parent);
```

React Example 12

```
import React from "react";

function Block(props) {
  console.log(`Block value=${props.value}`);
  return <div className="block">{props.value}</div>;
}

const MemorizedBlock = React.memo(Block);

const Parent = () => {
  const [list, setList] = React.useState([]);

  const handleUnshift = () => {
    setList((prevList) => {
      const newItem = {
        id: Date.now(),
        value: prevList.length
      };
      const newList = [newItem, ...prevList];
      console.log(
        `%cClicked unshift, the updated list is [${JSON.stringify(newList)}]`,
        "color: green;"
      );
      return newList;
    });
  };

  return (
    <div className="App">
      <p>React Example 12</p>
      <div className="box">
        <button onClick={handleUnshift}>Unshift</button>
      </div>
      <div className="container">
        {list.map(({ id, value }, i) => (
          <MemorizedBlock key={id} value={value} />
        ))}
      </div>
    </div>
  );
};

export default React.memo(Parent);
```