Week 6.1

React Hooks

In this lecture, Harkirat explored essential concepts in React development, emphasizing the significance of a single parent component for efficient reconciliation and rerendering. The discussion delved into strategies for minimizing rerenders, highlighting the use of memoization with **useMemo** and the importance of keys in array iteration. The lecture also introduced the concept of wrapper components, showcasing their role in maintaining consistent styling. Furthermore, Harkirat provided insights into the choice between class-based and functional components, concluding with a brief overview of the **useEffect** hook. These insights offer a solid foundation for understanding core React principles.

React Returns

In React, a component can only return a single root element, commonly wrapped in a parent container (like a div). This rule exists because React needs a single entry point to render and manage the component's output.

Problem Statement

```jsx
src > ⚛ App.jsx > ⬡ App
1
2   function App() {
3     return (
4       <Header title="my name is harkirat" />
5       <Header title="My name is raman" />
6     )
7   }
8
9   function Header({title}) {
10    return <div>
11      {title}
12    </div>
13  }
14
15  export default App
16
```

One of the most prominent reasons for it is Reconciliation. The single-root element rule in React facilitates the reconciliation process, where React efficiently updates the real DOM based on changes in the virtual DOM. By having a single root element, React can easily perform the comparison between the previous and current states of the virtual DOM.

Reconciliation

Reconciliation involves identifying what parts of the virtual DOM have changed and efficiently updating only those parts in the actual DOM. The single-root structure simplifies this process by providing a clear entry point for React to determine where updates should occur.

In addition to reconciliation, it aids in maintaining a straightforward and predictable structure in React components, making the code more readable and understandable. This constraint encourages developers to create components with well-defined boundaries, which enhances code organization and modularity.

While a single root element is required, React provides a feature called fragments (<></> or <React.Fragment></React.Fragment>) that allows you to group multiple elements without introducing an extra node in the real DOM. Fragments don't create an additional parent in the DOM but still satisfy the single-root rule.

Solution


```
const MyComponent = () => {

  return (

    <>
```

```
    <Header />

    <MainContent />

    <Footer />

  </>

 );

};
```

In summary, the single-root rule ensures a clear and efficient rendering process, simplifies styling and layout, and maintains consistency in React components.

Object Destructuring

Object destructuring is a feature in JavaScript that allows you to extract values from objects and assign them to variables in a more concise and convenient way. This can make your code cleaner and more readable. Here's a brief explanation with an example:

Basic Object Destructuring:

```
// Original Object

const person = { firstName: 'John', lastName: 'Doe', age: 30 };


// Destructuring

const { firstName, lastName, age } = person;


// Extracted Values

console.log(firstName); // Output: John

console.log(lastName);  // Output: Doe

console.log(age);       // Output: 30
```

Default Values:

You can also provide default values in case the property is not present in the object:

```
const { firstName, lastName, age, gender = 'Unknown' } = person;

console.log(gender); // Output: Unknown (since 'gender' is not present in the 'person' object)
```

Variable Assignment:

You can use a different variable name during destructuring:

```
const { firstName: first, lastName: last } = person;
```

```
console.log(first); // Output: John

console.log(last);  // Output: Doe
```

Nested Object Destructuring:

Destructuring also works with nested objects:

```
const student = { name: 'Alice', details: { grade: 'A', age: 21 } };


const { name, details: { grade, age } } = student;


console.log(name);  // Output: Alice

console.log(grade); // Output: A

console.log(age);   // Output: 21
```

Object destructuring provides a concise and expressive way to extract values from objects, making your code more readable and maintainable.

Re-rendering in React

Rerendering in React refers to the process of updating and rendering components to reflect changes in the application's state or props. When there's a change in the state or props of a component, React re-renders that component and any affected child components. It's important to note that a rerender doesn't necessarily mean a complete re-rendering of the entire DOM; instead, React efficiently updates only the necessary parts of the DOM.

Basically, anytime a final DOM manipulation happens or when react actually updates the DOM it is called a rerender.

When Does a Rerender Happen?

1. Changes in a state variable utilized within the component.

2. A re-render of a parent component, which subsequently triggers the re-rendering of all its child components. This cascading effect ensures synchronization throughout the component tree.

Problem Statement

- As it is known, we use React to create dynamic websites, which is often achieved through the use of components that can respond to user interactions, state changes, or incoming props.

- A crucial principle guiding efficient React applications is the minimization of unnecessary rerenders. Rerenders occur when there are alterations in a component's state or props, and the rule of thumb is to keep these rerenders to a minimum for optimal performance.

- Consider a scenario with a webpage featuring a counter button, a text element reflecting the change in the count state and a static "Hello, World!" text. While clicking the counter button might trigger a rerender of the text element containing the count due to state changes, it's essential to prevent the unnecessary rerendering of static elements.

Solutions

There are broadly 2 ways of minimizing the amount of rerenders

1. Push the State down

2. By Using Memoization

Pushing the State Down:

Pushing the state down in React refers to the practice of managing state at the lowest possible level in the component tree. By doing so, you localize the state to the components that absolutely need it, reducing unnecessary re-renders in higher-level components.

When state is kept at a higher level in the component tree, any changes to that state can trigger re-renders for all child components, even if they don't directly use or depend on that particular piece of state. However, by pushing the state down and ensuring that each component only has access to the state it needs, you can minimize the impact of state changes on the overall component tree.
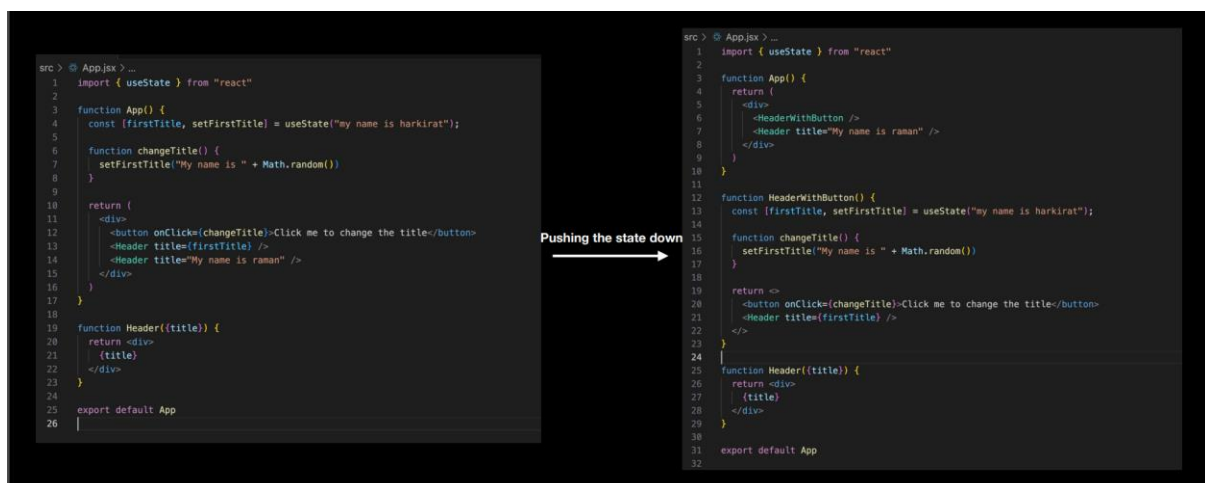

For example, if a specific piece of state is only relevant to a small portion of your application, keeping that state localized to the components in that section prevents unnecessary re-renders elsewhere. This practice contributes to a more efficient and performant React application.

By Using Memoization:

The above problem of reducing the number of rerenders can also be tackled using Memoization. Memoization in React, achieved through the **useMemo** hook, is a technique used to optimize performance by memoizing (caching) the results of expensive calculations. This is particularly useful when dealing with computations that don't need to be recalculated on every render, preventing unnecessary recalculations and re-renders.

In the context of minimizing re-renders, **useMemo** is often employed to memoize the results of computations derived from state or props. By doing so, you can ensure that the expensive computation is only performed when the dependencies (specified as the second argument to **useMemo**) change.
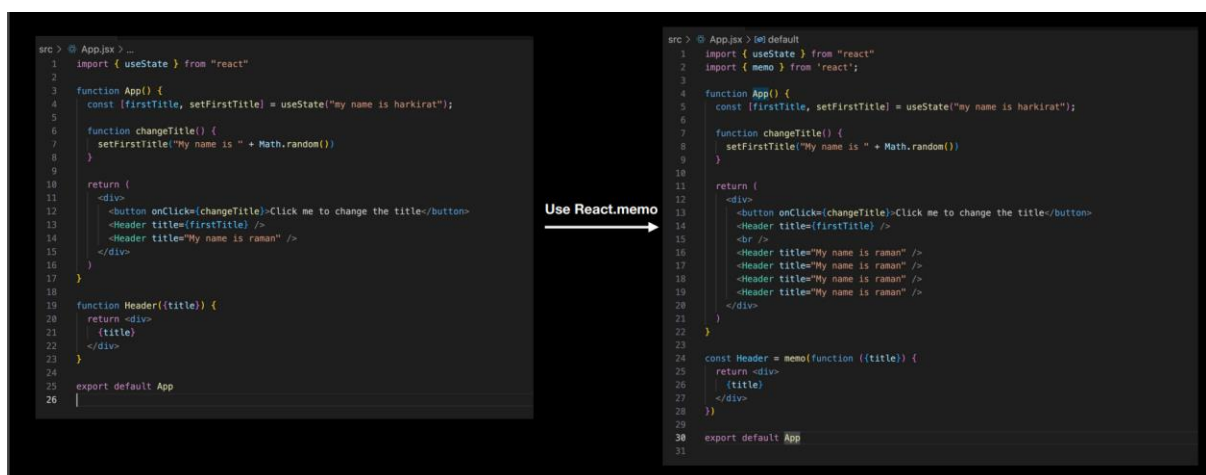
By using **useMemo**, you can strategically memoize computations to optimize performance and minimize the impact of re-renders in React.

Significance of Key in React

In React, when rendering a list of elements using the map function, it is crucial to assign a unique key prop to each element. The "key" is a special attribute that helps React identify which items have changed, been added, or been removed. This is essential for efficient updates and preventing unnecessary re-renders of the entire list.

When the key prop is not provided or not unique within the list, React can't efficiently track the changes, leading to potential issues in the application's performance and rendering.

Here's a simple example illustrating the importance of keys in a todo app:

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

const TodoList = ({ todos }) => (
  <ul>
    {todos.map(todo => (
      // Each todo item needs a unique key
      <li key={todo.id}>{todo.text}</li>
    ))}
  </ul>
);

const App = () => {
  const [todos, setTodos] = useState([
    { id: 1, text: 'Learn React' },
    { id: 2, text: 'Build a Todo App' },
    { id: 3, text: 'Deploy to production' },
  ]);

  const addTodo = () => {
    // Simulating adding a new todo
    const newTodo = { id: todos.length + 1, text: 'New Todo' };
    setTodos([...todos, newTodo]);
  };

  return (
    <div>
      <button onClick={addTodo}>Add Todo</button>
      <TodoList todos={todos} />
    </div>
  );
};
```

export default App;

In this example, each todo item in the list has a unique id that serves as the key prop. When a new todo is added, the key helps React efficiently update and re-render only the necessary parts of the list, maintaining performance and ensuring a smooth user experience.

Wrapper Components

In React, wrapper components are used to encapsulate and group common styling or thematic elements that need to be applied consistently across different parts of an application. These components act as containers for specific sections or functionalities, allowing for a clean and modular structure.

Let's consider an example where we have a wrapper component called Card that provides a consistent styling for various content sections, such as blog posts. The Card component maintains the overall styling, while different contents can be dynamically injected.

```
// CardWrapper.js

import React from 'react';

const CardWrapper = ({ children }) => {
  return (
    <div style={{ border: '1px solid #ccc', padding: '16px', margin: '16px', borderRadius: '8px' }}>
      {children}
    </div>
  );
};

export default CardWrapper;
```

Now, we can use this CardWrapper component to create specific cards for different content, such as blog posts, by providing the dynamic content as children:

```
// BlogPost.js

import React from 'react';
import CardWrapper from './CardWrapper';

const BlogPost = ({ title, content }) => {
  return (
    <CardWrapper>
      <h2>{title}</h2>
      <p>{content}</p>
    </CardWrapper>
  );
};

export default BlogPost;
```

With this structure, we maintain a consistent card styling across different sections of our application, promoting reusability and making it easy to manage the overall theme. This approach is especially beneficial when you want to keep a uniform appearance for similar components while varying their internal content.

Class Components vs Functional Components

In React, components are the building blocks of a user interface. There are two main types of components: class-based components and functional components.

1. **Class-Based Components:**

   - Class-based components are ES6 classes that extend from React.Component.
   - They have access to the lifecycle methods provided by React, such as componentDidMount, componentDidUpdate, and componentWillUnmount.
   - State and lifecycle methods are managed within class-based components.
   - They were the primary type of components before the introduction of hooks in React 16.8.

Example of a class-based component:

```
import React, { Component } from 'react';

class MyClassComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      // state initialization
    };
  }

  componentDidMount() {
    // code to run after component mounts
  }

  render() {
    return <div>Hello from class-based component</div>;
  }
}

export default MyClassComponent;
```

1. **Functional Components:**

   - Functional components are simpler and more concise. They are essentially JavaScript functions that take props as an argument and return React elements.

   - With the introduction of React hooks in version 16.8, functional components gained the ability to manage state and use lifecycle methods through hooks like useState and useEffect.

   - They are generally easier to read and write.

Example of a functional component:

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

```
const MyFunctionalComponent = () => {

  const [state, setState] = useState(/* initial state */);


  useEffect(() => {

    // code to run after component mounts or when state/props change

  }, [/* dependencies */]);


  return <div>Hello from functional component</div>;
};


export default MyFunctionalComponent;
```

Note:

- Functional components are now the preferred way to write components in React due to their simplicity and the additional capabilities provided by hooks.

- Hooks like useState and useEffect allow functional components to manage state and perform side effects, making them as powerful as class-based components.

- Class-based components are still used in some codebases, especially in projects that haven't migrated to functional components or are working with older React versions.

React Hooks

React Hooks are functions that allow functional components in React to have state and lifecycle features that were previously available only in class components. Hooks were introduced in React 16.8 to enable developers to use state and other React features without writing a class.

Using these hooks, developers can manage state, handle side effects, optimize performance, and create more reusable and readable functional components in React applications. Each hook serves a specific purpose, contributing to a more modular and maintainable codebase.

Some commonly used React Hooks are: useEffect, useMemo, useCallback, useRef, useReducer, useContext, useLayoutEffect

useEffect()

useEffect is a React Hook used for performing side effects in functional components. It is often used for tasks such as data fetching, subscriptions, or manually changing the DOM. The useEffect hook accepts two arguments: a function that contains the code to execute, and an optional array of dependencies that determines when the effect should run.
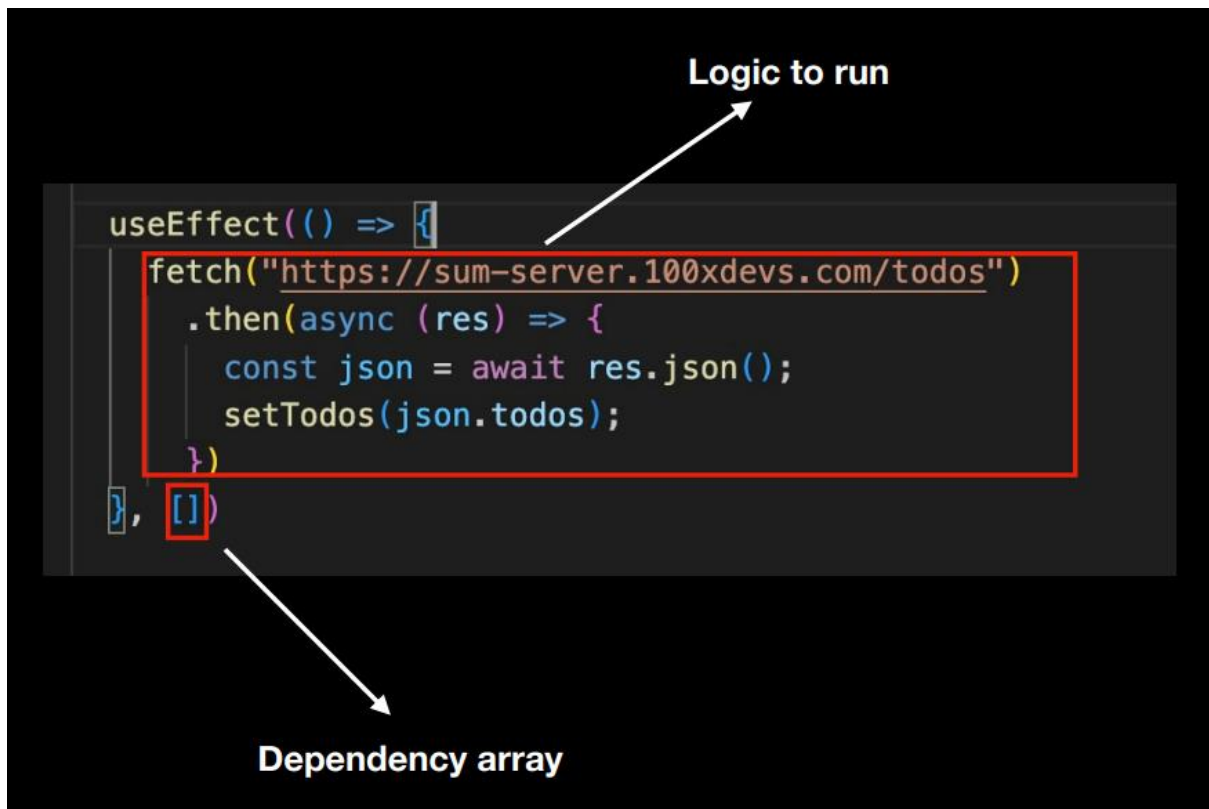
Here's an example of how to use useEffect:

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


const DataFetcher = () => {
 const [data, setData] = useState(null);


 useEffect(() => {
   // Effect will run after the component is mounted
   const fetchData = async () => {
    try {
      // Simulating a data fetching operation
      const response = await fetch('<https://api.example.com/data>');
      const result = await response.json();
      setData(result);
    } catch (error) {
```

```
      console.error('Error fetching data:', error);

    }

  };


  fetchData();


  // Effect cleanup (will run before unmounting)

  return () => {

    console.log('Component will unmount. Cleanup here.');

  };

}, []); // Empty dependency array means the effect runs once after mount


  return (

   <div>

    {data ? (

     <p>Data: {data}</p>

    ) : (

     <p>Loading data...</p>

    )}

   </div>

  );

};


export default DataFetcher;
```

In this example:

1. We import useState and useEffect from 'react'.

2. Inside the DataFetcher component, we use useState to manage the state of the data variable.

3. The useEffect hook is employed to perform the data fetching operation when the component is mounted. The empty dependency array [] ensures that the effect runs only once after the initial render.

4. The fetchData function, declared inside the effect, simulates an asynchronous data fetching operation. Upon success, it updates the data state.

5. The component returns content based on whether the data has been fetched.

useEffect is a powerful tool for managing side effects in React components, providing a clean way to handle asynchronous operations and component lifecycle events.