

# What is react js

?

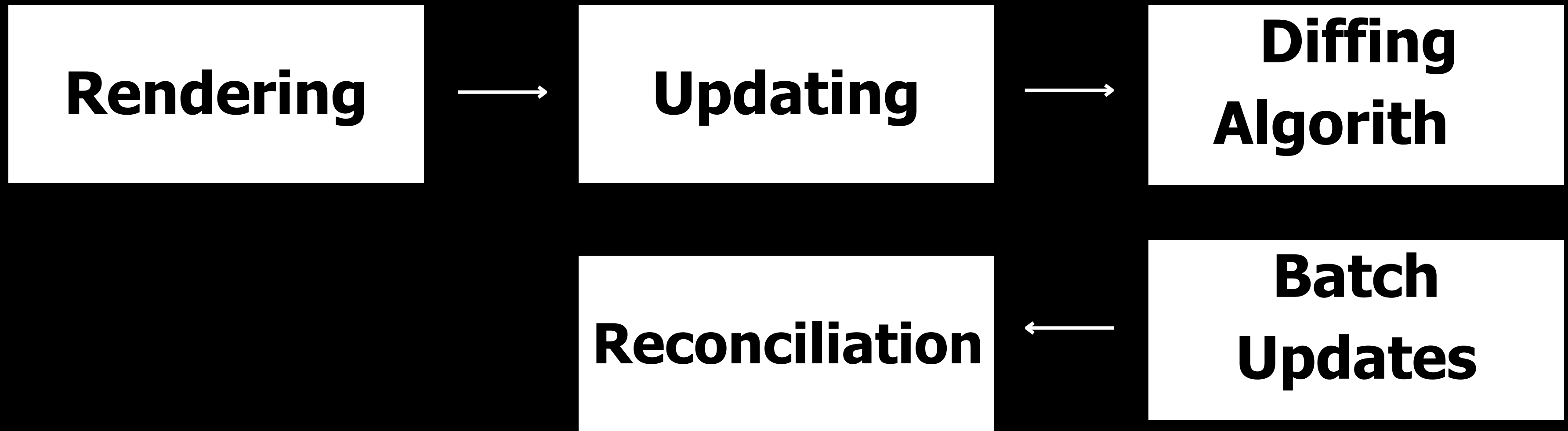
React is an open-source JavaScript library developed for building user interfaces, particularly for single-page applications.

# what are the major features of react?

- **Virtual DOM:** React uses a virtual DOM to improve performance by minimizing direct DOM manipulations.
- **JSX:** JSX stands for JavaScript XML, which allows writing HTML in React components.
- **Components:** React is component-based, meaning the UI is built using reusable components.
- **One-way Data Binding:** Data flows in one direction, making the application easier to understand and debug.
- **High Performance:** React optimizes updates by using a virtual DOM and efficiently re-rendering components.
- **Unidirectional Data Flow:** Data flows in a single direction, which provides better control over the entire application.

# What is virtual DOM and how it works ?

Virtual DOM is a lightweight, in-memory representation of the real DOM elements generated by React components. React keeps a copy of the actual DOM structure in memory, called the Virtual DOM, and uses it to optimize updates and rendering.



# What are components in react

?

Components are the building blocks of a React application. They are reusable pieces of UI that can be nested, managed, and handled independently.

**Class Based Components**

**Functional Components**

# Explain Class components with example

```
import React, { Component } from 'react';  
  
class ClassComponentExample extends Component {  
  render() {  
    return <h1>Hello</h1>;  
  }  
}  
  
export default ClassComponentExample;
```

# Explain functional components with example

```
import React from 'react';

function FunctionalComponent() {
  return <h1>Hello</h1>;
}

export default FunctionalComponent;
```

```
import React from 'react';

✓ const FunctionalComponent = () => {
  return <h1>Hello</h1>;
}

export default FunctionalComponent;
```

# What is JSX?

- JSX stands for JavaScript XML.
- It allows us to write HTML elements in JavaScript and place them in the DOM without using methods like createElement() or appendChild().

```
import React from "react";

function HelloWorld() {
  return <h1>Hello, world!</h1>;
}

export default HelloWorld;
```



```
React.createElement('h1', null, 'Hello, world!')
```



# How to export and import components ?

We can export components using export default or named exports, and import them using import.

```
import React from "react";
```

```
const Home = () => {  
  return <h1>Home!</h1>;  
};
```

```
export default Home;
```

```
import React from "react";  
import Home from "../home";
```

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

```
export default App;
```



# How to use nested components ?

```
import React from 'react';

function Menus(){
  return <div>Menus</div>
}

function Header() {
  return (
    <header>
      <h1>Header content</h1>
      <Menus/>
    </header>
  );
}

export default Header;
```

```
import React from 'react';
import Header from './Header';
import Footer from './Footer';

function App() {
  return (
    <div>
      <Header />
      <h1>Main Content</h1>
      <Footer />
    </div>
  );
}

export default App;
```

# What is state in react ?

In React, state is an object that represents the parts of the app that can change. Each component can have its own state, which can be managed within the component and used to render the UI. When the state changes, React re-renders the component to reflect the new state.

# What is state in react ?

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    // Initializing state
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <h1>Count: {this.state.count}</h1>
    );
  }
}
export default Counter;
```

```
import React, { Component } from 'react';
class Counter extends Component {
  state = {
    count : 0
  }
  render() {
    return (
      <h1>Count: {this.state.count}</h1>
    );
  }
}
export default Counter;
```

```
import React, { useState } from 'react';
function Counter() {
  // Initializing state with useState hook
  const [count, setCount] = useState(0);

  return (
    <h1>Count: {count}</h1>
  );
}

export default Counter;
```



# How to update state in react ?

State in React is updated using the `setState` method in class components or the `useState` hook in functional components.

```
import React, { Component } from "react";
class Counter extends Component {
  state = { count: 0 };

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

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

export default Counter;
```

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

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

  function handleClick(){
    setCount(count + 1)
  }

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

export default Counter;
```

# What is setState callback?

The setState method can accept a callback function as the second argument, which is executed once the state has been updated and the component has re-rendered.

```
import React, { Component } from "react";
class Counter extends Component {
  state = {
    count: 0,
  };

  increment = () => {
    this.setState({ count: this.state.count + 1 },
      () => {
        console.log("State updated:", this.state.count);
      });
  };

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

export default Counter;
```

# Why you should not update state directly , explain with example

Updating state directly does not trigger a re-render of the component, leading to inconsistencies in the UI. Instead, always use `setState` or state hooks.

```
class Counter extends Component {
  state = {
    count: 0,
  };
  increment = () => {
    // Incorrect way
    this.state.count = this.state.count + 1;
    // Does not re-render the component

    // Correct way
    this.setState({ count: this.state.count + 1 });
    // Triggers a re-render
  };
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.increment}>Click me</button>
      </div>
    );
  }
}
```



# What are props in react ?

Props are used to pass data and event handlers to child components. Props ensure a one-way data flow, from parent to child.

Props cannot be modified by the child component that receives them.

```
import React from 'react';

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

// Parent Component
function App() {
  return (
    <div>
      <Greeting name="John" />
      <Greeting name="Jane" />
    </div>
  );
}

export default App;
```

# What is difference between state and props

?

- **State is a built-in object used to store data that may change over the lifecycle of a component. It is managed within the component itself.**
- **State is mutable. It can be updated using the `setState` method in class components or the `useState` hook in functional components.**
- **State is local to the component and cannot be accessed or modified by child components.**

- **Props (short for properties) are used to pass data from a parent component to a child component. They are read-only and immutable within the child component.**
- **Props are immutable. Once passed to a child component, they cannot be modified by the child.**
- **Props are passed from a parent component to a child component and can be accessed by the child.**

# What is lifting state up in react

?

Lifting State Up is a pattern in React where state is moved up to the closest common ancestor of components that need to share that state.

Single Source of Truth: By managing the state in the parent component, you ensure that the state is consistent across multiple child components.

Simplified State Management: The state logic is centralized, making it easier to maintain and debug.

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

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

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

  return (
    <div>
      <h1>Lifting State Up Example</h1>
      <Counter count={count} onIncrement={handleIncrement}/>
    </div>
  );
}

export default App;
```

```
import React from 'react';

function Counter({ count, onIncrement }) {
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={onIncrement}>Increment</button>
    </div>
  );
}

export default Counter;
```

# What is children prop in react

?

The children prop is a special property in React used to pass the content that is nested inside a component.

```
import React from 'react';

function Container(props) {
  return (
    <div>
      {props.children}
    </div>
  );
}

export default Container;
```

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

function App() {
  return (
    <div>
      <h1>Using the children Prop</h1>
      <Container>
        <p>This is a paragraph inside the Container.</p>
      </Container>
      <Container>
        <h2>This is a heading inside another Container.</h2>
        <button>Click Me</button>
      </Container>
    </div>
  );
}

export default App;
```



# What is defaultProps in React?

defaultProps is used to set default values for the props in a component.

```
function Greeting(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}  
  
Greeting.defaultProps = {  
  name: 'Guest',  
};  
  
// Usage  
<Greeting /> // Renders "Hello, Guest!"
```

# What are fragments in react and its advantages ?

Fragments allow you to group multiple elements without adding extra nodes to the DOM.

```
import React from 'react';

function List() {
  return (
    <React.Fragment>
      <li>Item 1</li>
      <li>Item 2</li>
    </React.Fragment>
  );
}



// Short syntax
function List() {
  return (
    <>
      <li>Item 1</li>
      <li>Item 2</li>
    </>
  );
}
```



# How to use styling in react is ?

We can use inline styles, CSS stylesheets, or CSS-in-JS libraries like styled- components.

```
function StyledComponent() {  
  return (  
    <div  
      style={{  
        color: "blue",  
        backgroundColor: "lightgray",  
      }}  
    >  
      This is a styled component  
    </div>  
  );  
}
```

```
/* styles.css */  
.container {  
  color:  blue;  
  background-color:  lightgray;  
}  
  
import './styles.css';  
  
function StyledComponent() {  
  return <div className="container">This is a styled component</div>;  
}
```

# How can you conditionally render components in React?

We can use JavaScript conditional operators (like if, &&, ? :) to conditionally render components.

```
function Greeting({ isLoggedIn , flag }) {  
  
  if (isLoggedIn) {  
    return <h1>Welcome back!</h1>;  
  } else {  
    return flag ? <h1>Please sign up.</h1>:  
              <h1>Please sign in.</h1>  
  }  
  
}  
  
// Usage  
<Greeting isLoggedIn={true} flag ={false} />
```

# How to render list of data in react ?

We can use the map function to iterate over an array and render each item.



```
function NumberList({ numbers }) {  
  const listItems = numbers.map((number) =>  
    <li key={number.toString()}>{number}  
  </li>);  
  return <ul>{listItems}</ul>;  
}
```

// Usage

```
<NumberList numbers={[1, 2, 3, 4, 5]} />
```

# What is key prop?

The key prop is a unique identifier for each element in a list, used by React to identify which items have changed, are added, or removed.



```
function NumberList({ numbers }) {  
  const listItems = numbers.map((number) =>  
    <li key={number.toString()}>{number}  
  </li>);  
  return <ul>{listItems}</ul>;  
}
```

// Usage

```
<NumberList numbers={[1, 2, 3, 4, 5]} />
```



# Why indexes for keys are not recommended?

Using indexes as keys can lead to performance issues and unexpected behavior when list items are reordered or removed. Keys should be unique and stable.

# How to handle buttons in react ?

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

function Counter() {
  // Initialize state with useState hook
  const [count, setCount] = useState(0);
  // Event handler function for button click
  const handleIncrement = () => {
    setCount(count + 1);
  };
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}

export default Counter;
```

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

class Counter extends Component {
  state = {
    count : 0
  };
  // Event handler function for button click
  handleIncrement = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.handleIncrement}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```



# How to handle inputs in react ?

We can use controlled components where form data is handled by the component's state.

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

function NameForm() {
  const [name, setName] = useState("");

  const handleChange = (event) => {
    setName(event.target.value);
  };

  return (
    <label>
      Name:
      <input type="text" value={name} onChange={handleChange} />
    </label>
  );
}

export default NameForm;
```

# Explain lifecycle methods in react

Lifecycle methods in React are special methods that get called at different stages of a component's lifecycle.

- Mounting: When a component is being inserted into the DOM.
- Updating: When a component's state or props change.

Unmounting: When a component is being removed from the DOM.

```
class LifecycleMethods extends React.Component {  
  componentDidMount() {  
    console.log('Component mounted');  
  }  
  
  componentDidUpdate(prevProps, prevState) {  
    console.log('Component updated');  
  }  
  
  componentWillUnmount() {  
    console.log('Component will unmount');  
  }  
  
  render() {  
    return <div>Lifecycle Methods</div>;  
  }  
}
```

# What are the popular hooks in react and explain it's usage ?

- useState: Manages state in functional components.
- useEffect: Manages side effects in functional components.
- useContext: Consumes context in functional components.
- useReducer: Manages state with a reducer function. For more complex state management
- useRef: Accesses DOM elements or stores mutable values.
- useCallback: performance improvement
- usecase useMemo: performance improvement usecase

# What is useState and how to manage state using it ?

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

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

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

export default Counter;
```

# What is useEffect hook and how to manage side effects ?

useEffect is a hook that manages side effects like data fetching, subscriptions, or manually changing the DOM.

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

function DataFetcher() {
  useEffect(() => {}, []); // Empty array means this effect runs only once

  useEffect(() => {}, [dependency]); // run if dependency value changes

  useEffect(() => {
    return () => {};
  }, []); // cleanup method

  return <div></div>;
}

export default DataFetcher;
```



# How to implement data fetching in react is ?

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

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

  async function fetchData() {
    const response = await fetch("api-endpoint");
    const result = await response.json();

    if (result) setData(result);
  }

  useEffect(() => {
    fetchData();
  }, []); // Empty array means this effect runs only once

  return <div>{data ? data.title : "Loading..."}</div>;
}

export default DataFetchingExample;
```



# How to manage loading state ?

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

function DataFetchingExample() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false)

  async function fetchData() {
    setLoading(true)
    const response = await fetch("api-endpoint");
    const result = await response.json();

    if (result) {
      setData(result);
      setLoading(false)
    }
  }

  if(loading) return <h1>Loading data! Please wait</h1>

  useEffect(() => {
    fetchData();
  }, []); // Empty array means this effect runs only once

  return <div>{data ? data.title : "Loading..."}</div>;
}

export default DataFetchingExample;
```

# What is prop drilling and how to avoid it ?

Prop drilling occurs when you pass data through many layers of components. It can be avoided using the Context API or state management libraries like Redux.

```
function CompA(){  
  return <CompB title="Prop drilling"/>  
}  
  
function CompB({title}){  
  return <CompC title={title}/>  
}  
  
function CompC({title}){  
  return <CompD title={title}/>  
}  
  
function CompD({title}){  
  return <h3>{title}</h3>  
}
```

# What is the Context API in React, and why is it used?

Context API in React provides a way to share values (like data or functions) between components without having to pass props through every level of the component tree. It is used to avoid prop drilling.

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

// Create a Context
const MyContext = createContext();

// Provider Component
export default function MyProvider({ children }) {
  const value = "Hello, World!";
  return (
    <MyContext.Provider value={value}>
      {children}
    </MyContext.Provider>
  );
}

export default function App(){
  return <MyProvider><div>Child Components</div></MyProvider>
}
```



# How do you consume context using the useContext hook?

The useContext hook allows functional components to access context values directly.

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

function MyComponent() {
  const value = useContext(MyContext);
  return <div>{value}</div>;
}

export default MyComponent
```



# How can you update context values?

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

const MyContext = createContext();

function MyProvider({ children }) {
  const [value, setValue] = useState("Hello, World!");
  return (
    <MyContext.Provider value={{ value, setValue }}>
      {children}
    </MyContext.Provider>
  );
}

function MyComponent() {
  const { value, setValue } = useContext(MyContext);
  return (
    <div>
      <p>{value}</p>
      <button onClick={() => setValue("New Value")}>Update</button>
    </div>
  );
}
```

# How do you use multiple Contexts in a single component?

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

const FirstContext = createContext();
const SecondContext = createContext();

function MyProvider({ children }) {
  return (
    <FirstContext.Provider value="First Value">
      <SecondContext.Provider value="Second Value">
        {children}
      </SecondContext.Provider>
    </FirstContext.Provider>
  );
}

function MyComponent() {
  const firstValue = useContext(FirstContext);
  const secondValue = useContext(SecondContext);
  return (
    <div>
      <p>{firstValue}</p>
      <p>{secondValue}</p>
    </div>
  );
}
```

# What are the advantages of using the Context API over prop drilling?

Context API reduces the need for prop drilling, making the code more readable and maintainable. It allows for easy sharing of state and functions across the component tree without passing props through every level.

```
// Without Context API (prop drilling)
<Parent>
  <Child>
    <GrandChild value={value} />
  </Child>
</Parent>

// With Context API
<MyProvider>
  <GrandChild />
</MyProvider>
```



# What is the useReducer hook, and when should you use it?

The useReducer hook is used for state management in React. It is suitable for handling more complex state logic compared to useState.

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

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}
```



# Can you use useReducer with complex state objects?

```
const initialState = { count: 0, text: '' };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { ...state, count: state.count + 1 };
    case 'updateText':
      return { ...state, text: action.payload };
    default:
      throw new Error();
  }
}
```

# How do you pass additional arguments to the reducer function?

```
function reducer(state, action) {  
  switch (action.type) {  
    case 'update':  
      return { ...state, value: action.payload.value };  
    default:  
      throw new Error();  
  }  
}  
  
dispatch({ type: 'update', payload: { value: 42 } });
```

# How do you handle side effects with useReducer?

```
const initialState = { data: null, loading: true };

function reducer(state, action) {
  switch (action.type) {
    case 'fetchSuccess':
      return { data: action.payload, loading: false };
    default:
      throw new Error();
  }
}

export default function DataFetcher() {
  const [state, dispatch] = useReducer(reducer, initialState);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => dispatch({ type: 'fetchSuccess', payload: data }));
  }, []);

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

  return <div>Data: {state.data}</div>;
}
```



# What is useRef

## hook ?

The useRef hook is used to access and interact with DOM elements directly and to persist mutable values across renders without causing re-renders.

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

export default function FocusInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

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



# How can useRef be used to store mutable values?

useRef can store any mutable value, and changes to the ref do not cause re-renders.

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

function Counter() {
  const countRef = useRef(0);

  const increment = () => {
    countRef.current += 1;
    console.log(countRef.current);
  };

  return <button onClick={increment}>Increment</button>;
}
```

# What is forwardRef and when would you use it?

forwardRef is a function that allows you to pass refs through components to access DOM elements or child component instances.

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

const MyInput = forwardRef((props, ref) => (
  <input ref={ref} {...props} />
));

function App() {
  const inputRef = useRef(null);

  return <MyInput ref={inputRef} />;
}
```

# How to manage forms in react ?

Forms in React can be managed using controlled components where form data is handled by the component's state.

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

function Form() {
  const [name, setName] = useState('');

  const handleChange = (event) => {
    setName(event.target.value);
  };

  const handleSubmit = (event) => {
    alert('A name was submitted: ' + name);
    event.preventDefault();
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" value={name} onChange={handleChange} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default Form;
```

# What are Custom Hooks and Why Do We Need Them?

Custom Hooks in React are JavaScript functions that allow you to reuse stateful logic across multiple components. They enable you to extract and share common logic without repeating code, promoting code reusability and separation of concerns.

- **Code Reusability:** Custom hooks allow you to reuse stateful logic without duplicating code.
- **Separation of Concerns:** They help separate the logic from the component's structure, making the code more modular and easier to maintain.

**Cleaner Code:** By moving common logic into custom hooks, components become cleaner and more focused on their core responsibilities.



# Implement useFetch custom hook/Custom hook example ?

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

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

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch(url);
        const result = await response.json();
        setData(result);
      } catch (error) {
        console.error('Error fetching data:', error);
      } finally {
        setLoading(false);
      }
    }
    fetchData();
  }, [url]);

  return { data, loading };
}

export default useFetch;
```

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

function DataComponent() {
  const { data, loading } = useFetch('https://jsonplaceholder.typicode.com/todos/1');

  if (loading) return <p>Loading...</p>;
  if (!data) return <p>No data</p>;

  return (
    <div>
      <h1>{data.title}</h1>
      <p>ID: {data.id}</p>
      <p>Completed: {data.completed ? 'Yes' : 'No'}</p>
    </div>
  );
}

export default DataComponent;
```

# Implement useWindowSize custom hook

```
import { useState, useEffect } from 'react';
function useWindowSize() {
  // Initialize state with the current window dimensions
  const [windowSize, setWindowSize] = useState({
    width: window.innerWidth,
    height: window.innerHeight,
  });
  // Effect to handle window resize events
  useEffect(() => {
    // Handler function to update state with new window dimensions
    function handleResize() {
      setWindowSize({
        width: window.innerWidth,
        height: window.innerHeight,
      });
    }

    // Add event listener for window resize
    window.addEventListener('resize', handleResize);

    // Cleanup function to remove event listener
    return () => {
      window.removeEventListener('resize', handleResize);
    };
  }, []); // Empty dependency array ensures effect runs only on mount and unmount
  return windowSize;
}
export default useWindowSize;
```

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

function WindowSizeComponent() {
  // Use the custom hook to get the current window size
  const { width, height } = useWindowSize();

  return (
    <div>
      <h1>Window Size</h1>
      <p>Width: {width}px</p>
      <p>Height: {height}px</p>
    </div>
  );
}

export default WindowSizeComponent;
```

# What is React Router DOM and why is it used?

React Router DOM is a routing library built on top of React Router. It enables dynamic routing in web applications, allowing you to define routes and navigate between different components without reloading the page.



# How do you create a basic route in React Router DOM?

A basic route is created using the Route component, which maps a URL path to a specific element

```
<Route path="/home" element={<Home />} />
```



# How to implement basic routing using react router dom ?

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const App = () => (
  <BrowserRouter>
    <Routes>
      <Route path="/home" element={<Home />} />
      <Route path="/login" element={<Login />} />
    </Routes>
  </BrowserRouter>
);

export default App
```

# How to create a link to another route using React Router DOM?

Use the Link component to create navigation links.

```
import { Link } from 'react-router-dom';

const Navigation = () => (
  <nav>
    <Link to="/home">Home</Link>
    <Link to="/about">About</Link>
  </nav>
);

export default Navigation
```

# How do you use URL parameters / Dynamic routing in React Router DOM?

```
import { useParams } from 'react-router-dom';

const User = () => {
  const { userId } = useParams();
  return <div>User ID: {userId}</div>;
};

// Route definition
<Route path="/user/:userId" element={<User />} />
```

# How can you perform a redirect in React Router DOM?

Use the Navigate component to perform a redirect.

```
import { Navigate } from 'react-router-dom';

const Login = () => {
  const isLoggedIn = true;
  if (isLoggedIn) {
    return <Navigate to="/home" />;
  }
  return <div>Please log in</div>;
};

export default Login
```



# What is a Routes component in React Router DOM ?

The Routes component is used to define a set of routes, where only the first matching route is rendered.

```
import { Routes, Route } from 'react-router-dom';

const App = () => (
  <Routes>
    <Route path="/home" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="/contact" element={<Contact />} />
  </Routes>
);

export default App;
```

# How do you handle nested routes in React Router DOM?

```
import { Routes, Route, Outlet } from 'react-router-dom';

const Dashboard = () => (
  <div>
    <h1>Dashboard</h1>
    <Outlet />
  </div>
);

const App = () => (
  <Routes>
    <Route path="/dashboard" element={<Dashboard />}>
      <Route path="profile" element={<Profile />} />
      <Route path="settings" element={<Settings />} />
    </Route>
  </Routes>
);

export default App;
```

# How can you handle 404 errors (not found) in React Router DOM ?

Use a Route without a path prop inside Routes to catch all unmatched routes.

```
import { Routes, Route } from 'react-router-dom';

const NotFound = () => <h1>404 - Not Found</h1>;

const App = () => (
  <Routes>
    <Route path="/home" element={<Home />} />
    <Route path="/about" element={<About />} />
    <Route path="*" element={<NotFound />} />
  </Routes>
);

export default App;
```



# How do you programmatically navigate using React Router DOM ?

Use the `useNavigate` hook to navigate programmatically within your components.

```
import { useNavigate } from 'react-router-dom';  
  
const App = () => {  
  const navigate = useNavigate();  
  const handleLogin = () => {  
    // Perform login logic  
    navigate('/home');  
  };  
  
  return <button onClick={handleLogin}>Log in</button>;  
};  
  
export default App;
```



# Explain useCallback hook with example

The useCallback hook is used to memoize callback functions. This means that the function provided to useCallback will only be recreated if one of its dependencies has changed. This is particularly useful when passing callbacks to child components that are optimized with React.memo, as it can prevent unnecessary renders.

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

const MyComponent = ({ expensiveComputation }) => {
  const [count, setCount] = useState(0);

  const memoizedCallback = useCallback(() => {
    expensiveComputation();
  }, [expensiveComputation]);

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

const MyChildComponent = React.memo(({ callback }) => {
  // This component will not re-render unless the callback changes
  return <button onClick={callback}>Run Expensive Computation</button>;
});
```

# Explain useMemo hook with example

The useMemo hook is used to memoize expensive calculations so that they are not recalculated on every render. It takes a function to compute a value and an array of dependencies, and it only recomputes the value when one of the dependencies has changed.

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

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

  const computeExpensiveValue = (value) => {
    // Simulate an expensive computation
    for (let i = 0; i < 1000000000; i++) {}
    return value * 2;
  };

  const expensiveValue = useMemo(() => {
    return computeExpensiveValue(count);
  }, [count]);

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

# Explain React.memo with example

React.memo is a higher-order component that memoizes the result of a component. It prevents the component from re-rendering unless the props have changed. This is useful for optimizing performance by avoiding unnecessary renders of pure components.

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

const MyComponent = memo(({ value }) => {
  // This component will only re-render if the 'value' prop changes
  return <h1>{value}</h1>;
});

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

  return (
    <div>
      <MyComponent value={count} />
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};
```

# Explain the reconciliation process in React and how it works.

Reconciliation is the process React uses to update the DOM efficiently. It involves comparing the new virtual DOM with the previous one and determining the minimum number of changes needed to update the actual DOM.

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

const App = () => {
  const [items, setItems] = useState(['Item 1', 'Item 2', 'Item 3']);

  const addItem = () => {
    setItems([...items, `Item ${items.length + 1}`]);
  };

  return (
    <div>
      <button onClick={addItem}>Add Item</button>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
};
```



# What are Pure components ?

PureComponent is a base class in React that implements shouldComponentUpdate with a shallow prop and state comparison. It helps prevent unnecessary re-renders by ensuring that the component only re-renders when there are actual changes in props or state.

```
class ChildComponent extends PureComponent {
  render() {
    console.log('Child component rendered');
    return <div>Count: {this.props.count}</div>;
  }
}

class ParentComponent extends React.Component {
  state = { count: 0 };

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

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

# Explain higher order component with example

A Higher-Order Component (HOC) is a function that takes a component and returns a new component with added functionality. HOCs are used for reusing component logic and enhancing components with additional behavior.

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

const withLoading = (Component) => {
  return class WithLoading extends React.Component {
    state = { isLoading: true };

    componentDidMount() {
      setTimeout(() => {
        this.setState({ isLoading: false });
      }, 1000);
    }

    render() {
      if (this.state.isLoading) {
        return <p>Loading...</p>;
      }
      return <Component {...this.props} />;
    }
  };
};

const MyComponent = () => {
  return <h1>Component is loaded!</h1>;
};

export default withLoading(MyComponent);
```

# What is redux , explain core principles

Redux is a predictable state container for JavaScript apps. Redux acts as a centralized store for state management in your application.

- Single Source of Truth: The state of the application is stored in a single
- object. State is Read-Only: The only way to change the state is to emit an action, an object describing what happened.
- Changes are made with Pure Functions: Reducers are pure functions that take the previous state and an action, and return the next state.

# What are actions in Redux, explain with example?

Actions are plain JavaScript objects that describe what happened in the application. They must have a type property that indicates the type of action being performed.

```
// actions.js
export const increment = () => ({ type: 'INCREMENT' });
export const decrement = () => ({ type: 'DECREMENT' });
```



# Explain reducers in Redux with an example

Reducers are pure functions that take the current state and an action, and return a new state based on the action type.

```
const initialState = { count: 0 };

const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
};
```

# What is the role of the Redux store?

The store holds the whole state tree of the application. It allows access to the state via `getState()`, dispatching actions via `dispatch(action)`, and registering listeners via `subscribe(listener)`.

```
import { createStore } from 'redux';
|
const initialState = { count: 0 };

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

const store = createStore(counterReducer);

console.log(store.getState()); // { count: 0 }
store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // { count: 1 }
```

# How do you connect React components to Redux store using connect?

The connect function connects a React component to the Redux store. It maps state and dispatch to the component's props.

```
import React from 'react';
import { connect } from 'react-redux';
import { increment } from './actions';

const Counter = ({ count, increment }) => (
  <div>
    <p>{count}</p>
    <button onClick={increment}>Increment</button>
  </div>
);

const mapStateToProps = (state) => ({ count: state.count });
const mapDispatchToProps = { increment };

export default connect(mapStateToProps, mapDispatchToProps)(Counter);
|
```



# How do you use the useSelector and useDispatch hooks in a functional React component?

useSelector is used to access the Redux state, and useDispatch is used to dispatch actions in functional components.

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

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

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

export default Counter;
```



# What is Redux Toolkit?

Redux Toolkit is an official, opinionated toolset for efficient Redux development. It simplifies store setup, reduces boilerplate, and includes useful tools like `createSlice` and `createAsyncThunk`.

# How to configure store in redux toolkit ?

Redux Toolkit is an official, opinionated toolset for efficient Redux development. It simplifies store setup, reduces boilerplate, and includes useful tools like `createSlice` and `createAsyncThunk`.

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

const store = configureStore({
  reducer: {
    //pass all reducers
  },
});

export default store
```

# Explain createSlice in Redux Toolkit with an example.

createSlice is a function that generates action creators and action types, and creates a reducer based on an object of "slice" reducers.

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

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

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

# What are controlled components in React?

Controlled components are React components where the form data is handled by the React state. The input's value is always driven by the React state.

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

const ControlledInput = () => {
  const [value, setValue] = useState('');

  return (
    <div>
      <input type="text" value={value}
        onChange={e => setValue(e.target.value)}
      />
      <p>Value: {value}</p>
    </div>
  );
};

export default ControlledInput;
```



# What are uncontrolled components in React?

Uncontrolled components are React components where the form data is handled by the DOM itself. The input's value is not driven by the React state.

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

const UncontrolledInput = () => {
  const inputRef = useRef(null);

  const handleSubmit = e => {
    e.preventDefault();
    console.log(inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
};

export default UncontrolledInput;
```

# How do you optimize performance in React applications?

- Using `useMemo` and `useCallback` to memoize expensive calculations and functions.
- Implementing `shouldComponentUpdate` or using `React.memo` for Pure Components.
- Code splitting and lazy loading.

# What is code splitting in React?

Code splitting is a feature supported by React that allows you to split your code into various bundles which can then be loaded on demand.

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

const Home = lazy(() => import('./Home'));
const About = lazy(() => import('./About'));

const App = () => {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
        </Switch>
      </Suspense>
    </Router>
  );
};

export default App;
```

# What are render props in React? Give an example.

Render props are a technique for sharing code between React components using a prop whose value is a function. This function returns a React element and is used by the component to

```
import React from 'react';

const MyComponent = ({ renderProp }) => {
  return (
    <div>
      <h1>My Component</h1>
      {renderProp()}
    </div>
  );
};

const App = () => {
  return (
    <MyComponent
      renderProp={() => <p>This is a render prop!</p>}
    />
  );
};

export default App;
```



# What are portals in React?

Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is useful for things like modals, tooltips, and overlays.

```
import ReactDOM from 'react-dom';
import React, { useState } from 'react';

const Modal = ({ isOpen, children }) => {
  if (!isOpen) return null;
  return ReactDOM.createPortal(
    <div>
      <h1>Modal</h1>
      {children}
    </div>,
    document.body
  );
};

const App = () => {
  const [isOpen, setIsOpen] = useState(false);
  return (
    <div>
      <button onClick={() => setIsOpen(true)}>Open Modal</button>
      <Modal isOpen={isOpen}>
        <p>This is a modal!</p>
        <button onClick={() => setIsOpen(false)}>Close</button>
      </Modal>
    </div>
  );
};

export default App;
```

# How do you implement lazy loading in React?

Lazy loading in React can be implemented using the `React.lazy` and `Suspense` components. This allows you to load components on demand, improving initial load times.

```
import React, { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

const App = () => {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
};

export default App;
```

# How do you define props for a functional component in TypeScript?

```
import React from 'react';

interface Props {
  title: string;
  count: number;
}

const MyComponent: React.FC<Props> = ({ title, count }) => {
  return (
    <div>
      <h1>{title}</h1>
      <p>Count: {count}</p>
    </div>
  );
};

export default MyComponent;
```

# How do you use the useState hook with TypeScript?

We can define the type of the state variable by specifying it in the useState generic.

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

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

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

export default App;
```



# How do you type event handlers in React with TypeScript?

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

const App = () => {
  const [value, setValue] = useState<string>('');

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    setValue(e.target.value);
  };

  return (
    <div>
      <input type="text" value={value} onChange={handleChange} />
      <p>Value: {value}</p>
    </div>
  );
};

export default App;
```

# How do you handle optional props in React components with TypeScript?

In TypeScript, you can handle optional props by using the ? operator in the props interface or type alias.

```
import React from 'react';

interface Props {
  title: string;
  subtitle?: string;
}

const MyComponent: React.FC<Props> = ({ title, subtitle }) => {
  return (
    <div>
      <h1>{title}</h1>
      {subtitle && <p>{subtitle}</p>}
    </div>
  );
};

export default MyComponent;
```

# How do you use the useReducer hook with TypeScript?

```
app / index.tsx / State
import React, { useReducer } from 'react';

interface State {
  count: number;
}

interface Action {
  type: 'increment' | 'decrement';
}

const initialState: State = { count: 0 };
const reducer = (state: State, action: Action): State => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

const App = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  > return ( ...
  );
};

export default App;
```



# How do you type the context API in React with TypeScript?

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

interface ContextValue {
  count: number;
  increment: () => void;
}

const CountContext = createContext<ContextValue>({
  count: 0,
  increment: () => {},
});

const CountProvider: React.FC<{children: React.ReactNode}> = ({ children }) => {
  const [count, setCount] = useState<number>(0);

  const increment = () => setCount(count + 1);

  return (
    <CountContext.Provider value={{ count, increment }}>
      {children}
    </CountContext.Provider>
  );
};
```

```
const CountDisplay: React.FC = () => {
  const { count } = useContext(CountContext);

  return <h1>Count: {count}</h1>;
};

const App = () => {
  return (
    <CountProvider>
      <CountDisplay />
    </CountProvider>
  );
};

export default App;
```



# How do you write a simple test in Jest?

Jest is a JavaScript testing framework maintained by Facebook. It is commonly used with React because it provides a simple and powerful testing solution with features like snapshot testing, coverage reporting, and built-in assertions.



```
test('adds 1 + 2 to equal 3', () => {  
  expect(1 + 2).toBe(3);  
});
```

# How do you render a component for testing using React Testing Library?

```
import { render } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders MyComponent', () => {
  render(<MyComponent />);
});
```

# How can you find elements in the DOM using React Testing Library?

```
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

test('finds an element by text', () => {
  render(<MyComponent />);
  const element = screen.getByText(/my text/i);
  expect(element).toBeInTheDocument();
});
```



# How do you simulate user events in React Testing Library?

```
import { render, screen, fireEvent } from '@testing-library/react';
import MyComponent from './MyComponent';

test('increments counter on click', () => {
  render(<MyComponent />);
  const button = screen.getByRole('button', { name: /increment/i });
  fireEvent.click(button);
  expect(screen.getByText(/count: 1/i)).toBeInTheDocument();
});
```



# How can you test component props with React Testing Library?

```
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

test('passes props correctly', () => {
  render(<MyComponent title="Test Title" />);
  expect(screen.getByText(/test title/i)).toBeInTheDocument();
});
```

# Create a Controlled Input Component

Build a controlled component called `TextInput` that renders an input field and a button. When the button is clicked, an alert should display the current input value.

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

function TextInput() {
  const [value, setValue] = useState('');

  const handleChange = (e) => setValue(e.target.value);

  const handleClick = () => alert(`Current input: ${value}`);

  return (
    <div>
      <input type="text" value={value} onChange={handleChange} />
      <button onClick={handleClick}>Show Input</button>
    </div>
  );
}

export default TextInput;
```

# Implement toggle Visibility of a Component

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

function ChildComponent() {
  return <div>Child Component</div>;
}

function ToggleComponent() {
  const [isVisible, setIsVisible] = useState(true);

  return (
    <div>
      <button onClick={() => setIsVisible(!isVisible)}>
        {isVisible ? 'Hide' : 'Show'} Component
      </button>
      {isVisible && <ChildComponent />}
    </div>
  );
}

export default ToggleComponent;
```



# Fetch Data from an API and Display it , along with loading state

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

function DataFetcher() {
  const [data, setData] = useState(null);
  //implement loading logic

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

  return <div>{data ? data.title : 'Loading...'}</div>;
}

export default DataFetcher;
```



# Create a Reusable Button Component with Props

```
import React from "react";

function Button({ text, onClick }) {
  return <button onClick={onClick}>{text}</button>;
}

export default function App() {
  return <Button
    text={"Login"}
    onClick={() => console.log("clicked")}
  />;
}
```

# Build a Component that Uses an Effect to Perform Cleanup.

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

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

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

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

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

export default Timer;
```

# Implement a Context with a Reducer for Global State Management

```
import React, { createContext, useReducer, useContext } from 'react';
const CounterContext = createContext();
const initialState = { count: 0 };

function counterReducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error(`Unhandled action type: ${action.type}`);
  }
}

export function CounterProvider({ children }) {
  const [state, dispatch] = useReducer(counterReducer, initialState);
  return (
    <CounterContext.Provider value={{ state, dispatch }}>
      {children}
    </CounterContext.Provider>
  );
}

export function useCounter() {
  const context = useContext(CounterContext);
  if (!context) {
    throw new Error('useCounter must be used within a CounterProvider');
  }
  return context;
}
```



# Build a Component with Conditional Rendering Based on Props.

```
import React from 'react';

function StatusMessage({ status }) {
  let message;
  switch (status) {
    case 'loading':
      message = 'Loading...';
      break;
    case 'success':
      message = 'Data loaded successfully!';
      break;
    case 'error':
      message = 'Error loading data.';
      break;
    default:
      message = 'Unknown status.';
      break;
  }

  return <p>{message}</p>;
}

export default StatusMessage;
```



# Implement a simple form component

```
app / index.jsx / LoginForm
import React, { useState } from 'react';

function LoginForm() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Username: ${username}, Password: ${password}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
        placeholder="Username"
      />
      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        placeholder="Password"
      />
      <button type="submit">Submit</button>
    </form>
  );
}

export default LoginForm;
```