W9 - React

▼ Starting a react project locally

There are various ways to bootstrap a react project locally. Vite is the most widely used one today.

Vite

Ref - https://vite.dev/guide/

Vite (French word for "quick", pronounced /vit/, like "veet") is a build tool that aims to provide a faster and leaner development experience for modern web projects. It consists of two major parts:

A dev server that provides **rich feature enhancements** over **native ES modules**, for example extremely fast **Hot Module Replacement (HMR)**.

A build command that bundles your code with **Rollup**, pre-configured to output highly optimized static assets for production.

Intializing a react project

npm create vite@latest

▼ Components

In React, components are the building blocks of the user interface. They allow you to split the UI into independent, reusable pieces that can be thought of as custom, self-contained HTML elements.

▼ useState

`useState` is a Hook that lets you add state to functional components. It returns an array with the current state and a function to update it.

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

const Counter = () => {
```

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

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

Notification count code

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

function App() {
  return (
    <div style={{background: "#dfe6e9", height: "100vh" }}>
      <ToggleMessage />
      <ToggleMessage />
      <ToggleMessage />
    </div>
  )
}

// the component isnt re-rendering
// because we havent used a state variable

const ToggleMessage = () => {
  let [notificationCount, setNotificationCount] = useState
(0);

  console.log("re-render");
  function increment() {
```

```
    setNotificationCount(notificationCount + 1);
  }

  return (
      <div>
          <button onClick={increment}>
              Increase count
          </button>
          {notificationCount}
      </div>
  );
};
```

Post component

```
const style = { width: 200, backgroundColor: "white", borde
rRadius: 10, borderColor: "gray", borderWidth: 1, padding:
20 }

export function PostComponent({name, subtitle, time, image,
description}) {
  return <div style={style}>
    <div style={{display: "flex"}}>
      <img src={image} style={{
        width: 30,
        height: 30,
        borderRadius: 20
      }} />
      <div style={{fontSize: 10, marginLeft: 10}}>
        <b>
          {name}
        </b>
```

```
        <div>{subtitle}</div>
        {(time !== undefined) ? <div style={{display: 'fle
x'}}>
          <div>{time}</div>
          <img src={"<https://media.istockphoto.com/id/9313
36618/vector/clock-vector-icon-isolated.jpg?s=612x612&w=0&k
=20&c=I8EBJl8i6olqcrhAtKko74ydFEVbfCQ6s5Pbsx6vfas=>"} style
={{width: 12, height: 12}} />
        </div> : null}
      </div>
    </div>
    <div style={{fontSize: 12}}>
     {description}
    </div>
 </div>
}
```

- Solution

```
import { useState } from "react";
import { PostComponent } from "./Post";

function App() {
  const [posts, setPosts] = useState([]);

  const postComponents = posts.map(post => <PostComponen
t
    name={post.name}
    subtitle={post.subtitle}
    time={post.title}
    image={post.image}
    description={post.description}
  />)

  function addPost() {
```

```
    setPosts([...posts, {
      name: "harkirat",
      subtitle: "10000 followers",
      time: "2m ago",
      image: "<https://appx-wsb-gcp-mcdn.akamai.net.in/subject/2023-01-17-0.17044360120951185.jpg>",
      description: "What to know how to win big? Check out how these folks won $6000 in bounties."
    }])
  }

  return (
    <div style={{background: "#dfe6e9", height: "100vh", }}>
      <button onClick={addPost}>Add post</button>
      <div style={{display: "flex", justifyContent: "center" }}>
        <div>
          {postComponents}
        </div>
      </div>
    </div>
  )
}

export default App
```

▼ UseEffect

Before we understand `useEffect` , let's understand what are `Side effects`.

Side effects

Side effects are operations that interact with the outside world or have effects beyond the component's rendering. Examples include:

- **Fetching data** from an API.
- **Modifying the DOM** manually.
- **Subscribing to events** (like WebSocket connections, timers, or browser events).
- **Starting a clock**

These are called side effects because they don't just compute output based on the input—they affect things outside the component itself.

Problem in running side effects in React components

If you try to introduce side effects directly in the rendering logic of a component (in the return statement or before it), React would run that code every time the component renders. This can lead to:

- **Unnecessary or duplicated effects** (like multiple API calls).
- **Inconsistent behavior** (side effects might happen before rendering finishes).
- **Performance issues** (side effects could block rendering or cause excessive re-rendering).

How `useEffect` Manages Side Effects:

The `useEffect` hook lets you perform side effects in functional components in a safe, predictable way:

```
useEffect(() => {
  // Code here is the "effect" — this is where side effects
happen
  fetchData();

  // Optionally, return a cleanup function that runs when t
he component unmounts.
  return () => {
    // Cleanup code, e.g., unsubscribing from an event or c
```

```
learing timers.
  };
}, [/* dependencies */]);
```

- **The first argument** to `useEffect` is the effect function, where you put the code that performs the side effect.
- **The second argument** is the dependencies array, which controls when the effect runs. This array tells React to re-run the effect only when certain values (props or state) change. If you pass an empty array `[]`, the effect will only run **once** after the initial render.
- **Optional Cleanup**: If your side effect needs cleanup (e.g., unsubscribing from a WebSocket, clearing intervals), `useEffect` allows you to return a function that React will call when the component unmounts or before re-running the effect.

To recap

`useEffect` is a Hook that lets you perform side effects in functional components. It can be used for data fetching, subscriptions, or manually changing the DOM.

Linkedin like topbar

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

function App() {
  const [currentTab, setCurrentTab] = useState(1);
  const [tabData, setTabData] = useState({});
  const [loading, setLoading] = useState(true);

  useEffect(function() {
    setLoading(true);
    fetch("<https://jsonplaceholder.typicode.com/todos/>" +
currentTab)
      .then(async res => {
```

```
        const json = await res.json();
        setTabData(json);
        setLoading(false);
      });

  }, [])

  return <div>
    <button onClick={function() {
      setCurrentTab(1)
    }} style={{color: currentTab == 1 ? "red" : "black"}}>T
odo #1</button>
    <button onClick={function() {
      setCurrentTab(2)
    }} style={{color: currentTab == 2 ? "red" : "black"}}>T
odo #2</button>
    <button onClick={function() {
      setCurrentTab(3)
    }} style={{color: currentTab == 3 ? "red" : "black"}}>T
odo #3</button>
    <button onClick={function() {
      setCurrentTab(4)
    }} style={{color: currentTab == 4 ? "red" : "black"}}>T
odo #4</button>
<br />
    {loading ? "Loading..." : tabData.title}
  </div>
}

export default App
```

Create a Countdown

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

```
const Timer = () => {
    const [seconds, setSeconds] = useState(0);

    useEffect(() => {
        const interval = setInterval(() => {
            setSeconds(prev => prev + 1);
        }, 1000);

        return () => clearInterval(interval); // Cleanup on
unmount
    }, []);

    return <div>{seconds} seconds elapsed</div>;
};
```

Fetching data

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

const UserList = () => {
    const [users, setUsers] = useState([]);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
        const fetchData = async () => {
            try {
                const response = await fetch('<https://json
placeholder.typicode.com/users>');
                const data = await response.json();
                setUsers(data);
            } catch (error) {
                console.error('Error fetching data:', erro
r);
            } finally {
```

```
                setLoading(false);
            }
        };

        fetchData();
    }, []); // Empty dependency array means this runs once
when the component mounts.

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

    return (
        <ul>
            {users.map(user => (
                <li key={user.id}>{user.name}</li>
            ))}
        </ul>
    );
};

export default UserList;
```

▼ props

Props are the way to pass data from one component to another in React.

```
import React from 'react';

const Greeting = ({ name }) => {
  return <h1>Hello, {name}!</h1>;
};

const App = () => {
  return <Greeting name="Alice" />;
};
```

▼ Conditional Rendering

You can render different components or elements based on certain conditions.

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

const ToggleMessage = () => {
    const [isVisible, setIsVisible] = useState(false);

    return (
        <div>
            <button onClick={() => setIsVisible(!isVisibl
e)}>
                Toggle Message
            </button>
            {isVisible && <p>This message is conditionally
rendered!</p>}
        </div>
    );
};
```

▼ Children

Card component

```
import React from 'react';

const Card = ({ children }) => {
    return (
        <div style={{
            border: '1px solid #ccc',
            borderRadius: '5px',
            padding: '20px',
```

```
            margin: '10px',
            boxShadow: '2px 2px 5px rgba(0, 0, 0, 0.1)',
        }}>
            {children}
        </div>
    );
};

const App = () => {
    return (
        <div>
            <Card>
                <h2>Card Title</h2>
                <p>This is some content inside the card.</p
>
            </Card>
            <Card>
                <h2>Another Card</h2>
                <p>This card has different content!</p>
            </Card>
        </div>
    );
};

export default App;
```

Modals

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

const Modal = ({ isOpen, onClose, children }) => {
    if (!isOpen) return null;

    return (
```

```
        <div style={{
            position: 'fixed',
            top: 0,
            left: 0,
            right: 0,
            bottom: 0,
            backgroundColor: 'rgba(0, 0, 0, 0.5)',
            display: 'flex',
            justifyContent: 'center',
            alignItems: 'center',
        }}>
            <div style={{
                background: 'white',
                padding: '20px',
                borderRadius: '5px',
            }}>
                <button onClick={onClose}>Close</button>
                {children}
            </div>
        </div>
    );
};

const App = () => {
    const [isModalOpen, setModalOpen] = useState(false);

    return (
        <div>
            <button onClick={() => setModalOpen(true)}>Open
Modal</button>
            <Modal isOpen={isModalOpen} onClose={() => setM
odalOpen(false)}>
                <h2>Modal Title</h2>
                <p>This is some content inside the modal.</
p>
            </Modal>
```

```
        </div>
    );
};

export default App;
```

Collapsible Section

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

const Collapsible = ({ title, children }) => {
    const [isOpen, setIsOpen] = useState(false);

    return (
        <div>
            <button onClick={() => setIsOpen(!isOpen)}>
                {title} {isOpen ? '-' : '+'}
            </button>
            {isOpen && <div>{children}</div>}
        </div>
    );
};

const App = () => {
    return (
        <div>
            <Collapsible title="Section 1">
                <p>This is the content of section 1.</p>
            </Collapsible>
            <Collapsible title="Section 2">
                <p>This is the content of section 2.</p>
            </Collapsible>
        </div>
    );
};
```

```
export default App;
```

▼ Lists and Keys

When rendering lists, each item should have a unique key prop for React to track changes efficiently.

```
import React from 'react';

const ItemList = ({ items }) => {
    return (
        <ul>
            {items.map(item => (
                <li key={item.id}>{item.name}</li>
            ))}
        </ul>
    );
};

const App = () => {
    const items = [
        { id: 1, name: 'Item 1' },
        { id: 2, name: 'Item 2' },
        { id: 3, name: 'Item 3' },
    ];

    return <ItemList items={items} />;
};
```

▼ Inline styling

Inline styling in React allows you to apply CSS styles directly to elements using a JavaScript object

```
function MyComponent() {
  return (
    <div style={{ backgroundColor: 'blue', color: 'white'
}}>
      Hello, World!
    </div>
  );
}
```

▼ Class based vs functional components

Earlier, React code was written using `Class based` components. Slowly functional components were introduced and today they are the ones you'll se everywhere.

Ref - https://github.com/processing/p5.js-web-editor/issues/2358

Class components are classes that extend `React.Component`, while functional components are simpler and can use Hooks.

Class based components

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

class ClassCounter extends Component {
    state = { count: 0 };

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

    render() {
        return (
            <div>
                <p>Count: {this.state.count}</p>
                <button onClick={this.increment}>Increment
</button>
```

```
            </div>
        );
    }
}
```

Functional components

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

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

    function increment() {
        setCount(count => count + 1)
    }

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

▼ Lifecycle events

In React, lifecycle events (or lifecycle methods) refer to the specific points in a component's life where you can execute code in response to changes or actions. These events help you manage tasks such as data fetching, subscriptions, and cleaning up resources.

Class-Based Lifecycle Methods

In class components, lifecycle methods are divided into three main phases:

1. **Mounting**: When the component is being inserted into the DOM.
    - `constructor()`: Called when the component is initialized.
    - `componentDidMount()`: Called immediately after the component is mounted. Ideal for data fetching.
2. **Updating**: When the component is being re-rendered due to changes in props or state.
    - `componentDidUpdate(prevProps, prevState)`: Called after the component has updated. Good for operations based on prop/state changes.
3. **Unmounting**: When the component is being removed from the DOM.
    - `componentWillUnmount()`: Ideal for cleanup tasks, like invalidating timers or canceling network requests.

Code

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

  componentDidMount() {
    console.log('Component mounted');
  }

  componentDidUpdate(prevProps, prevState) {
    console.log('Component updated');
  }

  componentWillUnmount() {
    console.log('Component will unmount');
  }

  render() {
```

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

Functional component lifecycle events

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

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

  useEffect(() => {
    console.log('Component mounted or count updated');

  }, [count]); // Runs on mount and when count changes

    useEffect(() => {
            console.log('Component mounted');
    return () => {
      console.log('Component will unmount');
    };
    }, [])

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

```
</button>
    </div>
  );
}
```

▼ Error boundary

Error boundaries are React components that catch JavaScript errors in their child component tree and display a fallback UI.

Error boundaries only exist in class based components

```
import React from 'react';

class ErrorBoundary extends React.Component {
    constructor(props) {
        super(props);
        this.state = { hasError: false };
    }

    static getDerivedStateFromError(error) {
        return { hasError: true };
    }

    componentDidCatch(error, info) {
        console.error("Error caught:", error, info);
    }

    render() {
        if (this.state.hasError) {
            return <h1>Something went wrong.</h1>;
        }

        return this.props.children;
    }
}
```

```
const BuggyComponent = () => {
    throw new Error("I crashed!");
};

const App = () => {
    return (
        <ErrorBoundary>
            <BuggyComponent />
        </ErrorBoundary>
    );
};
```

▼ Fragment

In React, a component can return a single parent element, but it can contain multiple children within that single parent

Wrong code

```
const MyComponent = () => {
    return (
        <h1>Hello</h1>
        <p>World</p> // This line will cause an error
    );
};
```

Right code

```
const MyComponent = () => {
    return (
        <>
            <h1>Hello</h1>
            <p>World</p>
        </>
```

```
    );
};
```

▼ Single page applications, routing

Single Page Applications (SPAs) are web applications that load a single HTML page and dynamically update that page as the user interacts with the app. This approach allows for a smoother user experience compared to traditional multi-page applications (MPAs), where each interaction often requires a full page reload.

Library to use for routing - https://reactrouter.com/en/main

- Install react-router-dom

```
npm i react-router-dom
```

- Update App.jsx with the respective routes

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-
dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";

export default function App() {
  return (
    <BrowserRouter>
        <Routes>
        <Route index element={<Home />} />
        <Route path="blogs" element={<Blogs />} />
        <Route path="contact" element={<Contact />} />
        <Route path="*" element={<NoPage />} />
```

```
      </Routes>
    </BrowserRouter>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

Code from class

```
import './App.css'
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

function App() {

  return <div>
    <BrowserRouter>
      <Link to="/">Allen</Link>
      |
      <Link to="/neet/online-coaching-class-11">Class 11</Link>
      |
      <Link to="/neet/online-coaching-class-12">Class 12</Link>
      <Routes>
        <Route path="/neet/online-coaching-class-11" element={<Class11Program />} />
        <Route path="/neet/online-coaching-class-12" element={<Class12Program />} />
        <Route path="/" element={<Landing />} />
      </Routes>
    </BrowserRouter>
  </div>
```

```
}

function Landing() {
  return <div>
    Welcome to allen
  </div>
}

function Class11Program() {
  return <div>
      NEET programs for Class 11th
  </div>
}

function Class12Program() {
  return <div>
      NEET programs for Class 12th
  </div>
}

export default App
```

▼ Layouts

Layouts let you `wrap` every route inside a certain component (think headers and footers)

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-
dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";
```

```
export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="blogs" element={<Blogs />} />
          <Route path="contact" element={<Contact />} />
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

▼ useRef

What is `useRef`?

In React, `useRef` is a hook that provides a way to create a **reference** to a value or a DOM element that persists across renders but **does not trigger a re-render** when the value changes.

Key Characteristics of `useRef`:

1. **Persistent Across Renders**: The value stored in `useRef` persists between component re-renders. This means the value of a `ref` does not get reset when the component re-renders, unlike regular variables.
2. **No Re-Renders on Change**: Changing the value of a `ref` (`ref.current`) does **not** cause a component to re-render. This is different from state (`useState`), which triggers a re-render when updated.

1. Focussing on an input box

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

function FocusInput() {
  // Step 1: Create a ref to store the input element
  const inputRef = useRef(null);

  // Step 2: Define the function to focus the input
  const handleFocus = () => {
    // Access the DOM node and call the focus method
    inputRef.current.focus();
  };

  return (
    <div>
      {/* Step 3: Attach the ref to the input element */}
      <input ref={inputRef} type="text" placeholder="Click
the button to focus me" />
      <button onClick={handleFocus}>Focus the input</button
>
    </div>
  );
}

export default FocusInput;
```

2. Scroll to bottom

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

function Chat() {
  const [messages, setMessages] = useState(["Hello!", "How
are you?"]);
  const chatBoxRef = useRef(null);
```

```
  // Function to simulate adding new messages
  const addMessage = () => {
    setMessages((prevMessages) => [...prevMessages, "New me
ssage!"]);
  };

  // Scroll to the bottom whenever a new message is added
  useEffect(() => {
    chatBoxRef.current.scrollTop = chatBoxRef.current.scrol
lHeight;
  }, [messages]);

  return (
    <div>
      <div
        ref={chatBoxRef}
        style={{ height: "200px", overflowY: "scroll", bord
er: "1px solid black" }}
      >
        {messages.map((msg, index) => (
          <div key={index}>{msg}</div>
        ))}
      </div>
      <button onClick={addMessage}>Add Message</button>
    </div>
  );
}

export default Chat;
```

3. Clock with start and stop functionality

Notice the re-renders in both of the following cases -

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

function Stopwatch() {
  const [time, setTime] = useState(0);
  const [intervalId, setIntervalId] = useState(null); // Us
e state to store the interval ID

  const startTimer = () => {
    if (intervalId !== null) return; // Already running, do
nothing

    const newIntervalId = setInterval(() => {
      setTime((prevTime) => prevTime + 1);
    }, 1000);

    // Store the interval ID in state (triggers re-render)
    setIntervalId(newIntervalId);
  };

  const stopTimer = () => {
    clearInterval(intervalId);

    // Clear the interval ID in state (triggers re-render)
    setIntervalId(null);
  };

  return (
    <div>
      <h1>Timer: {time}</h1>
      <button onClick={startTimer}>Start</button>
      <button onClick={stopTimer}>Stop</button>
    </div>
  );
}
```

```
export default Stopwatch;
```

```
// better code
import React, { useState, useRef } from 'react';

function Stopwatch() {
  const [time, setTime] = useState(0);
  const intervalRef = useRef(null);

  const startTimer = () => {
    if (intervalRef.current !== null) return; // Already ru
nning, do nothing

    intervalRef.current = setInterval(() => {
      setTime((prevTime) => prevTime + 1);
    }, 1000);
  };

  const stopTimer = () => {
    clearInterval(intervalRef.current);
    intervalRef.current = null;
  };

  return (
    <div>
      <h1>Timer: {time}</h1>
      <button onClick={startTimer}>Start</button>
      <button onClick={stopTimer}>Stop</button>
    </div>
  );
}
```

▼ Custom Hook

Custom hooks in React are a powerful feature that allows you to encapsulate and reuse stateful logic across different components. They are essentially JavaScript functions that can use React hooks internally. By creating custom hooks, you can abstract away complex logic, making your components cleaner and more manageable.

Why Use Custom Hooks?

1. **Reusability**: If you find yourself using the same logic in multiple components, a custom hook can help you avoid code duplication.
2. **Separation of Concerns**: They allow you to separate business logic from UI logic, making your components more focused and easier to understand.

Common custom hooks

- useFetch (https://swr.vercel.app/)

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

export const useFetch = (url: string) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<Error | null>(nul
l);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true);
      try {
        const response = await fetch(url);
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err as Error);
      } finally {
        setLoading(false);
```

```
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
};
```

- usePrevValue

```
```

- useFetch with re-fetching

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

export const useFetch = (url: string, interval: number |
null = null) => {
  const [data, setData] = useState<any>(null);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<Error | null>(nul
l);

  const fetchData = async () => {
    setLoading(true);
    try {
      const response = await fetch(url);
      const result = await response.json();
      setData(result);
    } catch (err) {
      setError(err as Error);
    } finally {
      setLoading(false);
    }
  };
```

```
  useEffect(() => {
    fetchData(); // Initial fetch

    if (interval !== null) {
      const fetchInterval = setInterval(() => {
        fetchData();
      }, interval);

      return () => clearInterval(fetchInterval); // Clea
r interval on cleanup
    }
  }, [url, interval]);

  return { data, loading, error };
};
```

- usePrev

```
import { useRef, useState, useEffect } from 'react'
import './App.css'

export const usePrev = (value) => {
  const ref = useRef();

  // Update the ref with the current value after each re
nder
  useEffect(() => {
    ref.current = value;
  }, [value]);

  // Return the previous value (current value of ref bef
ore it is updated)
  return ref.current;
```

```
};

function App() {
  const [count, setCount] = useState(0);
  const prevCount = usePrev(count); // Track the previou
s count value

  return (
    <div style={{ textAlign: 'center', marginTop: '50px'
}}>
      <h1>Counter with usePrev Hook</h1>
      <p>Current Count: {count}</p>
      <p>Previous Count: {prevCount}</p>
      <button onClick={() => setCount(count + 1)}>Increm
ent</button>
      <button onClick={() => setCount(count - 1)} style=
{{ marginLeft: '10px' }}>Decrement</button>
    </div>
  );
}

export default App
```

- useIsOnline

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

const useIsOnline = () => {
  const [isOnline, setIsOnline] = useState(navigator.onL
ine);

  useEffect(() => {
    const updateOnlineStatus = () => setIsOnline(navigat
or.onLine);
```

```
    window.addEventListener('online', updateOnlineStatu
s);
    window.addEventListener('offline', updateOnlineStatu
s);

    // Clean up the event listeners on component unmount
    return () => {
      window.removeEventListener('online', updateOnlineS
tatus);
      window.removeEventListener('offline', updateOnline
Status);
    };
  }, []);

  return isOnline;
};

export default useIsOnline;
```

- useDebounce (https://gist.github.com/hkirat/439a0be477e3c31b08c1f7e0f4582674)

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

const useDebounce = (value, delay) => {
    const [debouncedValue, setDebouncedValue] = useState
(value);

    useEffect(() => {
        const handler = setTimeout(() => {
            setDebouncedValue(value);
        }, delay);
```

```
        return () => {
            clearTimeout(handler);
        };
    }, [value, delay]);

    return debouncedValue;
};

export default useDebounce;
```

- If we want to debounce function (from ak)

```
const debounce = (func, delay) => {
  let timeout;
  return (...args) => {
    clearTimeout(timeout);  // Clears the previous ti
mer
    timeout = setTimeout(() => func(...args), delay);
// Starts a new timer
  };
};
```

Assignment

Read about swr and react-query

- Is `usePrev` hook that we have made indeed correct? Try out this example https://giacomocerquone.com/blog/life-death-useprevious-hook/

```
import { useRef, useState, useEffect } from 'react'
import './App.css'

export const usePrev = (value) => {
  const ref = useRef();
```

```
  // Update the ref with the current value after each re
nder
  useEffect(() => {
    ref.current = value;
  }, [value]);

  // Return the previous value (current value of ref bef
ore it is updated)
  return ref.current;
};

function App() {
  const [count, setCount] = useState(0);
  const prevCount = usePrev(count); // Track the previou
s count value
  const [x, setX] = useState(0);

  useEffect(() => {
    setInterval(() => {
      setX(x => x + 1);
    }, 1000)

  }, []);
  return (
    <div style={{ textAlign: 'center', marginTop: '50px'
}}>
      <h1>Counter with usePrev Hook</h1>
      <p>Current Count: {count}</p>
      <p>Previous Count: {prevCount}</p>
      {x}
      <button onClick={() => setCount(count + 1)}>Increm
ent</button>
      <button onClick={() => setCount(count - 1)} style=
{{ marginLeft: '10px' }}>Decrement</button>
    </div>
  );
```

```
}

export default App
```

- Better `usePrev`

```

```

Explore https://usehooks.com/

▼ Rolling up the state, unoptimal re-renders

As your application grows, you might find that multiple components need access to the same state. Instead of duplicating state in each component, you can lift the state up to the LCA, allowing the common ancestor to manage it.

Example

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

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

  return (
    <>
      <Incrase count={count} setCount={setCount} />
      <Decrease count={count} setCount={setCount} />
      <Value count={count} setCount={setCount} />
    </>
  );
}

function Decrease({ count, setCount }) {
  return <button onClick={() => setCount(count - 1)}>Decrea
se</button>;
}
```

```
function Increase({ count, setCount }) {
  return <button onClick={() => setCount(count + 1)}>Increase</button>;
}

function Value({ count }) {
  return <p>Count: {count}</p>;
}

// App Component
const App = () => {
  return <div>
    <Parent />
  </div>
};

export default App;
```

Lightbulb example

```
import { useState } from 'react'
import './App.css'

function App() {
  return <div>
    <LightBulb />
  </div>
}

function LightBulb() {
  const [bulbOn, setBulbOn] = useState(true)

  return <div>
    <BulbState bulbOn={bulbOn} />
```

```
      <ToggleBulbState bulbOn={bulbOn} setBulbOn={setBulbOn}
/>
    </div>
}

function BulbState({bulbOn}) {
  return <div>
    {bulbOn ? "Bulb on" : "Bulb off"}
  </div>
}

function ToggleBulbState({bulbOn, setBulbOn}) {

  function toggle() {
    // setBulbOn(currentState => !currentState)
    setBulbOn(!bulbOn)

  }

  return <div>
    <button onClick={toggle}>Toggle the bulb</button>
  </div>
}

export default App
```

▼ Prop Drilling

Prop drilling occurs when you need to pass data from a higher-level component down to a lower-level component that is several layers deep in the component tree. This often leads to the following issues:

- **Complexity:** You may have to pass props through many intermediate components that don't use the props themselves, just to get them to the component that needs them.

- **Maintenance:** It can make the code harder to maintain, as changes in the props structure require updates in multiple components.

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

// LightBulb Component
const LightBulb = ({ isOn }) => {
  return <div>The light is {isOn ? 'ON' : 'OFF'}</div>;
};

// LightSwitch Component
const LightSwitch = ({ toggleLight }) => {
  return <button onClick={toggleLight}>Toggle Light</button>;
};

// App Component
const App = () => {
  const [isLightOn, setIsLightOn] = useState(false);

  const toggleLight = () => {
    setIsLightOn((prev) => !prev);
  };

  return (
    <div>
      <LightBulb isOn={isLightOn} />
      <LightSwitch toggleLight={toggleLight} />
    </div>
  );
};

export default App;
```

▼ Context API

The Context API is a powerful feature in React that enables you to manage state across your application more effectively, especially when dealing with deeply nested components.

The Context API provides a way to share values (state, functions, etc.) between components without having to pass props down manually at every level.

Jargon

Context: This is created using React.createContext(). It serves as a container for the data you want to share.

Provider: This component wraps part of your application and provides the context value to all its descendants. Any component that is a child of this Provider can access the context.

Consumer: This component subscribes to context changes. It allows you to access the context value (using useContext hook)

Old code

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

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

  return (
    <>
      <Incrase count={count} setCount={setCount} />
      <Decrease count={count} setCount={setCount} />
      <Value count={count} setCount={setCount} />
    </>
  );
}

function Decrease({ count, setCount }) {
  return <button onClick={() => setCount(count - 1)}>Decrease
}
```

```
function Increase({ count, setCount }) {
  return <button onClick={() => setCount(count + 1)}>Increase
}

function Value({ count }) {
  return <p>Count: {count}</p>;
}

// App Component
const App = () => {
  return <div>
    <Parent />
  </div>
};

export default App;
```

New code

- Create a new `CountContext`

```
const CountContext = createContext();
```

- Create a CountContextProvider

```
function CountContextProvider({ children }) {
  const [count, setCount] = useState(0);

  return <CountContext.Provider value={{count, setCount}}>
    {children}
  </CountContext.Provider>
}
```

- Wrap your app inside the `CountContextProvider`

```
function Parent() {
  return (
    <CountContextProvider>
      <Incrase />
      <Decrease />
      <Value />
    </CountContextProvider>
  );
}
```

- Consume the `context` in individual components

```
function Decrease() {
  const {count, setCount} = useContext(CountContext);
  return <button onClick={() => setCount(count - 1)}>Decrea
se</button>;
}

function Incrase() {
  const {count, setCount} = useContext(CountContext);
  return <button onClick={() => setCount(count + 1)}>Increa
se</button>;
}

function Value() {
  const {count} = useContext(CountContext);
  return <p>Count: {count}</p>;
}
```

- Export the `App` component which renders `Parent`

```
const App = () => {
  return <div>
    <Parent />
  </div>
};
export default App;
```

▼ Testing the context api

Lets test our application, and observe the renders.

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

const CountContext = createContext();

function CountContextProvider({ children }) {
  const [count, setCount] = useState(0);

  return <CountContext.Provider value={{count, setCount}}>
    {children}
  </CountContext.Provider>
}

function Parent() {
  return (
    <CountContextProvider>
      <Incrase />
      <Decrease />
      <Value />
    </CountContextProvider>
  );
}
```

```
function Decrease() {
  const {count, setCount} = useContext(CountContext);
  return <button onClick={() => setCount(count - 1)}>Decrea
se</button>;
}

function Incrase() {
  const {count, setCount} = useContext(CountContext);
  return <button onClick={() => setCount(count + 1)}>Increa
se</button>;
}

function Value() {
  const {count} = useContext(CountContext);
  return <p>Count: {count}</p>;
}

// App Component
const App = () => {
  return <div>
    <Parent />
  </div>
};

export default App;
```

Optimising the app

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

const CountContext = createContext();

function CountContextProvider({ children }) {
```

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

  return <CountContext.Provider value={{count, setCount}}>
    {children}
  </CountContext.Provider>
}

function Parent() {
  return (
    <CountContextProvider>
      <Incrase />
      <Decrease />
      <Value />
    </CountContextProvider>
  );
}

function Decrease() {
  const {count, setCount} = useContext(CountContext);
  return <button onClick={() => setCount(count - 1)}>Decrea
se</button>;
}

function Incrase() {
  const {count, setCount} = useContext(CountContext);
  return <button onClick={() => setCount(count + 1)}>Increa
se</button>;
}

function Value() {
  const {count} = useContext(CountContext);
  return <p>Count: {count}</p>;
}

// App Component
const App = () => {
```

```
  return <div>
    <Parent />
  </div>
};

export default App;
```

What context does, and what it doesn't

The Context API primarily addresses the issue of prop drilling by allowing you to share state across your component tree without needing to pass props through every level.

It doesn't optimise renders in your application, which becomes important if/when your application becomes bigger

▼ Introducing recoil

To minimise the number of re-renders, and ensure that only components that are `subscribed` to a value `render`, state management was introduced.

There are many libraries that let you do state management -

1. mobx
2. recoil
3. redux

Here is a simple example with `recoil`

- npm install recoil

```
import React, { createContext, useContext, useState } from
'react';
import { RecoilRoot, atom, useRecoilValue, useSetRecoilStat
e } from 'recoil';

const count = atom({
  key: 'countState', // unique ID (with respect to other at
oms/selectors)
```

```
  default: 0, // default value (aka initial value)
});

function Parent() {
  return (
    <RecoilRoot>
      <Incrase />
      <Decrease />
      <Value />
    </RecoilRoot>
  );
}

function Decrease() {
  const setCount = useSetRecoilState(count);
  return <button onClick={() => setCount(count => count -
1)}>Decrease</button>;
}

function Incrase() {
  const setCount = useSetRecoilState(count);
  return <button onClick={() => setCount(count => count +
1)}>Increase</button>;
}

function Value() {
  const countValue = useRecoilValue(count);
  return <p>Count: {countValue}</p>;
}

// App Component
const App = () => {
  return <div>
    <Parent />
  </div>
};
```

```
export default App;
```