React Component LifeCyle:

React components go through a lifecycle that includes four phases: initialization, mounting, updating, and unmounting. Each phase has specific methods that can be overridden to perform actions during that phase.

1. Initialization - In this phase, the component is constructed with props and state.

2. Mounting Phase — In the mounting phase, a component is being created and inserted into the DOM for the first time

3. Update Phase — The update phase is triggered when a component's state or props change

4. Unmount Phase — The unmount phase is triggered when a component is removed from the DOM

Detailed explanation

1. Initialization

In this phase, the component is constructed with props and state.

constructor(props) - This is the first method that's called when the component is created. It's the perfect place to set up your component's initial state and bind methods to the component.

```
import React, { Component } from 'react';
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      // initial state
    };
    console.log("Constructor: Initialization phase");
  }

  render() {
    return <div>Hello, World!</div>;
  }
}
```

2. Mounting

Mounting is the phase when the component is being inserted into the DOM. The following methods are called in this order during mounting:

a. componentWillMount (deprecated in React 17+)

b. render - This method is called and returns the JSX that represents the component in the DOM.

c. componentDidMount - This method is called after the component has been added to the DOM. We use it to fetch data or set up event listeners.

```
class MyComponent extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = {
      // initial state
    };
  }

  componentDidMount() {
    console.log("Component did mount: Mounted phase");
    // Perform API calls or other setup tasks here
  }

  render() {
    console.log("Render: Mounting phase");
    return <div>Hello, World!</div>;
  }
}
```

3. Updating

Updating is the phase when the component is being re-rendered as a result of changes to props or state. The following methods are called in this order during updating:

a. componentWillReceiveProps (deprecated in React 17+)

b. shouldComponentUpdate - This method is called before the component is re-rendered.

c. componentWillUpdate (deprecated in React 17+)

d. render - This method is called next and returns the JSX that represents the updated component in the DOM.

e. componentDidUpdate - This method is called after the component has been re-rendered. We use it to perform any side effects like updating DOM or fetching new data.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: 0
    };
  }

  componentDidUpdate(prevProps, prevState) {
    console.log("Component did update: Updating phase");
    // Perform operations after update here
  }

  incrementCounter = () => {
    this.setState({ counter: this.state.counter + 1 });
  }

  render() {
    console.log("Render: Updating phase");
    return (
      <div>
        <p>Counter: {this.state.counter}</p>
```

```
        <button onClick={this.incrementCounter}>Increment</button>
      </div>
    );
  }
}
```

4. Unmounting

The unmount phase is the final phase in React lifecycle, which is triggered when a component is removed from the DOM. This can happen when a component is removed due to a parent component being removed or when the component is conditionally rendered and its condition becomes false.

The following method is called during unmounting:

componentWillUnmount - which allows us to perform any necessary cleanup operations such as removing event listeners or canceling network requests.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      // initial state
    };
  }

  componentDidMount() {
    console.log("Component did mount: Mounted phase");
  }

  componentWillUnmount() {
    console.log("Component will unmount: Unmounting phase");
    // Cleanup tasks like removing event listeners, cancelling network
requests, etc.
  }

  render() {
    console.log("Render: Mounting phase");
    return <div>Hello, World!</div>;
  }
}
```

Full Example:

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

export class MyComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        counter: 0
```

```
      };
      console.log("Constructor: Initialization phase");
    }

    componentDidMount() {
      console.log("Component did mount: Mounted phase");
    }

    componentDidUpdate(prevProps, prevState) {
      console.log("Component did update: Updating phase");
    }

    componentWillUnmount() {
      console.log("Component will unmount: Unmounting phase");
    }

    incrementCounter = () => {
      this.setState({ counter: this.state.counter + 1 });
    }

    render() {
      console.log("Render: Mounting/Updating phase");
      return (
        <div>
          <p>Counter: {this.state.counter}</p>
          <button onClick={this.incrementCounter}>Increment</button>
        </div>
      );
    }
  }
```

The above example demonstrates the React component lifecycle with log statements to observe when each method is called. This can be useful for debugging and understanding the component's behavior during its lifecycle.

PureComponent in React

A PureComponent in React is a type of class component that implements a shallow comparison of props and state to decide whether the component should update. This can improve performance by avoiding unnecessary re-renders when the props and state have not changed.

Shallow Comparison

A shallow comparison checks if the primitive values of props and state are the same. For objects and arrays, it checks if the references are the same, not the content.

We use the **shouldComponentUpdate()** Lifecycle method to customize the default behavior and implement it when the React component should re-render or update itself.

Example of Creating a PureComponent: (MyPureComponent.js)

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

class MyPureComponent extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
      data: { value: 10 }
    };
  }

  incrementCount = () => {
    this.setState({ count: this.state.count + 1 });
  };

  updateData = () => {
    this.setState({ data: { value: this.state.data.value + 1 } });
  };

  render() {
    console.log("Rendering MyPureComponent");
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <h2>Data Value: {this.state.data.value}</h2>
        <button              onClick={this.incrementCount}>Increment
Count</button>
        <button onClick={this.updateData}>Update Data</button>
      </div>
    );
  }
}

export default MyPureComponent;
```

Using the PureComponent: (App.js)

```
import React from 'react';
import ReactDOM from 'react-dom';
```

```
import MyPureComponent from './MyPureComponent';

const App = () => (
  <div>
    <MyPureComponent />
  </div>
);

ReactDOM.render(<App />, document.getElementById('root'));
```

Explanation

Constructor and Initial State:

The component initializes state with count and data objects.

Incrementing Count:

The incrementCount method updates the count in the state. When the button is clicked, this method is called, and the component re-renders because the state has changed.

Updating Data:

The updateData method updates the data object in the state. Even though the object reference changes, the content remains the same.

Key Points

Shallow Comparison:

The shallow comparison checks if this.state.count and this.state.data references have changed. If they haven't, the component does not re-render.

Performance Improvement:

If PureComponent notices that the props or state haven't changed, it skips the render method, thus improving performance.

Additional Example with Props

To demonstrate how PureComponent works with props, consider the following example:

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

class MyPureComponent extends PureComponent {
  render() {
    console.log("Rendering MyPureComponent");
    return (
      <div>
        <h1>{this.props.title}</h1>
      </div>
```

```
    );
  }
}

class ParentComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      title: 'Hello, World!'
    };
  }

  updateTitle = () => {
    this.setState({ title: 'Hello, World!' });
  };

  render() {
    return (
      <div>
        <MyPureComponent title={this.state.title} />
        <button onClick={this.updateTitle}>Update Title</button>
      </div>
    );
  }
}

export default ParentComponent;
```

Explanation

ParentComponent:

This component maintains the title in its state and passes it down to MyPureComponent as a prop.

Updating Title:

The updateTitle method sets the title to the same value it already holds. Since the value hasn't changed, MyPureComponent doesn't re-render due to the shallow comparison performed by PureComponent.

Conclusion

PureComponent is useful for optimizing performance by preventing unnecessary re-renders.

It performs a shallow comparison of props and state, which is efficient for determining if an update is needed.

By using PureComponent, you can make your React applications more efficient, especially in scenarios where components receive props or manage state frequently.

Higher Order Components (HOCs)

Higher-order components are a design pattern in React where a function takes a component as an argument and returns an enhanced new component.

In simpler terms, HOCs are functions that wrap existing components, providing them with additional props or behaviors.

The main benefit of HOCs is that they enable us to extend the functionality of multiple components without repeating the same code in each of them. This promotes code reuse and enhances the maintainability of your React applications.

Benefits of Higher Order Components

Reusability: HOCs allow you to encapsulate shared functionalities and apply them to multiple components, promoting code reuse.

Separation of Concerns: HOCs help maintain separate responsibilities, enabling your components to focus on their specific tasks.

Code Abstraction: HOCs abstract common logic from components, making them more concise and easier to understand.

Composability: You can combine various HOCs to compose complex functionalities into your components.

Higher Order Component Syntax

```
import React from 'react';

const withExtraProps = (WrappedComponent) => {
  return (props) => {
    return <WrappedComponent {...props} extraProp="This is an extra prop" />;
  };
};

export default withExtraProps;
```

Example Code:

HOCex.js

```
import Counterclick from "./Counterclick";
import Counterhover from "./Counterhover";

function HOCex() {
    return (
        <>
            <Counterclick/>
            <Counterhover/>
        </>
```

```
    );
}

export default HOCex;
```

Explanation:

HOCex.js is the main component that renders two components: Counterclick and Counterhover. Both of these components utilize the HOC pattern to manage and share state logic.

CounterHandler.js

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

function CounterHandler(BaseComponent) {
    const NewComponent = () =>{
        const [count, setCount] = useState(0);

        const incrementCount = () =>{
            setCount(count+1);
        }

        return <BaseComponent count = {count} incrementCount =
{incrementCount}/>
    };

    return NewComponent;
}

export default CounterHandler;
```

Explanation:

CounterHandler.js is a Higher Order Component (HOC) that adds counting functionality to any base component.

Description: This HOC adds state and functionality for counting to the provided BaseComponent.

useState: Manages the count state.

incrementCount: Increments the count state.

NewComponent: A new component that wraps the BaseComponent and provides it with count and incrementCount props.

CounterClick.js

```
import CounterHandler from "./CounterHandler";

function Counterclick(props) {
```

```
    return (
        <>
            <p>Click Counts: {props.count}</p>
            <button onClick={props.incrementCount}>
                Increment
            </button>
        </>
    );
}

export default CounterHandler(Counterclick);
```

Explanation:

CounterClick.js is a component that displays the number of times a button is clicked.

Description: This component displays the current count and a button to increment the count.

props.count: The current count, provided by the HOC.

props.incrementCount: Function to increment the count, provided by the HOC.

Counterhover.js

```
import CounterHandler from "./CounterHandler";

function Counterhover(props) {

    return (
        <>
            <p>Hover Counts: {props.count}</p>
            <button onMouseOver={props.incrementCount}>
                Increment
            </button>
        </>
    );
}

export default CounterHandler(Counterhover);
```

Explanation:

Counterhover.js is a component that displays the number of times a button is hovered over.

Description: This component displays the current count and a button to increment the count when hovered over.

props.count: The current count, provided by the HOC.

props.incrementCount: Function to increment the count, provided by the HOC.

By using HOCs, you can encapsulate and reuse logic across multiple components. In this example, CounterHandler provides the counting logic to both Counterclick and Counterhover, demonstrating the power and flexibility of HOCs in React.

Introduction to React Hooks

React Hooks are a feature introduced in React 16.8 that allow you to use state and other React features in functional components. Hooks provide a more powerful and flexible way to manage state, lifecycle methods, and other side effects in React components.

Why Hooks?

Before hooks, React components could be either function components (stateless) or class components (stateful). Class components could manage state and lifecycle methods, but they often led to more complex and less readable code.

Hooks were introduced to solve these problems and provide several benefits:

Simplified State Management: Hooks allow you to manage state directly in functional components.

Better Code Reusability: Hooks enable the reuse of stateful logic without the need for higher-order components (HOCs) or render props.

Improved Readability: Hooks simplify the component code by avoiding the verbosity of classes.

Basic Hooks

Here are some of the most commonly used React hooks:

1. useState

2. useEffect

3. useContext

4. useRef

5. useMemo

Understanding State in React

In React, the state represents the data that a component manages and can change over time. Traditionally, state management was primarily associated with class components. However, with the introduction of hooks, functional components can now also manage state.

Syntax

```
const [state, setState] = useState(initialState);
```

state: The current state value.

setState: A function that allows you to update the state.

initialState: The initial value of the state, which can be any data type (e.g., string, number, array, object).

Let's grub into examples to understand the versatility and practical application of the useState hook in React.

Example 1: Managing a Counter

In this example, we'll explore how to use the useState hook to create a simple counter in a React functional component. We'll initialize a state variable for the count and use the setCount function to update its value when a button is clicked.

This example will demonstrate the foundational usage of the useState hook for managing the basic state within a functional component, providing a clear understanding of its syntax and functionality.

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

function Counter() {
  // Declare a state variable named "count" and a function to update
it
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Explanation:

The above code snippet features a React functional component named Counter, which utilizes the useState hook to manage a numerical state representing a count.

Within the Counter function, a state variable called count is declared using the useState hook, initialized with a default value of 0.

The component's return statement includes a paragraph element that dynamically displays the current value of the count using the {count} placeholder.

Additionally, an "Increment" button is rendered, and its onClick event is set to a function that increments the value of count by 1 using the setCount function.

When the button is clicked, the setCount function is called with the updated value of the count, effectively incrementing the count by 1.

This example demonstrates how the useState hook can be utilized to manage a numerical state and enable user-triggered incrementing of the count within a React functional component.

Example 2: Handling Boolean State

Here, we'll delve into handling a boolean state using the useState hook.

We'll create a toggle functionality that switches between true and false when a button is clicked, showcasing how the useState hook can handle boolean values and enable dynamic user interactions.

This example will illustrate the versatility of the useState hook in managing different types of states within functional components.

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

function Toggle() {
  const [isToggled, setIsToggled] = useState(false);

  return (
    <div>
      <p>The toggle is {isToggled ? 'on' : 'off'}</p>
      <button                    onClick={()                    =>
setIsToggled(!isToggled)}>Toggle</button>
    </div>
  );
}
```

The above code snippet features a React functional component named Toggle, which employs the useState hook to manage a boolean state.

Within the Toggle function, a state variable called isToggled is declared using the useState hook, initialized with a default value of false. The component's return statement includes a paragraph element that dynamically displays the current state of the toggle based on the value of isToggled.

Additionally, a Toggle button is rendered, and its onClick event is set to a function that toggles the value of isToggled using the setIsToggled function.

When the button is clicked, the setIsToggled function is called with the opposite value of isToggled, effectively toggling the state between true and false.

This example demonstrates how the useState hook can be utilized to manage a simple boolean state and enable user-triggered toggling of the state within a React functional component.

Example 3: Managing a Dynamic Todo List

In this example, we'll tackle a more complex scenario by using the useState hook to manage a dynamic list of todos.

We'll add new todos, removing existing ones, and updating the state to reflect the changes. This example will demonstrate how the useState hook can handle more complicated state management, such as managing arrays and updating states based on user interactions.

By exploring this example, you will be understanding of how the useState hook can be utilized for managing dynamic data within a functional component, paving the way for building interactive and responsive user interfaces in React.

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

function TodoList() {
  const [todos, setTodos] = useState([]);
  const [inputValue, setInputValue] = useState('');
```

```
  const addTodo = () => {
    setTodos([...todos, inputValue]);
    setInputValue('');
  };

  const removeTodo = (index) => {
    const newTodos = [...todos];
    newTodos.splice(index, 1);
    setTodos(newTodos);
  };

  return (
    <div>
      <ul>
        {todos.map((todo, index) => (
          <li key={index}>
            {todo}
            <button onClick={() => removeTodo(index)}>Remove</button>
          </li>
        ))}
      </ul>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
      <button onClick={addTodo}>Add Todo</button>
    </div>
  );
}
```

In the above code snippet, within the component, two state variables are declared using the useState hook: todos to store an array of todos and inputValue to hold the current input value for adding new todos.

The addTodo function appends a new todo to the todos state using the setTodos function and then clears the inputValue by setting it to an empty string. On the other hand, the removeTodo function removes a todo from the todos state based on its index by creating a new array of todos using the filter method and updating the todos state with the new array.

The component renders an input field for adding new todos, a button to trigger the addition of todos, and an unordered list to display the todos.

Each todo in the list is accompanied by a Remove button, which, when clicked, triggers the removeTodo function with the index of the todo.

This example demonstrates how the useState hook can efficiently manage dynamic states within a React functional component, enabling the addition and removal of todos based on user interactions.

useEffect Hook

The useEffect hook in React is a powerful tool for handling side effects in functional components. Side effects are any operations that interact with the outside world, like data fetching, subscriptions, or manually altering the DOM.

useEffect allows you to perform such actions in a way that is both efficient and organized.

useEffect helps us in a better way to handle what happens when a component starts, updates, or ends, in a much simpler way.

The basic syntax of useEffect:

```
useEffect(() => {
  // Your code for the side effect goes here.
  // This code runs after every component render.

  return () => {
    // Optional cleanup code goes here.
    // This part runs when the component unmounts.
  };
}, [dependencies]); // The effect will re-run only if these dependencies change.
```

Effect Function: The function you pass to useEffect will run after the render is committed to the screen. This makes it perfect for updating the DOM, fetching data, setting up subscriptions, etc.

Dependency Array: The second argument is an array of dependencies. useEffect will only re-run if these dependencies have changed between renders. If you pass an empty array [], the effect runs once after the initial render, mimicing the behavior of componentDidMount in class components.

Cleanup Function: Optionally, your effect function can return a cleanup function. React calls this function before re-running the effect and when the component unmounts. It's the ideal place for cleanup operations, similar to componentWillUnmount in class components.

Key Points

Runs After Render: By default, useEffect runs after every render.

Dependencies: Providing a dependencies array allows you to control when the effect runs.

Cleanup: Returning a function from the effect allows you to clean up side effects (e.g., clearing timers, unsubscribing from events) when the component unmounts or before the effect runs again.

Multiple Effects: You can use multiple useEffect hooks in a single component to separate concerns.

Real-Time Examples: Understanding useEffect in Action

Example 1: Fetching Data on Mount (componentDidMount)

The useEffect hook allows us to perform actions when the component first renders, akin to componentDidMount in class components. Let's see this in action with data fetching:

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

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

    useEffect(() => {
        fetch('https://jsonplaceholder.typicode.com/posts/1')
            .then(response => response.json())
            .then(post => setData(post));
    }, []);

    if (!data) {
        return <div>Loading...</div>;
    }

    return <div>Title: {data.title}</div>;
}
```

This example demonstrates how useEffect can be used for fetching data when the component mounts. We used an empty dependency array to ensure that the effect runs only once, similar to componentDidMount.

Example 2: Responding to State Changes (componentDidUpdate)

useEffect can be used to perform an action in response to a state change, mirroring the functionality of componentDidUpdate.

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

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

    useEffect(() => {
        document.title = `Count: ${count}`;
    }, [count]);

    return (
        <div>
            <p>Count: {count}</p>
            <button     onClick={()     =>     setCount(count     +
1)}>Increment</button>
        </div>
    );
}
```

This Counter component uses useEffect to update the document title every time the count state changes. The dependency on [count] ensures the effect behaves similarly to componentDidUpdate, reacting to changes in specific values.

Example 3: Cleanup Operations (componentWillUnmount)

useEffect can be used to clean up or perform final actions before the component unmounts, like componentWillUnmount.

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

function TimerComponent() {
    const [seconds, setSeconds] = useState(0);

    useEffect(() => {
        const intervalId = setInterval(() => {
            setSeconds(prevSeconds => prevSeconds + 1);
        }, 1000);

        return () => clearInterval(intervalId);
    }, []);

    return <div>Timer: {seconds} seconds</div>;
}
```

The TimerComponent sets up a timer that increments every second. The return function in useEffect is a cleanup function that clears the interval, ensuring no side effects are left when the component unmounts, similar to componentWillUnmount.

These examples show just how easy it is to use the useEffect hook for different parts of a component's life in React. Whether it's fetching data when a component starts, updating things when state changes, or cleaning up when a component goes away, useEffect makes it all straightforward.

Understanding useEffect with Different Dependency Types

The useEffect hook's behavior can vary significantly with different types of dependencies. Let's look into how it behaves with objects, null, undefined, and primitive types like strings and numbers.

1. No Dependency Array

If you omit the dependency array, useEffect runs after every render.

```
useEffect(() => {
    console.log('This runs after every render');
});
```

Without a dependency array, the effect runs after every render of the component. This is useful when you have an effect that needs to update every time the component updates, regardless of what caused the update.

2. Empty Array []

An empty array means the effect runs once after the initial render, similar to componentDidMount.

```
useEffect(() => {
    console.log('This runs once, after the initial render');
```

```
}, []);
```
With an empty array, the effect behaves like a componentDidMount lifecycle method in class components. It runs once after the initial render and then never runs again. This is ideal for one-time setup operations.

3. Array with Specific Values

When you include specific values in the array, the effect runs when those values change.

```
const [count, setCount] = useState(0);

useEffect(() => {
    console.log('This runs when "count" changes');
}, [count]);
```
Here, the effect runs whenever the count state changes. It's a way to make your effect respond specifically to changes in certain data, similar to componentDidUpdate for specific props or state values.

4. Using Objects {} as Dependencies

React's dependency array doesn't perform deep comparisons. This means when you use objects as dependencies, React only compares their reference, not their content.

```
const [user, setUser] = useState({ name: "John", age: 30 });

useEffect(() => {
    console.log('Effect runs if the "user" object reference changes');
}, [user]);
```
In this snippet, useEffect will re-run only if the user object's reference changes, not its content. For instance, if you update the user's age while keeping the same object reference, the effect won't re-run. This behavior often leads to bugs and is generally not recommended unless you intentionally want to track object reference changes.

5. null and undefined

Using null or undefined as a dependency has a specific effect: the dependency array behaves as if it's not there.

```
useEffect(() => {
    console.log('This runs after every render, like having no
dependency array');
}, null); // or undefined
```
In this case, the effect runs after every render of the component, which is the default behavior when no dependency array is provided. It's akin to having an effect without any dependencies listed.

Understanding how different dependency types affect useEffect is crucial for using it effectively. While primitive types like strings and numbers are straightforward and reliable, objects require careful handling due to shallow comparison. Meanwhile, null and undefined effectively remove the dependency check, causing the effect to run after every render. Knowing these nuances helps in writing more efficient and bug-free React components.

useContext Hook

React useContext hook provides a way to pass data through the component tree without having to pass props down manually at every level.

Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

useContext syntax:

```
const value = useContext(MyContext);
```

Here,

MyContext: The context object that was created using React.createContext().

value: The current context value, based on the nearest matching MyContext.Provider above the calling component in the tree.

Creating and Using Context

To use the useContext hook, follow these steps:

1. Create a Context

2. Provide a Context Value

3. Consume the Context Value with useContext

Step 1: Create a Context

First, create a context object using React.createContext().

```
import React from 'react';

const MyContext = React.createContext();
```

Step 2: Provide a Context Value

Next, wrap your component tree with a MyContext.Provider and pass a value to the provider.

```
import React, { useState } from 'react';
import MyContext from './MyContext';

function App() {
  const [value, setValue] = useState('Hello, World!');

  return (
    <MyContext.Provider value={value}>
      <ChildComponent />
    </MyContext.Provider>
  );
}

export default App;
```

Step 3: Consume the Context Value with useContext

Now, you can use the useContext hook to consume the context value in any functional component.

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

function ChildComponent() {
  const value = useContext(MyContext);

  return (
    <div>
      <p>The context value is: {value}</p>
    </div>
  );
}

export default ChildComponent;
```

Toggling theme Example

Let's create a more comprehensive example to understand the useContext hook.

Creating the Context

```
import React from 'react';

const ThemeContext = React.createContext();

export default ThemeContext;
```

Providing the Context Value

```
import React, { useState } from 'react';
import ThemeContext from './ThemeContext';
import Toolbar from './Toolbar';

function App() {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

export default App;
```

Consuming the Context Value

```
import React, { useContext } from 'react';
import ThemeContext from './ThemeContext';

function Toolbar() {
  return (
    <div>
      <ThemeButton />
    </div>
  );
}

function ThemeButton() {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button
      onClick={toggleTheme}
      style={{
        backgroundColor: theme === 'light' ? '#fff' : '#333',
        color: theme === 'light' ? '#000' : '#fff',
      }}
    >
      Toggle Theme
    </button>
  );
}

export default Toolbar;
```

Explanation

ThemeContext: A context object created using React.createContext().

App Component: Provides the context value (theme and toggleTheme function) using ThemeContext.Provider.

Toolbar Component: Consumes the context value using the useContext hook to access the theme and toggleTheme function.

ThemeButton Component: Uses the context values to toggle the theme and apply styles based on the current theme.

Benefits of useContext

Simplified Access: Provides a simpler and cleaner way to access context values compared to the traditional Context.Consumer component.

Readability: Makes the code more readable and easier to understand by reducing nesting and boilerplate code.

Reusability: Enhances the reusability of components by decoupling them from specific context implementations.

useRef Hook

By now you're likely familiar with React state and its reactivity. We commonly use the useState hook to update state in React when we want our component to remember something. The issue arises when updating the state triggers a complete re-render of the component. While this reactivity is a core feature of React, there are situations where we want to update the state without triggering a re-render.

Take, for example, a stopwatch. Updating the state and re-rendering the stopwatch is desirable, but when a user clicks the pause button, we might want to track the pause duration without visibly re-rendering the component.

When you want component to remember some state but you do not want that update to trigger a re-render.

Just import it and save it in a variable, additionally you can mount that on your component as well.

Syntax:

```
import {useRef} from "react";
const ref=useRef(0) // we set intial value to 0
// This useRef will return object called current like {current:0}
```

If we console.log(ref.current), we will get 0.

The 'current' property in a ref is mutable; you can store anything inside it — whether an object, function, or any JavaScript-supported data type

What's interesting is that the value saved inside the 'current' property, which we can both read and write, won't trigger a re-render. React will remember this value throughout every render cycle, meaning it persists for the component's entire lifecycle.

Example 1 : Managing mutable data

Consider a counter that increments with each button click:

```
import React from 'react';

function Counter() {
  const countRef = useRef(0); // Store count in a ref

  function incrementCount() {
    countRef.current = countRef.current + 1; // Modify ref value
  }

  return (
    <div>
      <p>Count: {countRef.current}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

In this example, useRef is used to create a reference called countRef to store the count value. The incrementCount function increments the count value by accessing countRef.current and modifying it directly.

Example 2: Accessing / Updating DOM

```
import React from 'react';

function MyComponent() {
  const inputRef = useRef(null); // Create a ref

  function focusInput() {
    inputRef.current.focus(); // Access and modify the ref value
  }

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

In this example, useRef is employed to create a reference called inputRef and associate it with the <input> element. This association grants access to the DOM element directly using inputRef.current.

When the Focus Input button is clicked, the focusInput function retrieves the input element and invokes its focus() method to bring it into focus.

Difference between useState and useRef:

useState	useRef
Triggers a re-render when the state is updated.	Does not trigger a re-render when the ref is updated.
Returns a state variable that is immutable.	Returns a mutable object with a **current** property.
State is reset on each re-render.	Data stored in current persists across re-renders.

When to Use useRef:

Avoiding Unnecessary Renders:

Use useRef when you need to store a value that doesn't trigger a re-render, useful for tracking values without impacting the UI.

Persisting Data:

When you want a value to persist across re-renders without resetting, useRef is a suitable choice.

Imperative Operations:

For accessing and manipulating the DOM directly or for imperative operations where reactivity is not necessary.

Storing Mutable Data:

When dealing with mutable data that doesn’t need to be part of the component’s state.

useMemo Hook

The useMemo hook is used to memoize the result of a function so that the result can be reused when the inputs to the function have not changed. This can be particularly useful when dealing with computationally expensive operations or when working with large datasets

Example: Memoizing Expensive Computations

Consider a scenario where you have a React component that performs a complex computation, such as generating a Fibonacci sequence. The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, typically starting with 0 and 1. Computing the Fibonacci sequence for a large number can be computationally expensive. Let's see how the useMemo hook can be used to optimize this computation.

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

const FibonacciComponent = ({ n }) => {
  const calculateFibonacci = (num) => {
    if (num <= 1) return 1;
    return calculateFibonacci(num - 1) + calculateFibonacci(num - 2);
  };

  const memoizedResult = useMemo(() => calculateFibonacci(n), [n]);

  return <div>The {n}th Fibonacci number is {memoizedResult}</div>;
};
```

In this example, the calculateFibonacci function computes the nth Fibonacci number. By using the useMemo hook, we memoize the result of the calculateFibonacci function based on the input 'n'. If 'n' remains unchanged, useMemo will return the memoized result, avoiding unnecessary recalculations. This optimization can significantly improve the performance of the FibonacciComponent, especially when 'n' remains constant across renders.

Best Practices for Using useMemo

While the useMemo hook can be a powerful tool for optimizing performance, so it's important to use it judiciously. Here are some best practices for using useMemo effectively:

Measure performance impact: Use performance profiling tools to measure the impact of useMemo on the application's performance. This can help identify areas where memoization is most beneficial and where it may be unnecessary.

Keep dependencies minimal: When using useMemo, be mindful of the dependencies array. Only include the variables that the memoized value depends on, and avoid unnecessary dependencies that could trigger unnecessary recalculations.

Use memoization for expensive computations: Reserve the use of useMemo for computationally expensive operations or functions that are called frequently, such as complex filtering and sorting algorithms.

Custom Hooks

Custom hooks are a powerful feature in React that allow you to encapsulate and reuse logic across multiple components.

They enable you to extract component logic into reusable functions, making your code more modular, readable, and maintainable.

Custom hooks leverage the basic hooks like useState, useEffect, useContext, and others, and can call other custom hooks as well.

Benefits of Custom Hooks

Reusability: Encapsulate logic in a function that can be reused across multiple components.

Separation of Concerns: Keep component logic separated from UI logic, making components simpler and more focused.

Readability: Abstract complex logic into custom hooks, making the codebase easier to understand and maintain.

Testing: Simplifies testing by allowing you to test the hook logic independently of the UI.

Example 1 : Creating custom Hook for Toggling text

Let's create a simple custom hook that toggles a boolean state. This can be useful for handling visibility toggles, like showing or hiding elements.

Step 1: Create the Custom Hook

```
import { useState } from 'react';

function useToggle(initialValue = false) {
  const [value, setValue] = useState(initialValue);

  const toggle = () => {
    setValue((prevValue) => !prevValue);
  };

  return [value, toggle];
}

export default useToggle;
```

useToggle: The custom hook that manages a boolean state.

initialValue: The initial value of the toggle state, defaulting to false.

useState: Manages the toggle state.

toggle: A function that toggles the state between true and false.

Step 2: Use the Custom Hook in a Component

Let's use the useToggle custom hook to create a component that toggles the visibility of some text.

```
import React from 'react';
import useToggle from './useToggle';

function ToggleComponent() {
  const [isVisible, toggleVisibility] = useToggle();

  return (
    <div>
      <button onClick={toggleVisibility}>
        {isVisible ? 'Hide' : 'Show'} Text
      </button>
      {isVisible && <p>This text is visible!</p>}
    </div>
  );
}

export default ToggleComponent;
```

ToggleComponent: A component that uses the useToggle custom hook to manage the visibility of some text.

useToggle: The custom hook is called without arguments, so it defaults to false. It returns the current state (isVisible) and a function to toggle the state (toggleVisibility).

Button: A button that toggles the visibility of the text when clicked. The button text changes based on the visibility state.

Conditional Rendering: The text is conditionally rendered based on the isVisible state.

Example 2: Custom Hook for Fetching Data

Let's create a custom hook that fetches data from an API.

Step 1: Create the Custom Hook

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get(url);
        setData(response.data);
        setLoading(false);
      } catch (err) {
        setError(err);
        setLoading(false);
      }
    };
```

```
    fetchData();
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

useFetch: The custom hook that takes a url as an argument.

useState: Manages the data, loading, and error states.

useEffect: Fetches data from the given url when the hook is used.

Step 2: Use the Custom Hook in a Component

```
import React from 'react';
import useFetch from './useFetch';

function DataFetchingComponent() {
  const { data, loading, error } = 
useFetch('https://api.example.com/data');

  if (loading) return <p>Loading data...</p>;
  if (error) return <p>Error fetching data: {error.message}</p>;

  return (
    <div>
      <h1>Data</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default DataFetchingComponent;
```

DataFetchingComponent: A component that uses the useFetch custom hook to fetch data from an API.

useFetch: The custom hook is called with the API URL and returns data, loading, and error states.

Conditional Rendering: Displays loading and error messages appropriately, and renders the fetched data.

Rules for Creating Custom Hooks

There are certain rules and best practices you need to follow to ensure that custom hooks work correctly and efficiently.

Naming Convention:

Always start the name of the custom hook with the word use. This is a convention that React relies on to differentiate hooks from regular functions. For example, useFetch, useToggle, useForm.

Call Hooks Only at the Top Level:

Do not call hooks inside loops, conditions, or nested functions. Hooks should always be called at the top level of the custom hook or component to maintain a consistent order across multiple renders.

This rule ensures that hooks are called in the same order each time a component renders, which is crucial for React's hooks implementation to track the hooks correctly.

Call Hooks Only from Functional Components or Other Hooks:

Do not call hooks from regular JavaScript functions. Hooks can only be called from React function components or from other custom hooks.

This ensures that the React hook state and lifecycle are properly managed.

Lab Exercise 6

Lab6.js

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

const Lab6 = () => {
  const [weather, setWeather] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  const city = "Hyderabad"; //Replace with any city name
  const apiKey = '61a1d8419f9d307924240958d89e284d'; // Replace with
your actual OpenWeatherMap API key

  useEffect(() => {
    const fetchWeather = async () => {
      try {
        const response = await axios.get(

`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${a
piKey}`
        );
        setWeather(response.data);
        setLoading(false);
      } catch (err) {
        setError(err);
        setLoading(false);
      }
    };

    fetchWeather();
  }, []);

  if (loading) return <p>Loading weather...</p>;
  if (error) return <p>Error fetching weather: {error.message}</p>;

  return (
    <div>
      <h2>Weather in {weather.name}</h2>
      <p>Temperature: {(weather.main.temp - 273.15).toFixed(2)}°C</p>
      <p>Condition: {weather.weather[0].description}</p>
    </div>
  );
};

export default Lab6;
```

This above react component Lab6 performs the following tasks:

- Initializes state variables for storing weather data, loading status, and error information.
- Uses the useEffect hook to fetch weather data from the OpenWeatherMap API when the component mounts.

- Handles the loading state and displays a loading message while data is being fetched.
- Handles errors by displaying an error message if the data fetch fails.
- Displays the fetched weather data, including the city name, temperature in Celsius, and weather condition description.
- The component is then exported for use in other parts of the application.

App.js

```
import './App.css';
import Lab6 from './components/Lab6';

function App() {
  return (
    <div className="App">
      <header className="App-header">
       <Lab6/>
      </header>
    </div>
  );
export default App;
```

Lab Experiment 8

Add a Login component using ReactJs. User should be able to login, using pre-defind logins. Session should be maintained using react hooks.

To create a login component in React that uses predefined logins and maintains a session using hooks, follow these steps:

1. Set Up the Project
2. Create a Context for Authentication
3. Create the Login Component
4. Create a Component to Display After Login
5. Integrate Everything in the App Component

Step 1: Set Up the Project

Ensure you have a React project set up. If not, create one using Create React App:

npx create-react-app login-app

cd login-app

npm start

Step 2: Create a Context for Authentication

We'll create a context to manage authentication state and session.

src/AuthContext.js

Manages the authentication state and provides login and logout functions.

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

// Create a context for authentication
export const AuthContext = createContext();

const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  // Load user from localStorage if available
  useEffect(() => {
    const storedUser = localStorage.getItem('user');
    if (storedUser) {
      setUser(JSON.parse(storedUser));
    }
  }, []);

  const login = (username, password) => {
    // Define pre-defined users
    const predefinedUsers = [
      { username: 'user1', password: 'pass1' },
      { username: 'user2', password: 'pass2' },
    ];

    const user = predefinedUsers.find(
      (u) => u.username === username && u.password === password
    );
```

```
    if (user) {
      setUser(user);
      localStorage.setItem('user', JSON.stringify(user));
      return true;
    }
    return false;
  };

  const logout = () => {
    setUser(null);
    localStorage.removeItem('user');
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};

export default AuthProvider;
```

Step 3: Create the Login Component

Next, create the Login component that uses the AuthContext to handle login.

src/Login.js

Handles user input for username and password, and attempts to log in using the AuthContext.

```
import React, { useState, useContext } from 'react';
import { AuthContext } from './AuthContext';

const Login = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const { login } = useContext(AuthContext);

  const handleSubmit = (e) => {
    e.preventDefault();
    if (login(username, password)) {
      setError('');
      // Redirect or update the UI as needed
    } else {
      setError('Invalid username or password');
    }
  };

  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Username: </label>
          <input
            type="text"
            value={username}
```

```
            onChange={(e) => setUsername(e.target.value)}
          />
        </div>
        <div>
          <label>Password: </label>
          <input
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
          />
        </div>
        <button type="submit">Login</button>
      </form>
      {error && <p style={{ color: 'red' }}>{error}</p>}
    </div>
  );
};

export default Login;
```

Step 4: Create a Component to Display After Login

We'll create a simple component that displays a welcome message and a logout button.

src/Home.js

Displays a welcome message and a logout button for logged-in users.

```
import React, { useContext } from 'react';
import { AuthContext } from './AuthContext';

const Home = () => {
  const { user, logout } = useContext(AuthContext);

  return (
    <div>
      <h2>Welcome, {user.username}!</h2>
      <button onClick={logout}>Logout</button>
    </div>
  );
};

export default Home;
```

Step 5: Integrate Everything in the App Component

Finally, we'll integrate the Login and Home components in the App component, and conditionally render them based on the authentication state.

src/App.js

Conditionally renders the Login or Home component based on the user's authentication state.

```
import React, { useContext } from 'react';
import AuthProvider, { AuthContext } from './AuthContext';
import Login from './Login';
import Home from './Home';
```

```
function App() {
  const { user } = useContext(AuthContext);

  return (
    <div className="App">
      {user ? <Home /> : <Login />}
    </div>
  );
}

const AppWrapper = () => (
  <AuthProvider>
    <App />
  </AuthProvider>
);

export default AppWrapper;
```

By following these steps, you can create a simple login system in React with session management using hooks and context.