major features of react:-

React is a popular JavaScript library for building user interfaces, particularly single-page applications where data changes over time.

Declarative UI

React allows you to describe what the UI should look like in a declarative way. This makes the code more predictable and easier to debug.

```
const App = () => {
  return <h1>Hello, World!</h1>;
};
```

Component-Based Architecture

React encourages building UI components. Each component encapsulates its own structure, logic, and style, which can be reused and nested within other components.

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

const App = () => {
  return (
    <div>
      <Greeting />
      <p>Welcome to React.</p>
    </div>
  );
};
```

JSX Syntax

JSX is a syntax extension for JavaScript that looks similar to XML or HTML. It allows you to write HTML directly within JavaScript, making it easier to create React components.

```
const element = <h1>Hello, World!</h1>;
```

Virtual DOM

React uses a virtual DOM to optimize updates. When the state of an object changes, React updates the virtual DOM first, compares it with a snapshot from before the update, and only applies the differences to the actual DOM.

State and Lifecycle

React components can maintain internal state and have lifecycle methods to manage component creation, updating, and destruction.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  componentDidMount() {
    this.timerID = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, World!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

One-Way Data Binding

One-way data binding in React is a key concept that helps maintain a unidirectional data flow within an application.

It means that data flows in a single direction, from the parent component to the child component. This approach contrasts with two-way data binding, where changes in the UI can directly update the model and vice versa.

Data Flow: Data flows from the parent component down to the child components via props.

Unidirectional: The data flow is unidirectional, meaning the parent component is the source of truth.

Immutability: The child component cannot modify the data directly. Instead, it can trigger an event that requests the parent component to update the state, which then re-renders the child component with the updated data.

Hooks

React Hooks are functions that allow you to use state and other React features in functional components.

useState

The useState hook allows you to add state to functional components.

```
const [state, setState] = useState(initialState);
import React, { useState } from 'react';
const Counter = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
};
```

<u>useEffect:-</u>

The useEffect hook lets you perform side effects in functional components. It serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

```
useEffect(() => {
  // Side effect code
  return () => {
    // Cleanup code
  };
}, [dependencies]);
import React, { useState, useEffect } from 'react';

const Timer = () => {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const timer = setInterval(() => {
      setCount(prevCount => prevCount + 1);
    }, 1000);
    return () => clearInterval(timer);
  }, []);
  return <div>{count}</div>;
};
```

<u>useContext</u>

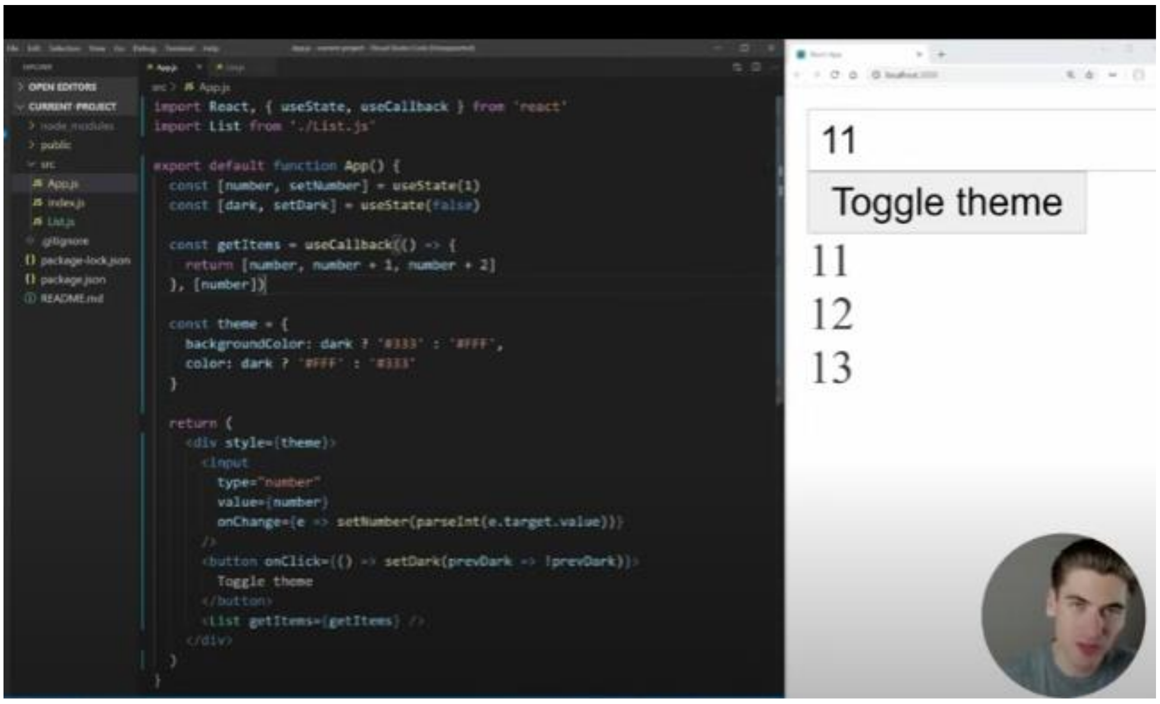
The useContext hook allows you to use the context in functional components. It takes a context object (the value returned from React.createContext) and returns the current context value.

```
const value = useContext(MyContext);
import React, { createContext, useContext } from 'react';


const ThemeContext = createContext('light');


const ThemedComponent = () => {
  const theme = useContext(ThemeContext);
  return <div>Current theme: {theme}</div>;
};
const App = () => (
  <ThemeContext.Provider value="dark">
    <ThemedComponent />
  </ThemeContext.Provider>
);
```

useCallback

The useCallback hook returns a memoized callback function. It's useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```



```
import React, { useState, useCallback } from 'react'
import List from './List.js'

export default function App() {
  const [number, setNumber] = useState(1)
  const [dark, setDark] = useState(false)

  const getItems = useCallback(() => {
    return [number, number + 1, number + 2]
  }, [number])

  const theme = {
    backgroundColor: dark ? '#333' : '#FFF',
    color: dark ? '#FFF' : '#333'
  }

  return (
    <div style={theme}>
      <input
        type="number"
        value={number}
        onChange={e => setNumber(parseInt(e.target.value))}
      />
      <button onClick={() => setDark(prevDark => !prevDark)}>
        Toggle theme
      </button>
      <List getItems={getItems} />
    </div>
  )
}
```

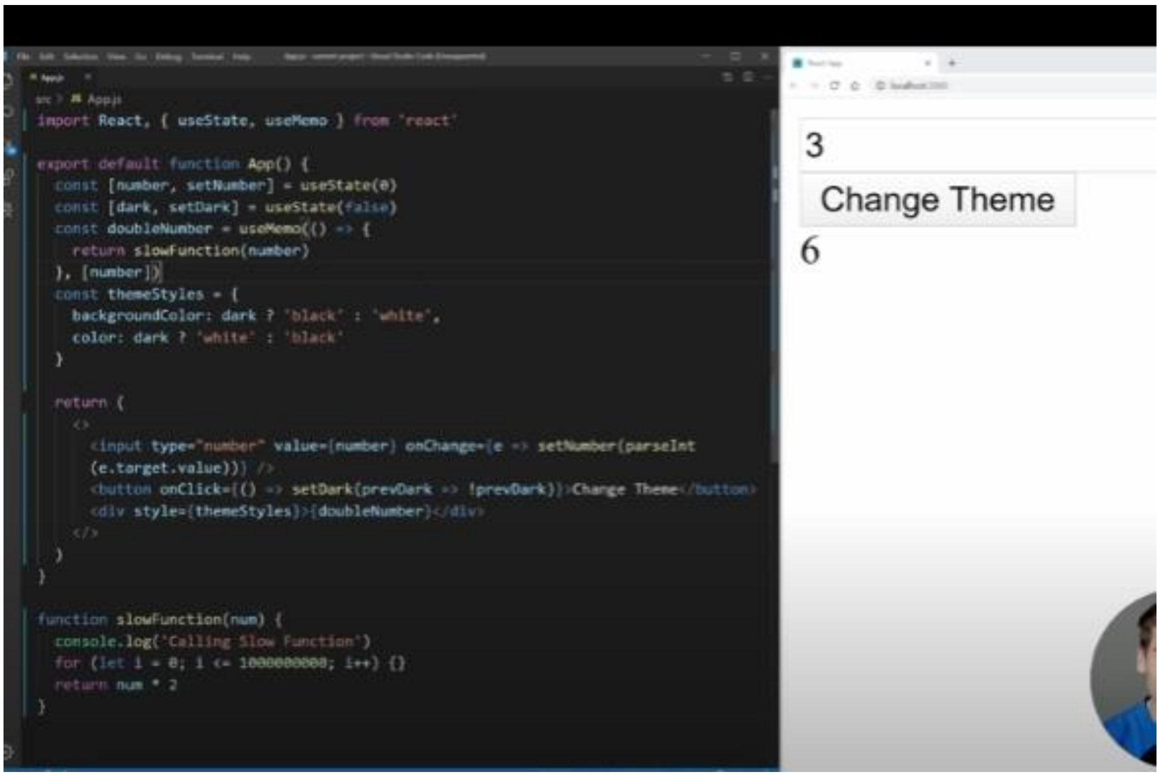
```
11
Toggle theme
11
12
13
```

useMemo

The useMemo hook returns a memoized value. It’s useful for optimizing performance by memoizing expensive calculations.

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

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

const App = () => {
  const [a, setA] = useState(1);
  const [b, setB] = useState(1);

  const sum = useMemo(() => {
    return a + b;
  }, [a, b]);

  return (
    <div>
      <p>Sum: {sum}</p>
      <button onClick={() => setA(a + 1)}>Increment A</button>
      <button onClick={() => setB(b + 1)}>Increment B</button>
    </div>
  );
};
```

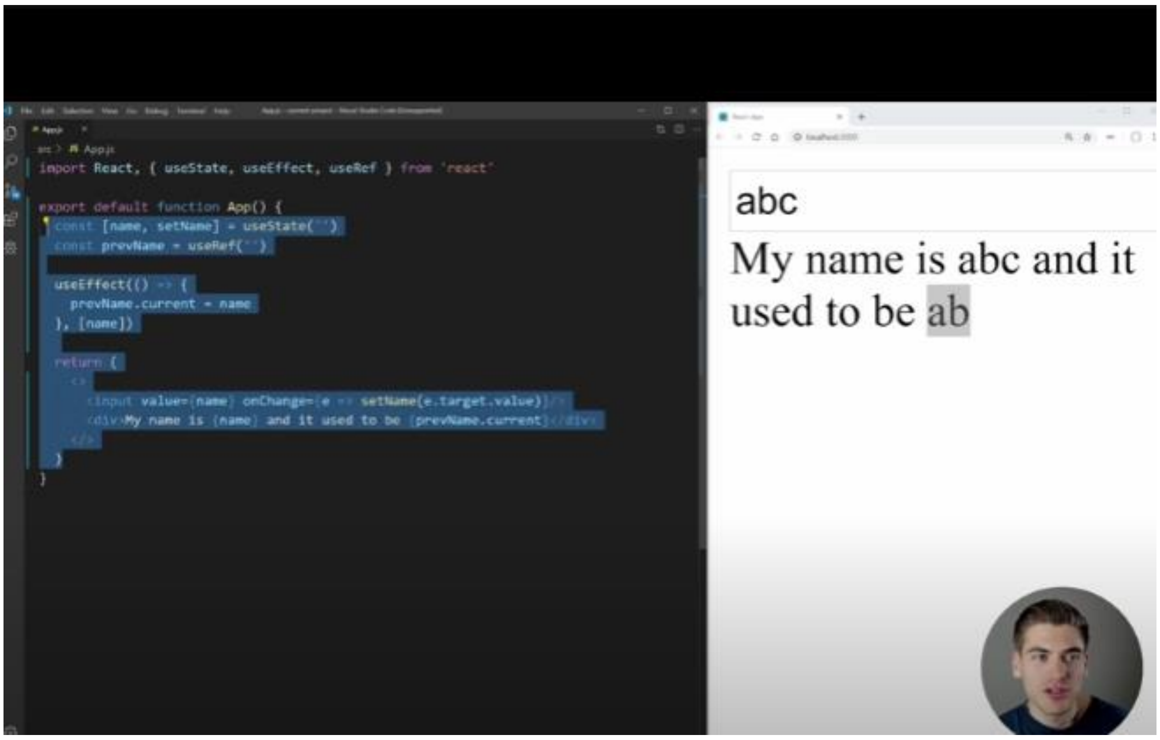


```
import React, { useState, useMemo } from 'react'

export default function App() {
  const [number, setNumber] = useState(0)
  const [dark, setDark] = useState(false)
  const doubleNumber = useMemo(() => {
    return slowFunction(number)
  }, [number])
  const themeStyles = {
    backgroundColor: dark ? 'black' : 'white',
    color: dark ? 'white' : 'black'
  }

  return (
    <>
      <input type="number" value={number} onChange={e => setNumber(parseInt
      (e.target.value))} />
      <button onClick={() => setDark(prevDark => !prevDark)}>Change Theme</button>
      <div style={themeStyles}>{doubleNumber}</div>
    </>
  )
}

function slowFunction(num) {
  console.log('Calling Slow Function')
  for (let i = 0; i <= 1000000000; i++) {}
  return num * 2
}
```

useRef

The useRef hook returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). It can be used to access a DOM element directly.

--it can be used to store previou values of a state also



```
src > App.js
import React, { useState, useEffect, useRef } from 'react'

export default function App() {
  const [name, setName] = useState('')
  const prevName = useRef('')

  useEffect(() => {
    prevName.current = name
  }, [name])

  return (
    <>
      <input value={name} onChange={e => setName(e.target.value)} />
      <div>My name is {name} and it used to be {prevName.current}</div>
    </>
  )
}
```

abc

My name is abc and it used to be ab

```
const refContainer = useRef(initialValue);
import React, { useRef } from 'react';

const FocusInput = () => {
  const inputRef = useRef(null);
  const handleClick = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus the input</button>
    </div>
  );
};
```

useLayoutEffect

The useLayoutEffect hook is similar to useEffect but fires synchronously after all DOM mutations. It can be used to read layout from the DOM and synchronously re-render.

```
useLayoutEffect(() => {
  // Side effect code
  return () => {
    // Cleanup code
  };
}, [dependencies]);
import React, { useLayoutEffect, useRef } from 'react';
const App = () => {
  const divRef = useRef();
  useLayoutEffect(() => {
    console.log(divRef.current.getBoundingClientRect());
  }, []);
  return <div ref={divRef}>Hello, world!</div>;
};
```

Context API

The React Context API is a powerful tool for managing global state in a React application. It allows you to share data across the component tree without having to pass props down manually at every level. This can be especially useful for themes, user authentication, settings, and other data that need to be accessible throughout the application.

Key Concepts of Context API

Context Object: Created with React.createContext(), it comes with a Provider and a Consumer component.

Provider: The Provider component wraps part of the component tree and supplies the context value.

Consumer: The Consumer component subscribes to context changes and allows components to access the context value.

useContext Hook: A more modern and convenient way to access context values in functional components.

High Performance

React optimizes performance through techniques like the virtual DOM and efficient diff algorithms, ensuring that updates to the UI are fast and minimal

Server-Side Rendering (SSR)

React can be rendered on the server using frameworks like Next.js, which improves performance and SEO for web applications.

benefits of using functional components:-

Functional components in React are simple JavaScript functions that accept props as an argument and return React elements. They have gained popularity due to their simplicity and the introduction of hooks, which allow them to manage state and side effects.

Simplicity and Readability

- **Less Boilerplate**: Functional components are simpler and have less boilerplate compared to class components. They are just functions that return JSX, making them easier to read and understand.

Hooks

- **State Management**: With the introduction of hooks (e.g., `useState`, `useEffect`), functional components can now handle state and side effects, which were previously only possible in class components.

state & props:-

State

State is a built-in object that allows React components to maintain and manage local data. It is typically used to handle data that can change over time, such as user inputs, fetched data, or dynamic content.

way to pass data from children to parent:-

Passing data from a child component to a parent component in React is typically done through callback functions.

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

function ParentComponent() {
  const [formData, setFormData] = useState({ name: '', email: '' });

  const handleFormSubmit = (data) => {
    setFormData(data);
  };

  return (
    <div>
      <h1>Form Data</h1>
      <p>Name: {formData.name}</p>
      <p>Email: {formData.email}</p>
      <FormChild onFormSubmit={handleFormSubmit} />
    </div>
  );
}

export default ParentComponent;

import React, { useState } from 'react';

function FormChild(props) {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    props.onFormSubmit({ name, email });
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Name:</label>
        <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
      </div>
      <div>
        <label>Email:</label>
        <input type="email" value={email} onChange={(e) => setEmail(e.target.value)} />
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormChild;
```

pure components:-

A React component is considered pure if it renders the same output for the same state and props. Pure components are useful for performance optimization because React can skip rendering them if their props and state have not changed.

There are two ways to create pure components in React:
Using the React.PureComponent base class:
Class components that extend the React.PureComponent class are treated as pure components. React will automatically compare the props and state of the component before rendering it, and will skip rendering if they have not changed.
Using functional components:
Functional components are always pure components, since they are simply functions that take props and return a React element.

```
import React from 'react';

class MyPureComponent extends React.PureComponent {
  render() {
    const { props } = this;
    return (
      <div>
        <h1>{props.title}</h1>
        <p>{props.content}</p>
      </div>
    );
  }
}

export default MyPureComponent;

import React from 'react';

const MyPureFunctionalComponent = ({ props }) => {
  return (
    <div>
      <h1>{props.title}</h1>
      <p>{props.content}</p>
    </div>
  );
};

export default MyPureFunctionalComponent;
```

lazy loading:-

Lazy loading is a design pattern used to defer the initialization of resources until they are actually needed. In the context of web development, it typically refers to the practice of delaying the loading of non-critical resources (like images, videos, scripts, or components) to improve the initial load time and performance of a web application.

React-router:-

React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.

what is hooks:-

Hooks are a feature introduced in React 16.8 that allow you to use state and other React features without writing a class. They enable function components to have access to React state and lifecycle features, which previously were only available in class components. Hooks simplify the code and make it more readable and maintainable.

Usememo:-

useMemo is a React Hook used for memoization, which is the process of storing the results of expensive function calls and returning the cached result when the same inputs occur again. In React, useMemo memoizes the result of a function so that it is only recomputed when its dependencies change. This can help optimize performance by avoiding unnecessary re-computations.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

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

function ExpensiveComponent({ a, b }) {
  const computeExpensiveValue = (a, b) => {
    console.log('Computing expensive value...');
    return a + b;
  };

  const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

  return <div>Memoized Value: {memoizedValue}</div>;
}

function App() {
  const [a, setA] = useState(1);
  const [b, setB] = useState(2);

  return (
    <div>
      <button onClick={() => setA(a + 1)}>Increment A</button>
      <button onClick={() => setB(b + 1)}>Increment B</button>
      <ExpensiveComponent a={a} b={b} />
    </div>
  );
}

export default App;
```

difference between usecallback and usememo

useCallback: Memoizes a callback function. It's used to prevent unnecessary re-creations of functions, which can help optimize performance when passing callbacks to child components that rely on reference equality to prevent re-renders.

useMemo: Memoizes a value or a computed result. It's used to avoid expensive calculations on every render by caching the result and recalculating it only when dependencies change.

```
const memoizedCallback = useCallback(() => {
  // function logic
}, [dependencies]);

const memoizedValue = useMemo(() => {
  return computeExpensiveValue(a, b);
}, [a, b]);
```

useCallback:

Use when you need to memoize a function to avoid passing a new function instance on every render.
Commonly used when passing callbacks to optimized child components (e.g., components wrapped in React.memo).

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

const Button = React.memo(({ onClick }) => {
  console.log('Button rendered');
  return <button onClick={onClick}>Click me</button>;
});

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

  const increment = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []);

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

export default App;
```
useMemo:

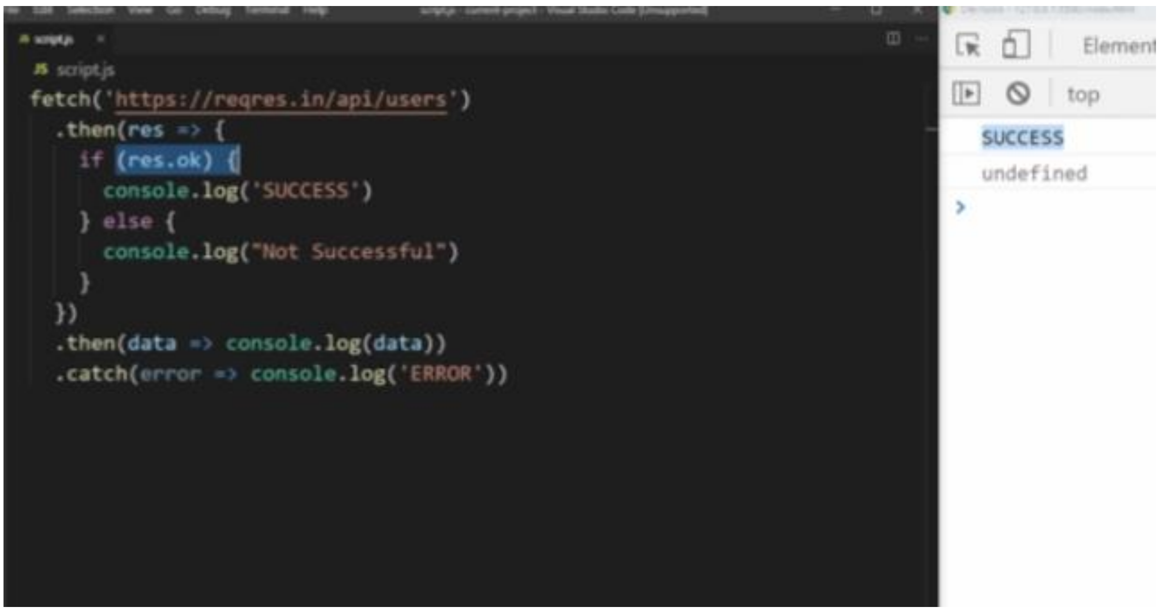
Use when you need to memoize a value that requires an expensive computation to derive.
Helps in performance optimization by avoiding recalculations on every render unless dependencies change.
```
import React, { useState, useMemo } from 'react';

const computeExpensiveValue = (a, b) => {
  console.log('Computing expensive value...');
  // Simulate expensive computation
  let result = 0;
  for (let i = 0; i < 1000000000; i++) {
    result += a + b;
  }
  return result;
};
```

```
const App = () => {
  const [a, setA] = useState(1);
  const [b, setB] = useState(2);

  const expensiveValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

  return (
    <div>
      <p>Expensive value: {expensiveValue}</p>
      <button onClick={() => setA(a + 1)}>Increment A</button>
      <button onClick={() => setB(b + 1)}>Increment B</button>
    </div>
  );
};

export default App;
```

methods to call api in react:-

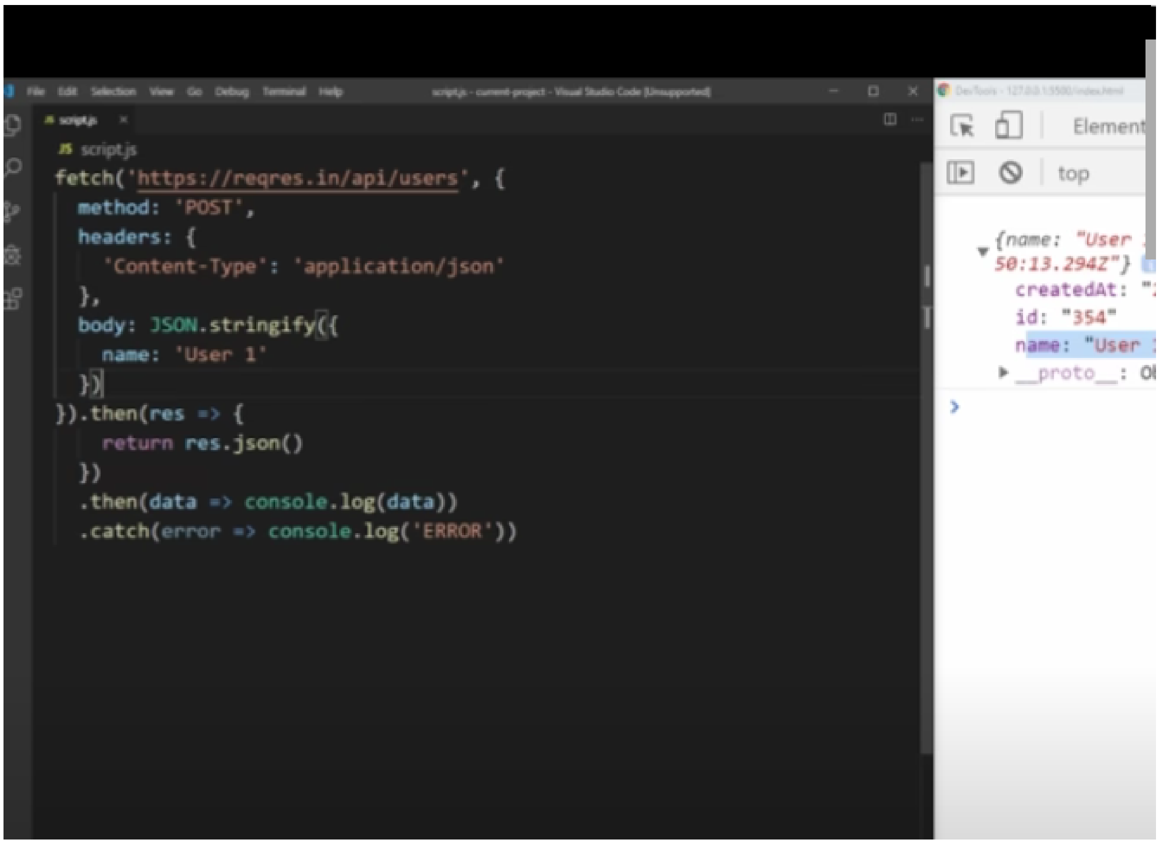
<u>Fetch API</u>

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

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

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data))
      .catch(error => console.error('Error fetching data:', error));
  }, []);

  return (
    <div>
      {/* Render data */}
    </div>
  );
}

export default MyComponent;
```



```
JS script.js
fetch('https://reqres.in/api/users')
  .then(res => {
    if (res.ok) {
      console.log('SUCCESS')
    } else {
      console.log("Not Successful")
    }
  })
  .then(data => console.log(data))
  .catch(error => console.log('ERROR'))
```

```
SUCCESS
undefined
>
```



```
JS script.js
fetch('https://reqres.in/api/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    name: 'User 1'
  })
}).then(res => {
    return res.json()
  })
  .then(data => console.log(data))
  .catch(error => console.log('ERROR'))
```

```
▼{name: "User
 50:13.294Z"}
   createdAt: "
   id: "354"
   name: "User
  ▶__proto__: Ob
>
```

Content-Type

In HTTP, the Content-Type header is used to specify the media type of the resource being sent to the server.

application/json

Description: Used for sending JSON-encoded data.
When to Use: When the server expects JSON data, which is common for APIs.
Why: JSON is lightweight, easy to parse, and widely used for data interchange.

```
fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ key: 'value' })
});
```

application/x-www-form-urlencoded

Description: Used for sending form data in URL-encoded format.
When to Use: When submitting form data, typically from an HTML form.
Why: It’s the standard encoding for form submissions and easily processed by server-side languages.

```
const params = new URLSearchParams();
params.append('key', 'value');

fetch('https://api.example.com/form', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded'
  },
  body: params
});
```

multipart/form-data

Description: Used for sending form data that includes files.
When to Use: When uploading files or sending large amounts of binary data.
Why: Allows for sending files and binary data as part of the form data.

```
const formData = new FormData();
formData.append('file', fileInput.files[0]);
formData.append('key', 'value');

fetch('https://api.example.com/upload', {
  method: 'POST',
  body: formData
});
```

text/plain

Description: Used for sending plain text data.
When to Use: When sending plain text without any special formatting or encoding.
Why: Simple and useful for basic text data without the need for encoding

```
fetch('https://api.example.com/text', {
  method: 'POST',
  headers: {
    'Content-Type': 'text/plain'
  },
  body: 'Just some plain text'
});
```

application/xml

Description: Used for sending XML data.
When to Use: When the server expects XML-formatted data.
Why: XML is used in certain legacy systems and for data that requires a strict hierarchical structure.

```
const xmlData =
`<note><to>User</to><from>API</from><message>Hello</message></note>`;

fetch('https://api.example.com/xml', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/xml'
  },
  body: xmlData
});
```

application/octet-stream

Description: Used for sending arbitrary binary data.
When to Use: When uploading binary files where the type is unknown or generic binary data.
Why: It's a default for binary data that doesn't fit other media types.

```
fetch('https://api.example.com/binary', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/octet-stream'
  },
  body: binaryData // Assume binaryData is an ArrayBuffer or Blob
});
```

application/pdf

Description: Used for sending PDF files.
When to Use: When uploading or sending PDF documents.
Why: It specifies that the content is in PDF format, allowing servers to handle it appropriately.

```
fetch('https://api.example.com/upload-pdf', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/pdf'
  },
  body: pdfFile // Assume pdfFile is a Blob or ArrayBuffer containing PDF data
});
```

Axios

Axios is a popular HTTP client library for making AJAX requests in JavaScript environments, including React applications. It provides a simple and elegant API for performing asynchronous HTTP requests to REST endpoints, interacting with APIs, and handling responses.

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

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

  useEffect(() => {
    const fetchData = async () => {
      const response = await axios.get('https://api.example.com/data');
      setData(response.data);
    };

    fetchData();
  }, []);

  return (
    <div>
      {data ? (
        <pre>{JSON.stringify(data, null, 2)}</pre>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
};
export default MyComponent;
```

Making Requests

Axios supports all HTTP methods (GET, POST, PUT, DELETE, etc.). Here are examples of making different types of requests:

GET Request

```
axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error(error);
  });
```

POST Request

```
axios.post('https://api.example.com/data', { key: 'value' })
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error(error);
  });
```

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

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

  useEffect(() => {
    axios.get('https://api.example.com/data')
      .then(response => setData(response.data))
      .catch(error => console.error('Error fetching data:', error));
  }, []);

  return (
    <div>
      {/* Render data */}
    </div>
  );
}

export default MyComponent;
```

Libraries like SWR or React Query

```
import React from 'react';
import useSWR from 'swr';

function MyComponent() {
  const { data, error } = useSWR('https://api.example.com/data', url =>
fetch(url).then(response => response.json()));

  if (error) return <div>Error fetching data</div>;
  if (!data) return <div>Loading...</div>;

  return (
    <div>
      {/* Render data */}
    </div>
  );
}

export default MyComponent;

import React from 'react';
import { useQuery } from 'react-query';

function MyComponent() {
  const { data, isLoading, isError } = useQuery('data', () =>
fetch('https://api.example.com/data').then(response => response.json()));

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error fetching data</div>;

  return (
    <div>
      {/* Render data */}
    </div>
  );
}

export default MyComponent;
```

async await react:-

Async functions, marked with the async keyword, allow us to write asynchronous code in a way that looks synchronous. Inside an async function, we can use the await keyword to pause the execution of the function until a promise is resolved, effectively waiting for the asynchronous operation to complete.

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

function MyComponent() {
  const [data, setData] = useState(null);
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get('https://api.example.com/data');
        setData(response.data);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      {/* Render data */}
    </div>
  );
}

export default MyComponent;
```

In this example, we define an async function named fetchData inside the useEffect hook. We use the await keyword to wait for the axios.get function to complete and resolve its promise, fetching data from the specified API endpoint. If an error occurs during the data fetching process, we catch it and log it to the console.

Promise:-

"In JavaScript, a promise is an object that represents the result of an asynchronous operation. It has three states: pending, fulfilled, and rejected. When a promise is pending, the asynchronous operation is still in progress. When it's fulfilled, the operation is successful, and when it's rejected, the operation has failed."

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

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

  useEffect(() => {
    axios.get('https://api.example.com/data')
      .then(response => {
        setData(response.data);
      })
      .catch(error => {
        console.error('Error fetching data:', error);
      });
  }, []);
  return (
    <div>
      {/* Render data */}
    </div>
  );
}

export default MyComponent;
```

In this example, we use the axios.get method to fetch data from an API endpoint. This method returns a promise that represents the asynchronous operation of fetching data. We then use the then method to handle the successful fulfillment of the promise by setting the fetched data to the component's state. If an error occurs during the data fetching process, we use the catch method to handle it and log the error."

strict mode:-

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

lifecycle in react:-

In React, the component lifecycle consists of a series of methods that are invoked at different stages of a component's existence. These methods can be broadly categorized into three phases: Mounting, Updating, and Unmounting. React provides lifecycle methods that allow developers to execute code at specific points in these phases.

Mounting Phase

The mounting phase covers the initial creation and insertion of a component into the DOM.

componentDidMount()

Description: Invoked immediately after a component is mounted (inserted into the DOM). Ideal for AJAX requests, setting up subscriptions, or modifying the DOM.

Updating Phase

The updating phase occurs when a component's state or props change.

shouldComponentUpdate(nextProps, nextState)

Description: Called to determine if a re-render is necessary. Returns true by default, allowing all updates. Useful for performance optimization.

componentDidUpdate(prevProps, prevState, snapshot)

Description: Invoked immediately after updating occurs. This method is not called for the initial render.

Unmounting Phase
The unmounting phase occurs when a component is being removed from the DOM.

componentWillUnmount()
Description: Called immediately before a component is destroyed. Use it to clean up subscriptions, timers, or any other resources that need to be released.

Error Handling
React also provides lifecycle methods for error handling.

static getDerivedStateFromError(error)

Description: Called when a descendant component throws an error. It allows setting an error state.

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

componentDidCatch(error, info)

Description: Called after an error has been thrown by a descendant component. Useful for logging error information.

```
componentDidCatch(error, info) {
  // Log error information
}
```

Functional Components and Hooks
With the introduction of React Hooks, many lifecycle methods can be managed within functional components using hooks like useEffect

useEffect:
Description: Combines the functionality of componentDidMount, componentDidUpdate, and componentWillUnmount.

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

function MyComponent() {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    // ComponentDidMount and componentDidUpdate logic
    return () => {
      // ComponentWillUnmount logic
    };
  }, [counter]); // Dependencies array

  return <div>{counter}</div>;
}
```

http methods :-

HTTP methods, also known as HTTP verbs, define the actions that can be performed on a given resource. Each method has a specific purpose and semantics. Here are the most commonly used HTTP methods:

GET

Purpose: Retrieve data from the server.
Usage: Used to request data from a specified resource.
Characteristics:
Safe: Does not alter the state of the server.
Idempotent: Multiple identical requests have the same effect as a single request.
Cacheable: Responses can be cached.

POST

Purpose: Submit data to the server to create or update a resource.
Usage: Used to send data to the server, typically resulting in a change in state or side effects on the server.
Characteristics:
Not idempotent: Multiple identical requests may have different effects.
Usually not cacheable.

PUT

Purpose: Update or create a resource at a specified URI.
Usage: Used to update an existing resource or create a new resource if it does not exist.
Characteristics:
Idempotent: Multiple identical requests have the same effect as a single request.

DELETE

Purpose: Delete a specified resource.
Usage: Used to delete the specified resource.
Characteristics:
Idempotent: Multiple identical requests have the same effect as a single request.

map function

The map function is a powerful array method in JavaScript that allows you to create a new array by applying a provided function to every element in the calling array. It is commonly used for transforming data.

browser router:-

React Router is a popular library for handling routing in React applications. It allows you to create single-page applications (SPAs) with navigation that feels like a multi-page app. react-router-dom is the version of React Router for web applications, and BrowserRouter is one of the primary components used to enable routing.

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch, Link } from 'react-router-dom';

// Define some components
function Home() {
  return <h2>Home</h2>;
}

function About() {
  return <h2>About</h2>;
}

function Users() {
  return <h2>Users</h2>;
}

// Main App component
function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/users">Users</Link>
            </li>
          </ul>
        </nav>

        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

export default App;

BrowserRouter:

Wraps the application and provides the routing context.
Uses the HTML5 history API for cleaner URLs (without the hash #).

Route:

Defines a path and the component that should be rendered when that path is matched.
Props: path, exact, component, render, and children.

Switch:

Renders the first child <Route> or <Redirect> that matches the location.
Ensures only one route is rendered at a time.

Link:

Provides navigation between routes.
Prevents full page reloads by handling the click events.

challenges faced in react:-

Developing applications with React can come with various challenges, especially as the application grows in complexity
State Management
Challenge: As the application grows, managing state across multiple components can become complex.

Solution:

Local State: Use the useState and useReducer hooks for managing local component state.
Global State: Utilize state management libraries such as Redux, MobX, or the Context API for managing global state.
Best Practices: Structure your state thoughtfully, avoid deeply nested state objects, and normalize state to reduce complexity.

Performance Optimization
Challenge: Ensuring that the application performs well, especially with large data sets or complex UIs.

Solution:

Memoization: Use React.memo to prevent unnecessary re-renders of functional components, and useMemo and useCallback hooks to memoize values and functions.
Lazy Loading: Implement code-splitting and lazy loading using React.lazy and Suspense to load components only when needed.
Virtualization: Use libraries like React Virtualized or React Window to efficiently render large lists and tables by rendering only visible items.

Handling Side Effects
Challenge: Managing side effects such as data fetching, subscriptions, and timers can be tricky.

Solution:

useEffect: Utilize the useEffect hook for handling side effects. Ensure proper cleanup by returning a cleanup function.
Custom Hooks: Create custom hooks to encapsulate and reuse side effect logic across components.

Component Communication
Challenge: Passing data and functions between deeply nested components can become cumbersome.

Solution:

Props: Use props for passing data and functions down the component tree.
Context API: Leverage the Context API to provide and consume data across the component tree without prop drilling.
State Management Libraries: Use Redux or MobX for more complex state management needs.

Sure. One significant challenge I faced was managing state in a large and complex React application. As the application grew, keeping the state synchronized and avoiding prop drilling became increasingly difficult. This complexity led to issues with maintainability and debugging.

To address this, I explored various state management solutions and decided to implement Redux. By structuring the state using Redux, I was able to centralize the state management, which made the application more predictable and easier to debug. I also used middleware like Redux Thunk to handle asynchronous actions, which simplified our data fetching logic.

Another challenge was optimizing the performance of our application, particularly with large data sets and complex UIs. We noticed that certain components were re-rendering unnecessarily, which degraded performance. To tackle this, I used React's memo function to memoize functional components, and implemented useMemo and useCallback hooks to memoize expensive calculations and functions. This reduced the number of re-renders and significantly improved the application's performance.

Additionally, we faced difficulties in testing our components effectively. Initially, our tests were not comprehensive and failed to cover many edge cases. To improve this, I introduced React Testing Library to our workflow, which encouraged writing tests that closely mimic user interactions. This, combined with Jest for unit testing, helped us achieve more reliable and maintainable test coverage.

Overall, these solutions led to a more maintainable codebase, better performance, and a higher level of confidence in our application through improved testing. These experiences have taught me the importance of choosing the right tools and approaches for the specific challenges faced in React development."

way to communicate from child component and parent component:-

In React, communication between a parent component and a child component is typically handled using props. The parent component can pass data to the child component via props, and the child component can communicate back to the parent by invoking callback functions passed down as props.

For example, here’s how you can pass data and a callback function from a parent to a child:

```
// ParentComponent.js
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const [message, setMessage] = useState('');

  const handleMessageChange = (newMessage) => {
    setMessage(newMessage);
  };

  return (
    <div>
      <h1>Parent Component</h1>
      <p>Message from Child: {message}</p>
      <ChildComponent onMessageChange={handleMessageChange} />
    </div>
  );
}

export default ParentComponent;

// ChildComponent.js
import React from 'react';

function ChildComponent({ onMessageChange }) {
  const handleClick = () => {
    onMessageChange('Hello from Child Component!');
  };
  return (
    <div>
      <h2>Child Component</h2>
      <button onClick={handleClick}>Send Message</button>
    </div>
  );
}

export default ChildComponent;
```

In this example, the ParentComponent passes a callback function handleMessageChange to the ChildComponent via props. When the button in the ChildComponent is clicked, it calls onMessageChange with a new message, which updates the state in the ParentComponent.

For more complex state management and communication between components that are not directly related, I use the Context API or a state management library like Redux. These tools allow you to create a global state that can be accessed and updated by any component in the application.

```
// MessageContext.js
import React, { createContext, useState } from 'react';

const MessageContext = createContext();

function MessageProvider({ children }) {
  const [message, setMessage] = useState('');

  return (
    <MessageContext.Provider value={{ message, setMessage }}>
      {children}
    </MessageContext.Provider>
  );
}

export { MessageContext, MessageProvider };


// ParentComponent.js
import React, { useContext } from 'react';
import { MessageContext } from './MessageContext';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const { message } = useContext(MessageContext);

  return (
    <div>
      <h1>Parent Component</h1>
      <p>Message from Child: {message}</p>
      <ChildComponent />
    </div>
  );
}

export default ParentComponent;

// ChildComponent.js
import React, { useContext } from 'react';
import { MessageContext } from './MessageContext';

function ChildComponent() {
  const { setMessage } = useContext(MessageContext);

  const handleClick = () => {
    setMessage('Hello from Child Component!');
  };

  return (
    <div>
      <h2>Child Component</h2>
      <button onClick={handleClick}>Send Message</button>
    </div>
  );
}

export default ChildComponent;
```

diff bw functional and class based components:-

Functional Components

Definition

Functional components are simpler and are defined as JavaScript functions. They receive props as an argument and return JSX.

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

State and Lifecycle

Hooks: With the introduction of Hooks in React 16.8, functional components can now use state and other React features. Hooks like useState, useEffect, etc., allow functional components to manage state and side effects.

Simpler: They are often easier to read and write, especially for components that are purely presentational or "dumb" components.

In React, there are two main types of components: functional components and class-based components. Each type has its own syntax and behavior. Here’s a breakdown of their differences:

Functional Components

Definition
Functional components are simpler and are defined as JavaScript functions. They receive props as an argument and return JSX.

Syntax
javascript
Copy code
```
function Greeting(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

<u>State and Lifecycle</u>
Hooks: With the introduction of Hooks in React 16.8, functional components can now use state and other React features. Hooks like useState, useEffect, etc., allow functional components to manage state and side effects.
Simpler: They are often easier to read and write, especially for components that are purely presentational or "dumb" components.

<u>Performance</u>
Functional components generally have a slight performance advantage due to their simpler structure and lack of method binding.

<u>Class-Based Components</u>
Definition
Class-based components are ES6 classes that extend from React.Component. They must define a render method that returns JSX.

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

In React, there are two main types of components: functional components and class-based components. Each type has its own syntax and behavior. Here's a breakdown of their differences:

Class-Based Components
Definition
Class-based components are ES6 classes that extend from React.Component. They must define a render method that returns JSX.

Syntax
javascript
Copy code

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```
<u>State and Lifecycle</u>
State and Methods: Class components manage state using this.state and update it with this.setState. They can also define lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.
More Boilerplate: They often require more boilerplate code, such as constructor functions and method binding.

Performance
Class components may have slightly more overhead due to the necessity of method binding and the larger component structure.

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

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

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}


import React, { Component } from 'react';

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }

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

usestate asynchronous or synchronous:-

The useState hook in React is synchronous in terms of updating the state immediately within the component's rendering process, but the updated state value does not take effect until the next render cycle. This can sometimes give the impression of being asynchronous because the state value you set is not immediately reflected in the component after the setState call within the same render cycle.

diff bw null and undefined:-

null

Type: null is an assignment value that represents the intentional absence of any object value. Its type is object.

Usage: null is typically used to explicitly indicate that a variable has no value. It is often used to reset or clear variables.

Assignment: null is explicitly assigned to a variable by the developer.

```
let a = null;
console.log(a); // Output: null
console.log(typeof a); // Output: "object"
```

undefined

Type: undefined is a primitive value that represents the absence of an assigned value. Its type is undefined.

Usage: undefined is the default value for uninitialized variables, function arguments that are not provided, and missing properties of objects.

Assignment: undefined can be assigned by the JavaScript engine automatically or explicitly by the developer, though explicitly assigning undefined is generally discouraged.

```
let b;
console.log(b); // Output: undefined
console.log(typeof b); // Output: "undefined"

function example(x) {
  console.log(x); // Output: undefined if no argument is passed
}
example();
```

Key Differences
Origin:

null is explicitly set by the developer to indicate "no value".
undefined is typically assigned by JavaScript when a variable is declared but not initialized, a function is called without all the arguments, or when trying to access an object property that doesn't exist.
Intent:

null indicates an intentional absence of value.
undefined indicates an unintentional absence of value or a value that hasn't been set yet.
Behavior:

null is often used to signify that an object or variable should be empty.
undefined is a signal that a variable has been declared but not yet assigned a value, or a function argument has not been provided.

error boundaries:-

Error boundaries are a feature in React that allow you to catch
JavaScript errors anywhere in the component tree below them, log
those errors, and display a fallback UI instead of crashing the
entire component tree. This feature is particularly useful
for enhancing the user experience by providing graceful error handling.

An error boundary is a React component that implements either static getDerivedStateFromError(error) or componentDidCatch(error, info) (or both). These lifecycle methods are used to catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.
Error boundaries are typically used to wrap components that might throw errors. When an error is caught, the error boundary can render a fallback UI instead of the crashed component tree.

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

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

  static getDerivedStateFromError(error) {
    // Update state so the next render shows the fallback UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    console.log(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

State Initialization:

The constructor initializes the state with hasError: false.
getDerivedStateFromError:

This static method is called when an error is thrown in a descendant component. It updates the state to hasError: true, triggering a re-render with the fallback UI.
componentDidCatch:

This lifecycle method is called after an error has been thrown by a descendant component. It receives two parameters:
error: The error that was thrown.
errorInfo: An object with a componentStack key containing information about which component threw the error.

render:

If an error has been caught (hasError: true), the fallback UI is rendered.
Otherwise, the children components are rendered normally.

```
import React from 'react';
import ErrorBoundary from './ErrorBoundary';
import MyComponent from './MyComponent';

const App = () => (
  <ErrorBoundary>
    <MyComponent />
  </ErrorBoundary>
);

export default App;
```

Multiple Error Boundaries:

You can use multiple error boundaries to isolate different parts of your application. This way, an error in one part of the UI does not affect other parts.

```
const App = () => (
  <div>
    <ErrorBoundary>
      <ComponentA />
    </ErrorBoundary>
    <ErrorBoundary>
      <ComponentB />
    </ErrorBoundary>
  </div>
);
```

for a parent mulitple child comp , need to render only child comp how to do that in react:-

```
import React, { useState } from 'react';
import ChildComponentA from './ChildComponentA';
import ChildComponentB from './ChildComponentB';
import ChildComponentC from './ChildComponentC';

const ParentComponent = () => {
  const [activeChild, setActiveChild] = useState('A');

  const renderActiveChild = () => {
    switch (activeChild) {
      case 'A':
        return <ChildComponentA />;
      case 'B':
        return <ChildComponentB />;
      case 'C':
        return <ChildComponentC />;
      default:
        return null;
    }
  };

  return (
    <div>
      <button onClick={() => setActiveChild('A')}>Show Child A</button>
      <button onClick={() => setActiveChild('B')}>Show Child B</button>
      <button onClick={() => setActiveChild('C')}>Show Child C</button>
      {renderActiveChild()}
    </div>
  );
};

export default ParentComponent;
```

higher order components:-

Higher-Order Components (HOCs) are a pattern in React for reusing component logic. An HOC is a function that takes a component and returns a new component with additional props, state, or behavior. HOCs are commonly used for cross-cutting concerns like logging, caching, and conditionally rendering components based on permissions or authentication.

```
const higherOrderComponent = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
};
```

why functional components used:-

Simplicity and Readability

Functional components are simpler and more readable compared to class components. They are essentially JavaScript functions that take props and return JSX, making them easier to understand and write, especially for developers who are new to React

Hooks for State and Lifecycle Management
With the introduction of React Hooks, functional components can now handle state and lifecycle events, which were previously only possible in class components. Hooks like useState and useEffect allow functional components to be just as powerful and even more flexible.
Avoiding this Keyword Confusion
Answer:
"Functional components eliminate the need for the this keyword, which can often lead to confusion and bugs, especially for those less experienced with JavaScript. This makes the code more intuitive and reduces potential errors."
Performance Benefits
Answer:
"Functional components are often more performant because they do not have the overhead of managing an instance of a class. This can lead to faster rendering and updating of components, particularly in complex applications."

5. Encouraging Best Practices
Answer:
"Using functional components encourages the use of best practices such as keeping components small and focused, which aligns with React's philosophy of creating reusable components. They also promote a more functional programming style, which can lead to more predictable and maintainable code."

6. Enhanced Code Reusability
Answer:
"Functional components, combined with hooks, enhance code reusability. Custom hooks allow you to extract component logic into reusable functions, which can be shared across multiple components, leading to a cleaner and more DRY (Don't Repeat Yourself) codebase."

Example

```
const useCounter = (initialValue = 0) => {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  return { count, increment, decrement };
};

const CounterComponent = () => {
  const { count, increment, decrement } = useCounter(10);
  return (
    <div>
      <p>{count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
```

I prefer using functional components in React for several reasons. Firstly, they are simpler and more readable, making the code easier to understand and maintain. The introduction of hooks allows functional components to manage state and lifecycle events without the complexity of class components. This means I can avoid the confusion and potential bugs associated with the this keyword. Additionally, functional components often lead to better performance and encourage best practices by promoting smaller, focused components. They also enhance code reusability through custom hooks, which makes the codebase cleaner and more maintainable. Finally, functional components are easier to test and debug due to their straightforward nature. Overall, functional components align well with modern React development and offer several practical advantages over class components

Redux toolkit:-
The createAsyncThunk function in Redux Toolkit is used to handle asynchronous actions (thunks) in a Redux application. The thunkAPI.getState method, which is available inside the thunk, allows you to access the current state of the Redux store.

createAsyncThunk: Defines an asynchronous action. It takes a string (action type) and an async function that performs the API call. Inside the async function, thunkAPI.getState() is used to access the current Redux state.

thunkAPI.getState: Used inside the thunk to access the current state of the Redux store. This can help in making decisions based on the current state, such as skipping an API call if the data is already available.

Slice with extraReducers: Handles different states of the async thunk (pending, fulfilled, and rejected), updating the state accordingly.

Store Setup: Integrates the slice reducer into the Redux store.

React Component: Dispatches the async thunk to fetch user data and renders the UI based on the state (loading, succeeded, failed).
configureStore:

Simplifies store setup with sensible defaults.
Automatically combines slice reducers, adds Redux middleware, and includes Redux DevTools integration.
createSlice:
Generates a slice of the Redux state, with actions and reducers automatically created.
Simplifies the process of defining reducers and actions for a specific part of the state.
createAsyncThunk:
Handles asynchronous logic and side effects, such as data fetching.
Automatically dispatches lifecycle actions (pending, fulfilled, rejected) to handle different states of the async operation.
createAsyncThunk:
Handles asynchronous logic and side effects, such as data fetching.
Automatically dispatches lifecycle actions (pending, fulfilled, rejected) to handle different states of the async operation.
createSelector (from Reselect):
Efficiently derives data from the Redux store using memoized selectors.
Prevents unnecessary re-renders by only recalculating derived data when the input data changes.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1; },
    decrement: (state) => { state.value -= 1; },
    incrementByAmount: (state, action) => { state.value += action.payload; }
  }
});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;

import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer
  }
});

export default store;

import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';
import axios from 'axios';

export const fetchUser = createAsyncThunk('user/fetchUser', async (userId) => {
  const response = await axios.get(/api/users/${userId});
  return response.data;
});
```

```
const userSlice = createSlice({
  name: 'user',
  initialState: { user: null, status: 'idle', error: null },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUser.pending, (state) => { state.status = 'loading'; })
      .addCase(fetchUser.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.user = action.payload;
      })
      .addCase(fetchUser.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.err…
```

useSelector

The useSelector hook is provided by react-redux and is used to extract data from the Redux store state. This hook allows you to access the Redux state and select the part of the state your component needs.

```
// Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './counterSlice';

const Counter = () => {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
<div>
<p>{count}</p>
<button onClick={() => dispatch(increment())}>Increment</button>
<button onClick={() => dispatch(decrement())}>Decrement</button>
</div>
  );
};

export default Counter;
```

useReducer

The useReducer hook is a React hook (not specific to Redux) that is used to manage local state in a component. It is an alternative to useState and is usually preferred when you have complex state logic or when the next state depends on the previous state.

useSelector

The useSelector hook is provided by react-redux and is used to extract data from the Redux store state. This hook allows you to access the Redux state and select the part of the state your component needs.

Comparison: useSelector vs. useReducer

useSelector:

Used with Redux to select state from the Redux store.
Ideal for accessing global state managed by Redux.
Can be combined with useDispatch to dispatch actions.
useReducer:

Used with React to manage local state within a component.
Suitable for complex local state logic.
Reducer function and initial state are defined within the component or module.

*create context

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

const TransferFormContext = createContext();
```

*create use context

```
export const useTransferFormContext = () => useContext(TransferFormContext);
```

*export provider

```
export const FormProvider = ({ children }) => {
//codes
 return (
   <TransferFormContext.Provider
     value={{
     //values to pass in context
     }}
   >
     {children}
   </TransferFormContext.Provider>
  );
};
```

*import in other necessary parent component and wrap it

```
import {
  FormProvider,
  useTransferFormContext,
} from "../../context/TransferFormContext";

<FormProvider>
          <OMAdminServices
            exclusionAppRef={exclusionAppRef}
            evidenceBaseUrlValue={evidenceBaseUrlValue}
          />
        </FormProvider>
```

Can you explain what Redux is?:-
Redux is a predictable state container for JavaScript applications. It is commonly used with libraries like React to manage the state of an application. Redux provides a centralized store to hold the state,

and rules ensure that the state can only be updated in a predictable way through actions and reducers

The store is a single JavaScript object that holds the entire state of your application. There is only one store in a Redux application, making it the single source of truth

Actions are plain JavaScript objects that describe an event that has occurred, typically containing a type property and some data. They are the only source of information for the store.

Reducers are pure functions that take the current state and an action as arguments and return a new state. They specify how the state changes in response to an action.

Dispatch is a method used to send actions to the Redux store. This is how we trigger state changes
Selectors are functions that extract specific pieces of data from the store. They help keep the logic of getting data out of the store centralized and reusable

Action:

"An action is dispatched by a user interaction or an event."
Reducer:

"The action is passed to the reducer, which processes it and returns a new state."
Store:

"The store updates its state based on the reducer's output."
View:

"The view re-renders based on the new state."

```
// Action
const increment = () => ({ type: 'INCREMENT' });

// Reducer
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    default:
      return state;
  }
};

// Store
const store = createStore(counterReducer);

// Dispatching an action
store.dispatch(increment());
console.log(store.getState()); // { count: 1 }
```

Additional Features of Redux
Apart from the core functionality, Redux has several additional features and libraries that enhance its

capabilities. Some of these include state persistence, devtools integration, and advanced middleware for handling side effects. Let's dive into some of these features and provide examples.

Redux Persist is a library used to save the Redux store's state to local storage or another storage mechanism and rehydrate it upon app reload. This is useful for persisting user sessions, app settings, or other data that should persist across page reloads.

```
npm install redux-persist
```

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage'; // defaults to localStorage for web
import counterReducer from './counterSlice';

const persistConfig = {
  key: 'root',
  storage,
};

const persistedReducer = persistReducer(persistConfig, counterReducer);

const store = configureStore({
  reducer: {
    counter: persistedReducer,
  },
});

const persistor = persistStore(store);

export { store, persistor };


// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { PersistGate } from 'redux-persist/integration/react';
import { store, persistor } from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>,
  document.getElementById('root')
);

// Counter.js
```

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './counterSlice';

const Counter = () => {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <p>{count}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
};

export default Counter;
```

Redux DevTools
Redux DevTools is a powerful tool for debugging Redux applications. It allows you to inspect every change to the state and the actions that caused those changes, time travel, and revert state changes.

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
import { composeWithDevTools } from 'redux-devtools-extension';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
  devTools: composeWithDevTools(),
});

export default store;
```