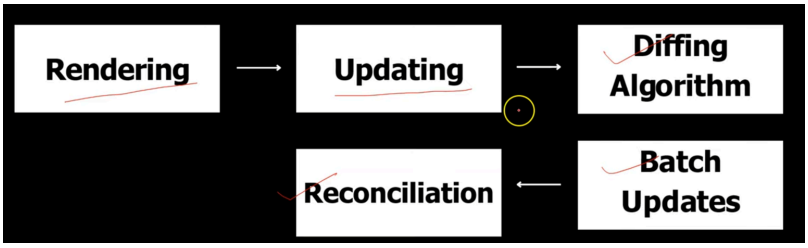
ReactJs

React is an open-source JS library. **It's major features are**:

- Virtual DOM: React uses virtual DOM to improve performance by minimizing direct DOM manipulations.
- JSX: It allows writing HTML content in React components
- Components: React is component-based, meaning the UI is built using reusable components

1. **WHAT IS VIRTUAL DOM?**

- Virtual DOM is a lightweight representation of the real DOM elements generated by React components. React keeps the copy of the actual DOM structure in memory called the virtual DOM and it uses this to optimize updates and rendering. It uses an algorithm known as `Diffing Algorithm` which helps in making this happen.
- The reason why Virtual DOM was introduced because DOM manipulation is quite slow when compared to other operations in JavaScript. The efficiency of the application gets affected when several DOM manipulations are being done.



- 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.
  How it happens ?:
- When a component's **state or props change**, React re-runs the component to generate a **new Virtual DOM tree**. React then compares this new tree with the **previous Virtual DOM snapshot** using its diffing algorithm.
  - If the element type is the same (`<div>` to `<div>`), React reuses the existing DOM node and only updates the changed attributes.

    - If the type is different (`<div>` to `<p>`), React destroys the old node and mounts a new one.
- Once differences are identified, React batches the required changes and applies only those updates to the **real DOM**, instead of re-rendering the entire UI. This process is called **reconciliation**, and it ensures minimal, efficient DOM mutations.

2. **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. We have 2 types in this:

- Class Based Components
- Functional Components

3. **EXPLAIN CLASS COMPONENTS WITH EXAMPLE?**

---

ex: `import React, { Component } from 'react'` class ClassComponentExample extends Component{ render(){ return

Hello

4. **EXPLAIN FUNCTIONAL COMPONENTS WITH EXAMPLE? WHY DID WE NEED FUNCTIONAL COMPONENTS?**

Functional components are basically just JavaScript functions that return JSX — which is how we write UI in React. They’re the simplest way to create components.
ex:

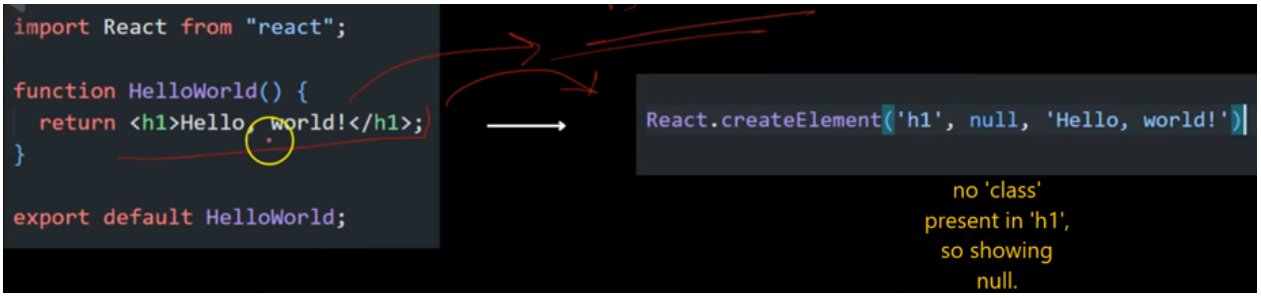
```
import React from 'react'
const FunctionalComponent = () = {
```

```
    return <h1> Hello </h1>
}
export default FunctionalComponent
```

Earlier, class components were used when we needed features like state or lifecycle methods. But with the introduction of **Hooks** (like `useState`, `useEffect`), functional components became much more powerful — now they can do everything class components can, but with less code and better readability.

5. **WHAT IS JSX?**

- JSX stands for JavaScript XML (eXtensible Markup Language)
- It allows us the write HTML elements in JavaScript and place them in DOM without using it's methods (Optional: like createElement() or appendChild()) For example:



6. **HOW TO EXPORT AND IMPORT COMPONENTS?**

We can export components using `export default` or `named exports`, and import them using `import`. For example:

```
import React from "react"
const Home = () => { return <h1> Home </h1> }
export default Home;
```

7. **HOW TO USE NESTED COMPONENTS?**

---

menu.jsx:

```
import React from "react";
function Menus(){
    return <div> ... </div>
}
function Header{
    return (
        <h1> Hi </h1>
        <Menus/>
    )
}
```

app.jsx:

```
import React from "react"
import Header from './Header'
function App(){
    return(
        <Header />
        <h1> Main </h1>
    )
}
```

8. **WHAT IS STATE IN REACT? WHY YOU SHOULD NOT UPDATE THE STATE DIRECTLY? HOW TO UPDATE THE STATE?**

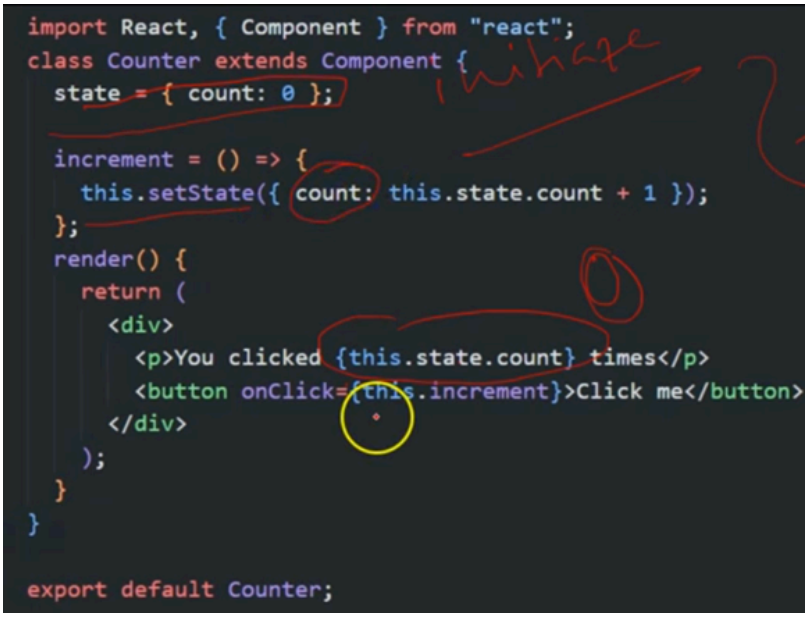
- In React, **state is an object that represents the parts of the application that can change**. Each component can maintain its own state, and React uses that state to decide what should be rendered in the UI.
- When we update the state using `setState` (in class components) or the `useState` hook (in functional components), React is notified of the change. React then schedules a re-render, creates a new Virtual DOM, runs the reconciliation process, and finally updates only the parts of the real DOM that actually changed. This ensures the UI stays in sync with the latest state.
- If we update the state directly, like `state = state + 1`, React has **no idea** that the state changed. Since React's internal mechanism wasn't triggered, it won't re-render the component, leading to **UI inconsistencies**. That's why we should always update state using `setState` or the `useState` setter —

because they don't just change the value, they also tell React *when* to re-render.

- Here is how to update the state:

```
import React, {useState} from 'react'
function Counter(){
    const [count, setCount] = useState(0);
    function handleClick(){
        setCount(count+1)
    }
}

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

export default Counter
```

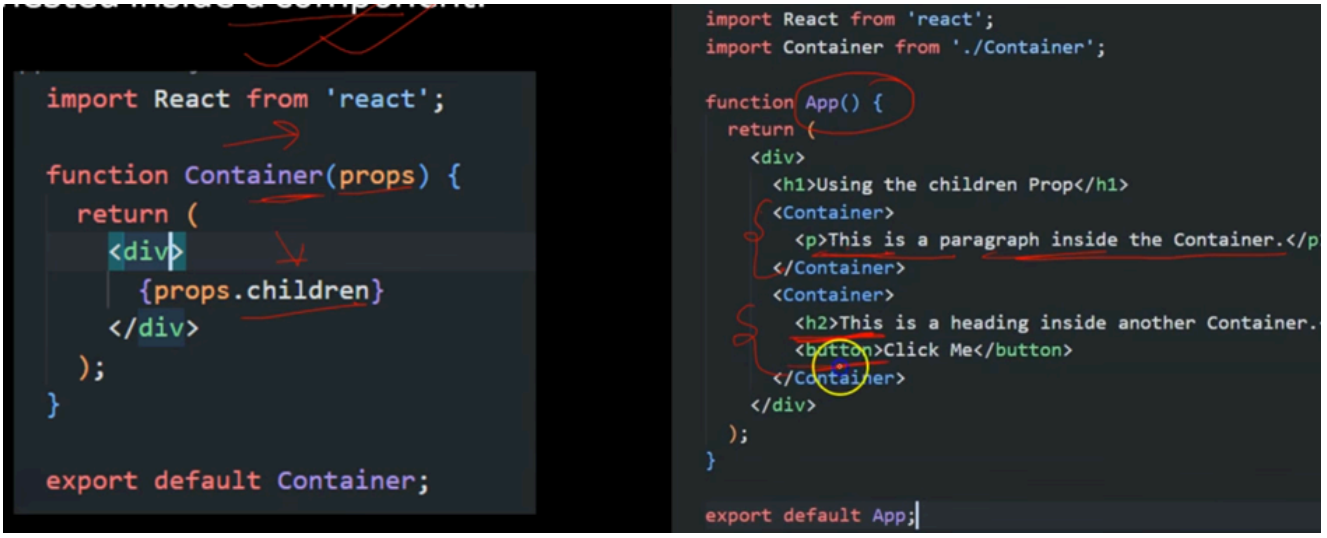
For class-based component:



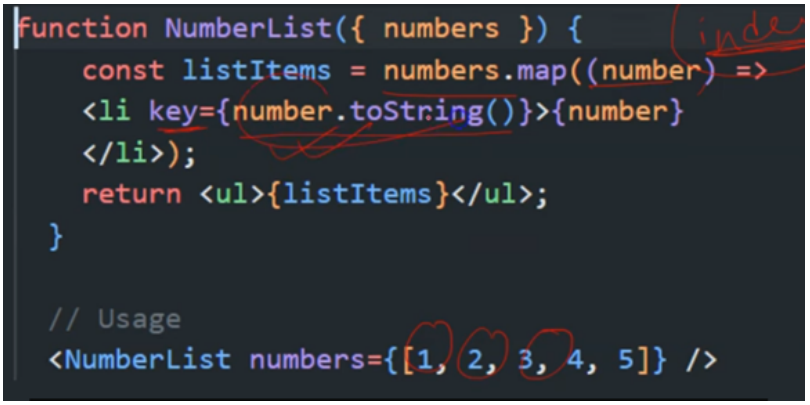
- **setState callback:**
  - The setState method can accept a callback function as the second argument which is executed once the state has been updated and the component has re-rendered.

9. **WHAT ARE PROPS IN REACT**

- Props are used to pass data and event handlers to children component.
- (Optional: Event handlers are functions that run in response to user actions — like clicks, typing, form submissions, etc. They're how we make a web page interactive.)
- Props ensure one-way data flow i.e. from parent to children.
- Props cannot be modified by the child component that receives them
- Children Props: It is a special property in React which is used to pass the content that is nested inside a component.



- Key Prop: It is a unique identifier for each element in the list used by React to identify the items that have been changed/added or removed. Using indexes as keys is not recommended as it can lead to performance issues and unexpected behavior when list items are reordered or removed. That's why keys should be unique.



- defaultProps: It 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!"
```

10. **WHAT IS THE DIFFERENCE BETWEEN STATE AND PROPS?**

State:

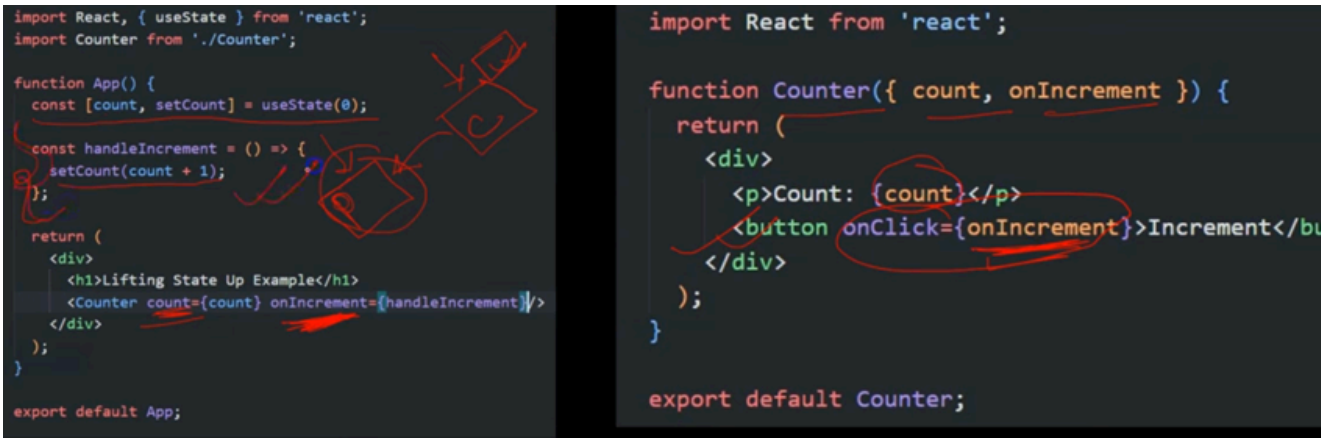
- State is a built-in object used to store data that can change over the lifecycle of a component.
- It is **managed within the component itself** and is **mutable**, meaning it can be updated using `setState` (class components) or `useState` (functional components).
- State is **local to the component**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

  Props:
- Props (short for *properties*) are used to **pass data from a parent component to a child component**.
- They are **read-only** and **immutable** inside the child component, meaning the child cannot modify them.
- Props allow parent components to configure or customize child components by supplying values.

11. **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.
  For example:



12. **WHAT ARE FRAGMENTS IN REACT AND ITS ADVANTAGES?**

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

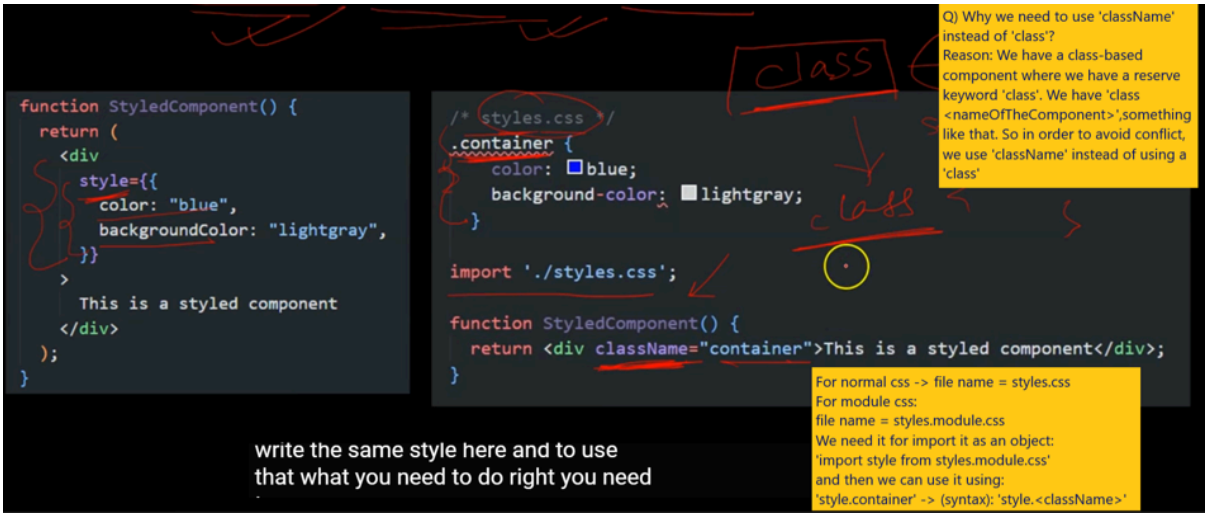
```
import React from "react"
function List(){
    return(
        <React.Fragment>
            <li> Hi </li>
        </React.Fragment>
    )
}

// Short syntax
function List(){
    return(
        <>
            <li> Hi </li>
        </>
    )
}
```

- **Difference between `<div>` and a Fragment?**
  - A `<div>` actually creates a **real DOM element**, while a Fragment (`<>` `</>`) is only a **React wrapper** and does not appear in the DOM.

- Using `<div>` may lead to unnecessary DOM nesting (extra layers), while Fragments avoid that.
- Use `<div>` when you need styling, attributes, or semantic meaning.
- Use Fragments when you just need to group children without affecting the DOM structure.

13. HOW TO USE STYLING IN REACTJS?

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



14. HOW CAN YOU CONDITIONALLY RENDER COMPONENTS IN REACT?

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

```
function Greeting({ isLoggedIn, flag }){
    if(isLoggedIn){
        return <p> Done </p>
    } else {
        return flag ? <p> SignUp </p> : <p> SignIn </p>
    }
}

// Usage
<Greeting isLoggedIn = {true} flag = {false} />
```

15. HOW DO YOU RENDER LIST OF DATA IN REACT?

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

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

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

16. WHY USIND INDEXES AS KEYS ARE NOT RECOMMENDED?

Using indexes as keys in React is not recommended because it can cause unexpected behavior when list items are added, removed, or reordered. React relies on keys to identify which elements have changed, and if we use indexes, React may mistakenly reuse DOM elements for different items. This can lead to UI bugs and performance issues. Keys should always be unique and stable, such as an `id`. For example, imagine we render a todo list:

- ❌ **Problem when using index as key:**

```
function TodoList() {
  const [todos, setTodos] = React.useState([
    { id: 1, task: "Learn React" },
    { id: 2, task: "Practice DSA" },
    { id: 3, task: "Build Projects" }
  ]);

  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>
          <input defaultValue={todo.task} />
        </li>
      ))}
      <button onClick={() => setTodos(todos.slice(1))}>
        Remove First
```

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

- Initially you see three inputs:
  - Learn React
  - Practice DSA
  - Build Projects
- Now click **Remove First** → we remove `"Learn React"`.
- React reuses the DOM nodes based on `index`, so:
  - The 2nd item `"Practice DSA"` shifts into index `0` and takes over the old input field.
  - The input field still contains the text `"Learn React"` because React reused the DOM element incorrectly.
- Result → UI shows wrong values (mismatch between state and DOM).

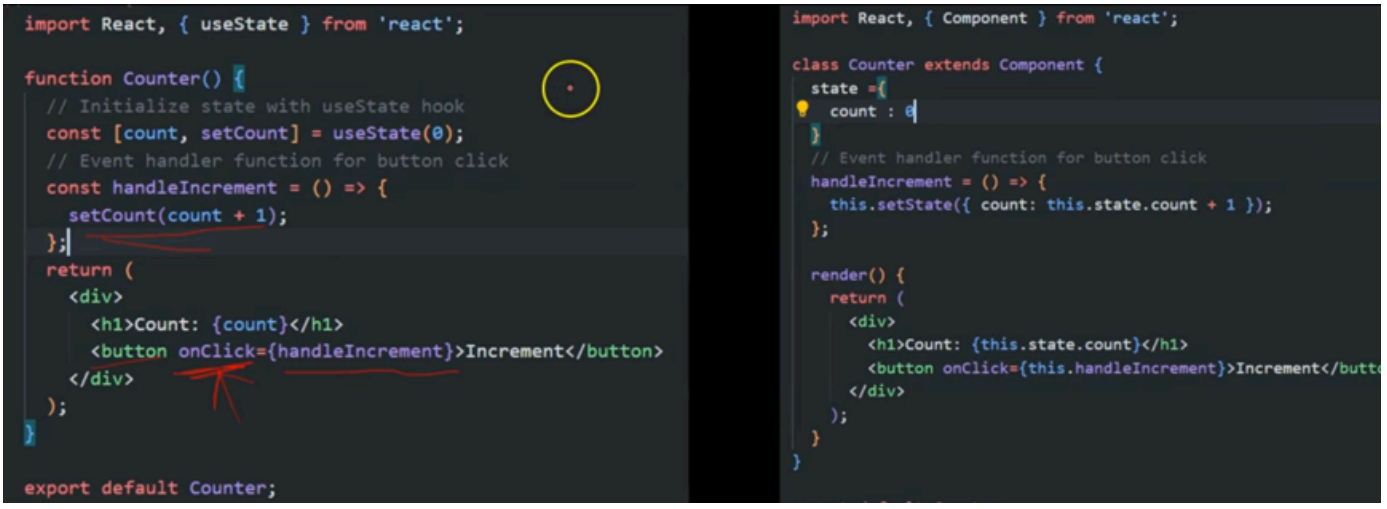
- ✅ **Correct way with unique key (id):**

```
function TodoList() {
  const [todos, setTodos] = React.useState([
    { id: 1, task: "Learn React" },
    { id: 2, task: "Practice DSA" },
    { id: 3, task: "Build Projects" }
  ]);

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <input defaultValue={todo.task} />
        </li>
      ))}
      <button onClick={() => setTodos(todos.slice(1))}>
        Remove First
      </button>
    </ul>
  );
}
```

- Same initial UI as before.
- When `"Learn React"` is removed, React knows exactly which item was removed (because of the unique `id` key).

- It **removes only that input** and keeps `"Practice DSA"` and `"Build Projects"` intact.
- Result → UI stays consistent with the state.

17. HOW TO HANDLE BUTTONSI N REACT?



18. HOW TO HANDLE INPUTS IN REACT?

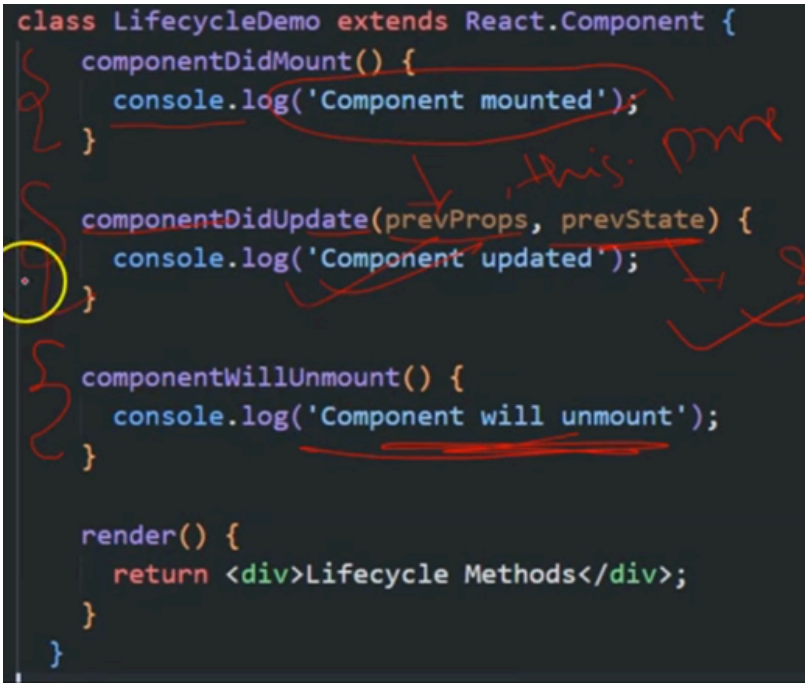
We can use controlled components where form data is handled by the component's state.
ex:

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

19. 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 the component is being created and inserted into the DOM.
- Updating : when the component is being re-rendered due to changes in state or props.
- Unmounting : when the component is being removed from the DOM.



20. 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

21. WHAT IS `useState` HOOK AND HOW TO MANAGE IT?

`useState` is a React Hook which helps in managing the state in functional components.
ex:

```
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>
    )
}
export default Counter
```

22. WHAT IS `useEffect` HOOK AND HOW TO MANAGE SIDE EFFECTS?

`useEffect` is a React Hook that manages side effects like data fetching or manually changing the DOM.
ex:

```
import React, { useEffect } from 'react'
const DataFetcher = () => {
    useEffect(() => {

    }, [dependency]) // Runs if dependency value changes. If empty
array, means this effect runs only once.

    return (
        <div>
        </div>
    )
}
export default DataFetcher
```

23. IMPLEMENT DATA FETCHING AND MANAGE LOADING STATE IN REACT?

---

ex:

```
import React, { useState, useEffect } from 'react'
const DataFetchingExample = () => {
    const [data, setData] = useState(null)
    const [loading, setLoading] = useState(false)

    const fetchData = async () => {
        setLoading(true)
        const response = await fetch('/api-endpoint')
        const result = await response.json()
        if(result){
            setData(true)
            setLoading(false)
        }
    }

    if(loading) return <h1> Loading data... </h1>

    useEffect(() => {
        fetchData()
    }, [])

    return (
        <div>
            {
                data ? data.title : "loading..."
            }
        </div>
    )
}
export default DataFetchingExample
```

24. WHAT IS PROP DRILLING AND HOW TO AVOID IT?

Prop drilling occurs when we have to pass the data through many layers of components. It can be avoided using the Context API or state management libraries like Redux which helps in making the code more readable and maintainable. Example of prop drilling:

```
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>
}
```

25. **WHAT IS CONTEXT API IN REACT?**

- 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.
- ex:

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

// Create a context
const MyContext = createContext()

// Provider Component
const MyProvider = ({ children }) => {
    const testMessage = "Hello world"
    return (
        <MyContext.Provider value={testMessage} >
            { children }
        </MyContext.Provider>
    )
} // children -> All the child components
export default MyProvider
```

```
const App = () => {
    return <MyProvider> <div> Child components </div> </MyProvider>
}
export default App
```

- Here, `value={testMessage}` means "all child components wrapped in this provider will have access to this value".
- How to consume the context using the `useContext hook?`
  ex:

```
import React, { useContext } from 'react'
import { MyContext } from './MyProvider'
const MyComponent = () => {
    const value = useContext(MyContext)
    return <div> {value} </div>
}
export default MyComponent
```

- How can we update the context values?
  - We can just make a `useState` variable and then pass it to the `value` inside `MyContext.Provider` as `value == {{ value, setValue }}`. Later we can consume the context (for ex: `const {value, setValue} useContext(MyContext)`) and update the context values.
- How do you use multiple contexts in a single component?

```jsx
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>
  );
}
```

- Advantages 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. For ex:
  - ```
    // Without Context API (Prop drilling)
    <Parent> // The Parent component has some data (value) and
    it needs to pass it to GrandChild.
      <Child>
          <GrandChild value = {value} />
      </Child>
    </Parent>

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

```
  <GrandChild />
</MyProvider>
```

26. WHAT IS `useReducer` HOOK AND WHEN TO USE IT?

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

- **When you say complex state logic, what do you mean specifically? Why can't we use `useState` then?**
  - So, `useState` is great when your component has simple state; like toggling a boolean, updating an input field, or switching tabs.
  - But when I say **complex state logic**, I mean cases where:
  - We have **multiple pieces of related state**, and
  - Those pieces of state are updated in **dependent or conditional ways**, or
  - Those updates involve more than just setting a new value; like resetting based on different actions.
  - That's where `useReducer` helps us.
  Let me give you a quick example: Suppose I'm building a **cart system**. Using `useState`, I'd have to manage:
- Items array
- Total price
- Discounts
- Maybe loading states or error messages
  All with separate `useState` calls. And then I'd need to carefully coordinate updates between them. It can get messy.
  But with `useReducer`, I can handle everything in **one reducer function**, where all logic is centralized. Based on the action type, I can update the state in a structured way. It's like having a mini-Redux inside my component. Cleaner, easier to maintain, and more scalable if the logic grows.
  ex:

```jsx
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>
  );
}
```

imported 'useRedu
from 'react', which
take a state and a
dispatch method.
'dispatch' will disp
the action with a 't
property. This 'typ
property is used to
identify different
and whenever the
a case which will
satisfy, it will retur
the updated state.
nothing satisfies, y
can return an erro
return the state.

ex: (for complex state management)

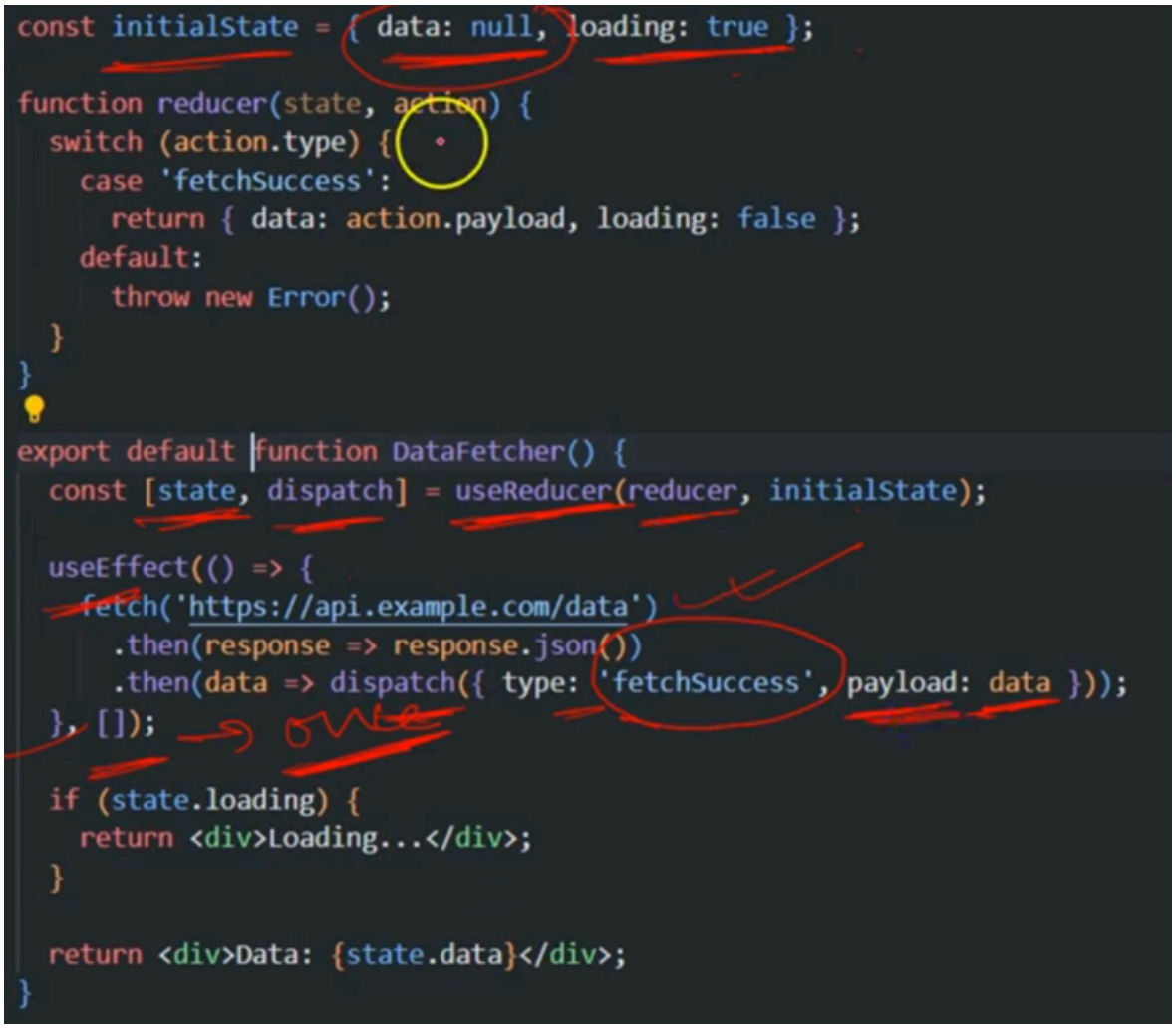
```jsx
const initialState = { count: 0, text: '' };

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();
  }
}
```

definitely yes because this is why why

- **Passing additional arguments to the reducer function:**

```jsx
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 } });
```

- **Handling side effects with `useReducer`:**

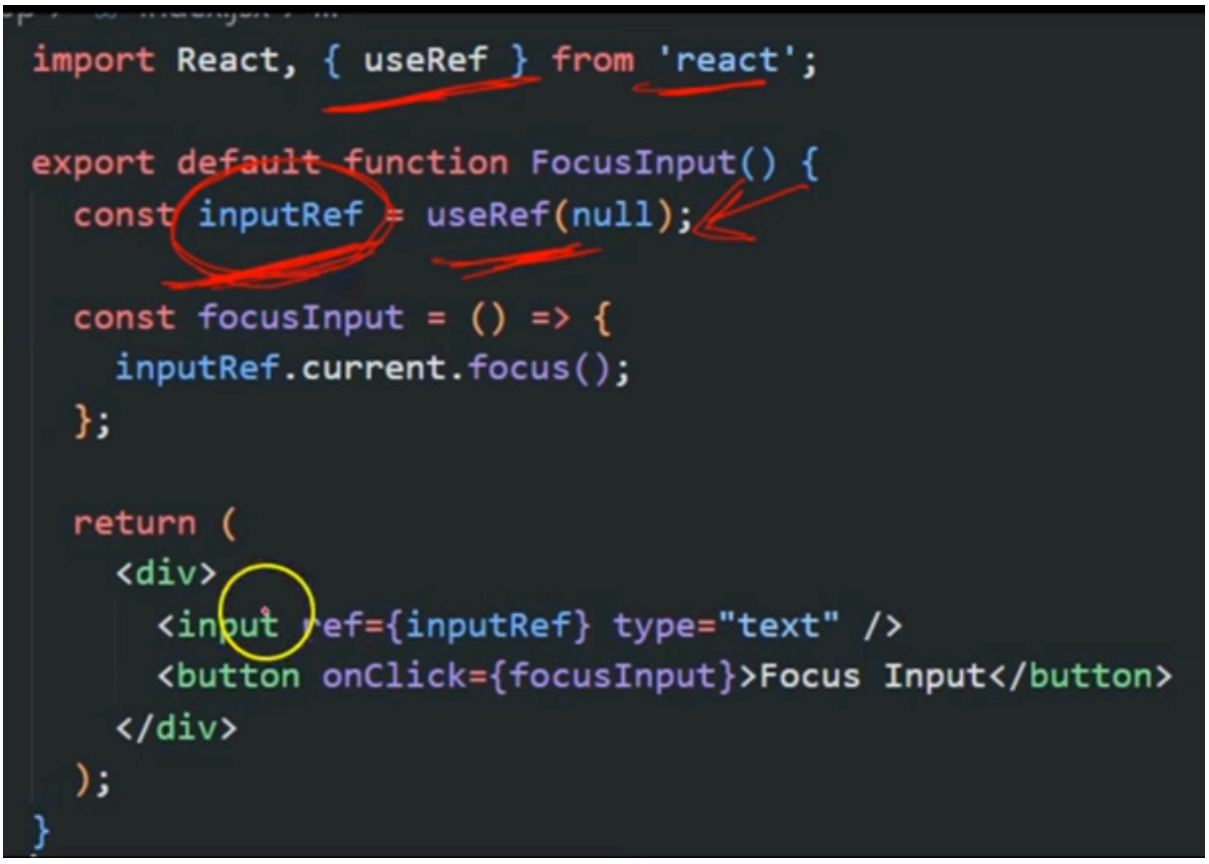
- 

```
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>;
}
```

27. **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 any re-renders.

ex:



- The `useRef` is initialized with `null` since the DOM element doesn't exists until the component renders.
- We use the `.current` function to store and update any mutable change.
- So, we were typing a text in the input field
- We clicked the button
- Now the focus shifts to the input field and we can see a blinking cursor that is ready to receive keyboard input.

28. **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. For example:

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

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

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

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

29. **HOW TO MANAGE FORMS IN REACT?**

Forms in react can be manages using controlled components where form data is handled by the component's state. For example:

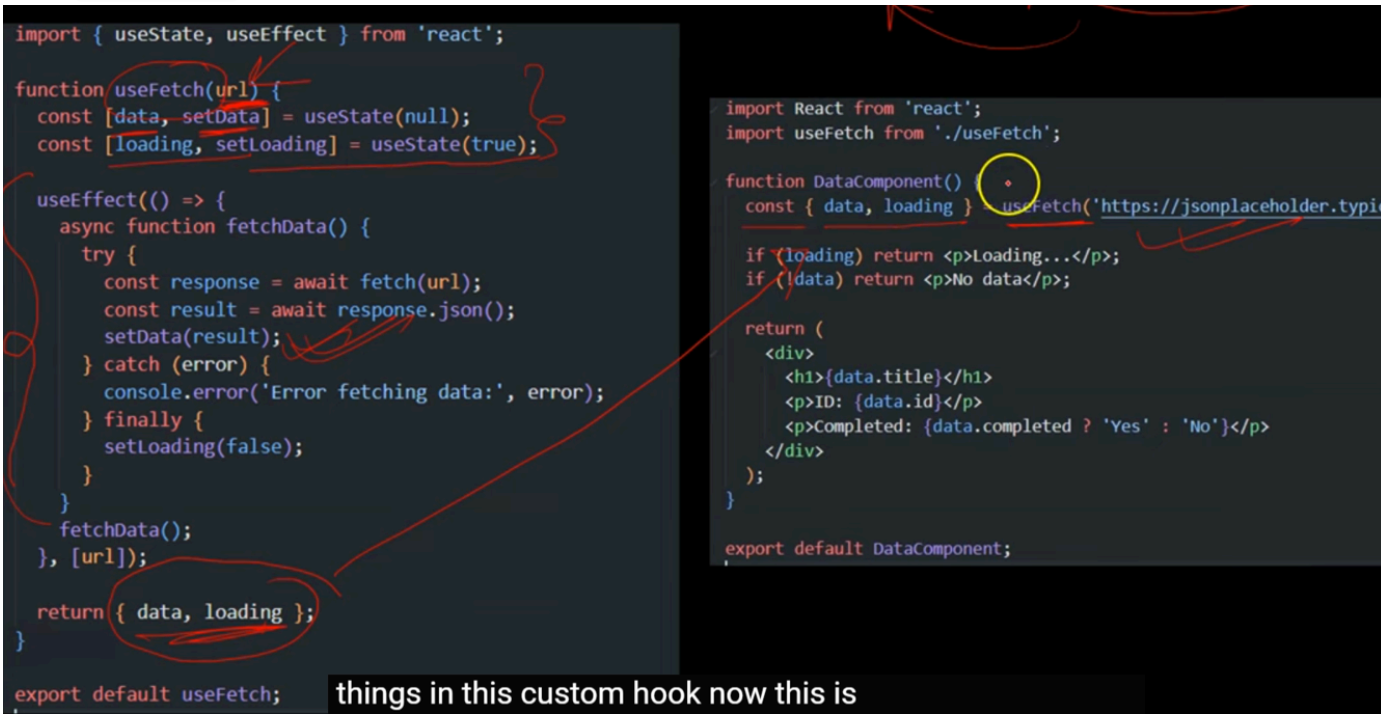
```jsx
import React, { useState } from 'react';

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;
```

event.preventDefault() : on clicking submit, our page will get refreshed, so to basically prevent that event, we are using the above keyword

30. **WHAT ARE CUSTOM HOOKS AND WHY DO WE NEED THEM?**

Custom Hooks in React are Javascript functions that allows us to reuse stateful logic across multiple components. They enables us the extract and share common logic without repeating the code, promoting code reusability and separation of concerns.

ex: (`useFetch` Custom Hook)



31. WHAT IS REACT ROUTER DOM AND HOW IS IT USED?

React Router DOM is a routing library built on top of React Router. It enables dynamic routing in web applications, allowing us to define routes and navigate between different components without reloading the page.
ex: (basic routing using react router dom)

```
import { BrowserRouter, Routes, Route } from 'react-router-dom'
const App = () => {
    return (
        <BrowserRouter>
            <Routes> // 'Routes' is used to define set of routes
where only first matching route is rendered.
                <Route path='/home' element = {<Home />} />
                <Route path='/login' element = {<Login />} />
                <Route path='*' element = {<NotFound />} /> //
Handles 404 error (not found). It catches all unmatched routes
            </Routes>
        </BrowserRouter>
    )
}
export default App
```

32. HOW TO CREATE A LINK TO ANOTHER ROUTE USING REACT ROUTER DOM?

Use the Link Component to create navigation links. For example:

```
import { Link } from 'react-router-dom'
const Navigation = () => {
    <nav>
        <Link to="/home"> Home </Link>
    </nav>
}
export default Navigation
```

33. HOW DO YOU USE URL PARAMETERS/ DYNAMIC ROUTING IN REACT ROUTER DOM?

---

ex:

```
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/>} />
```

34. HOW CAN YOU PERFORM A REDIRECT IN REACT ROUTER DOM?

- Use the Navigate component to perform a redirect.
- `<Navigate/>` : Works when the component re-renders and the condition is met.
  For example:

```
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;
```

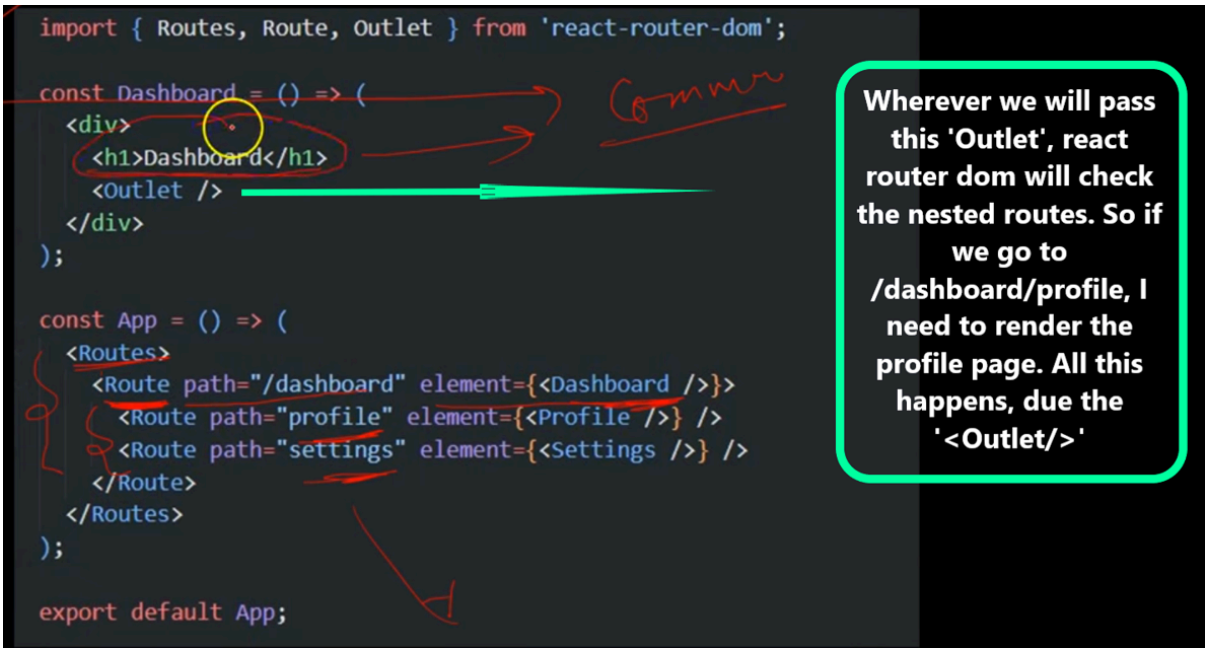
- To redirect programmatically i.e. You decide *when* to call `navigate()`, we can use `useNavigate()`
  For example:

```
import { useNavigate } from 'react-router-dom'

const App = () => {
    const navigate = useNavigate();
    const handleLogin = async () => {
      const success = await loginUser();
      if (success) {
        navigate("/dashboard"); // only after login success
      }
    }
    return <button onClick={handleLogin}> Log in </button>
};
```

35. **HOW DO YOU HANDLE NESTED ROUTES IN REACT?**

---

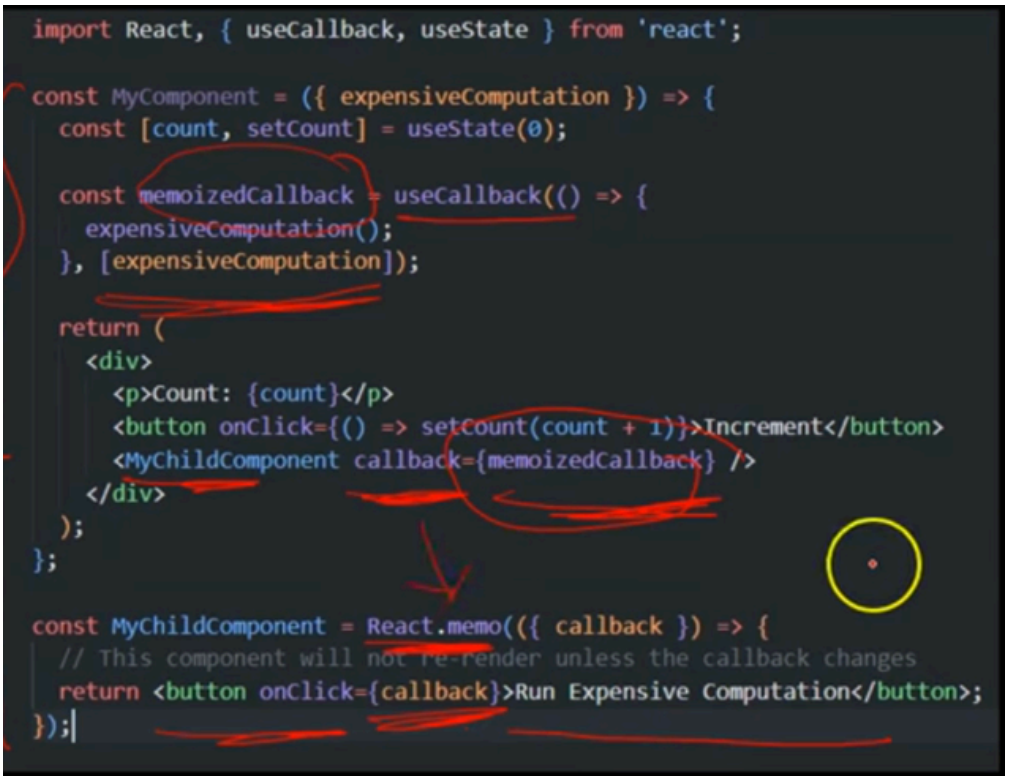
ex:



36. 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 it's dependencies has changed. This is useful when we are passing callbacks to child components that are optimized with React.memo, as it can prevent unnecessary renders.
- React.memo is a higher-order component that memoizes the result of the component. It prevents the component from re-rendering unless the props have changed.

ex:

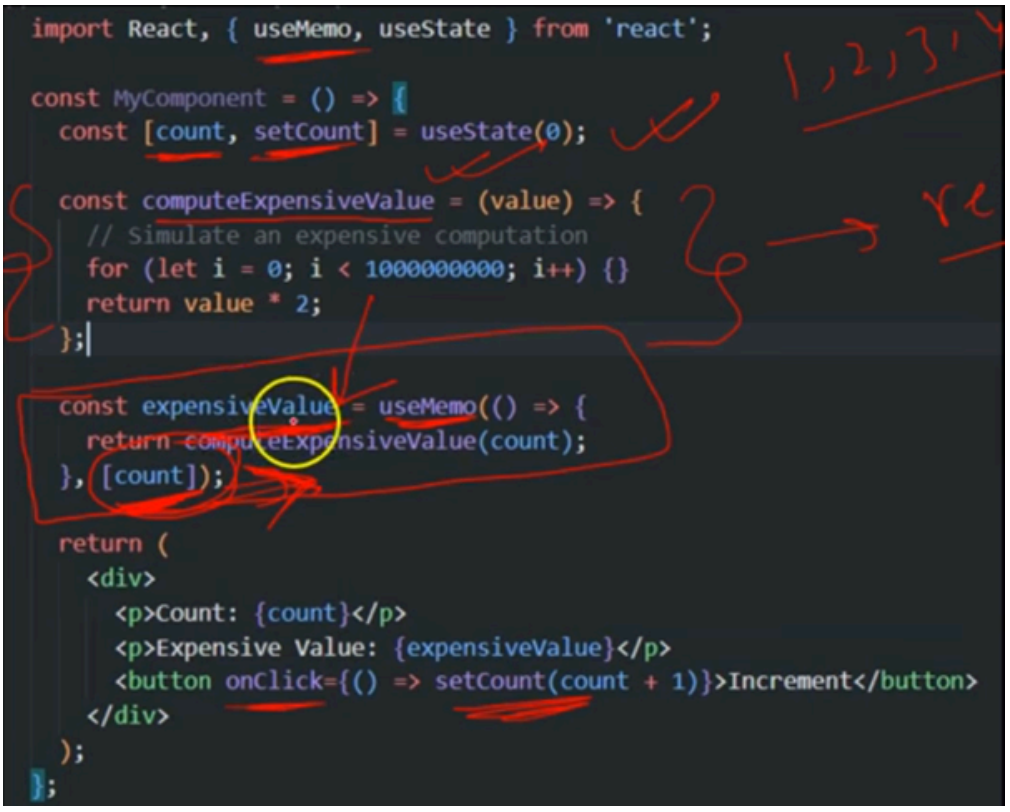


```jsx
import React, { useCallback, useState } from 'react';

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>;
});
```

37. **EXPLAIN `useMemo` HOOK WITH EXAMPLE?**

- The `useMemo` Hook is used to memoize expensive calculation so that they are not recalculated on every render.
- It is similar to `useCallback` hook, but there is a slight difference in both of them:
  - `useCallback` is used to memoize call function, on the other hand, `useMemo` is used to memoize a value and in both cases we need to pass a dependency array. Whenever there is change in dependency, the values will be

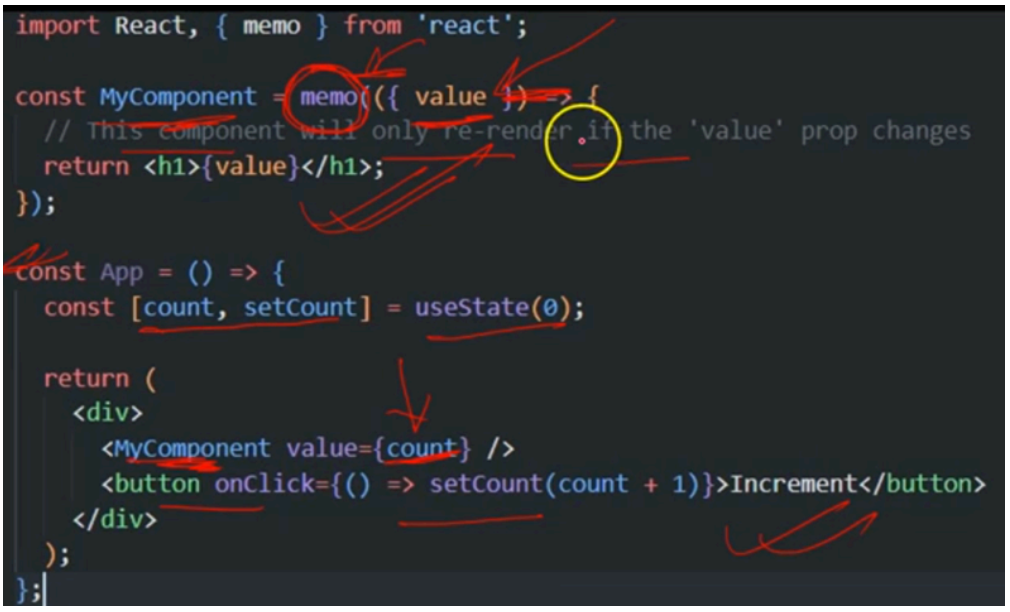
recalculated.



```jsx
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>
  );
};
```

38. EXPLAIN `React.memo` WITH EXAMPLE

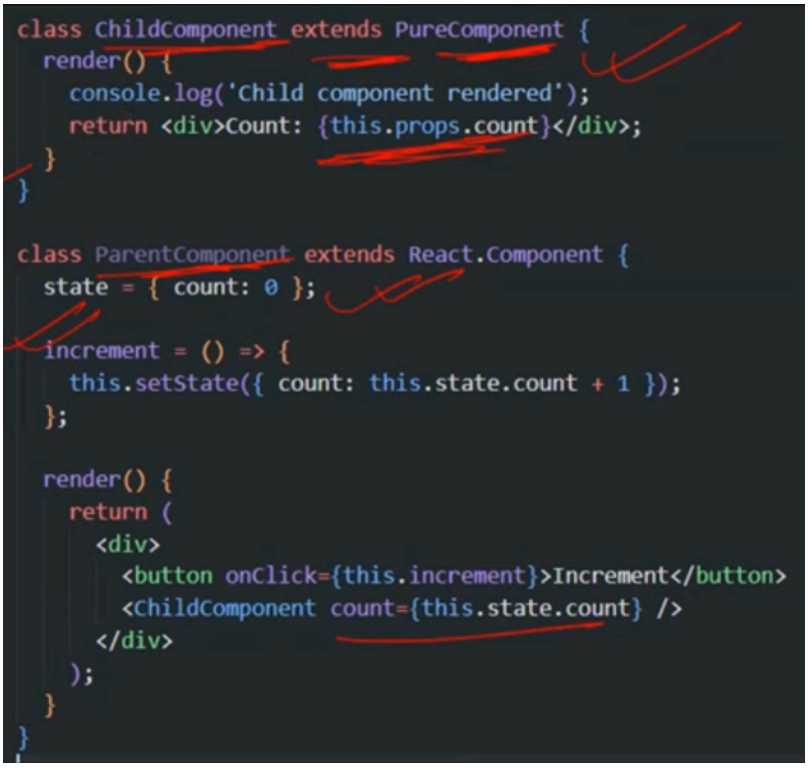
React.memo is a higher-order component that memoizes the result of a component. It prevents the component from re-rendering unless the props have changed. This is useful for optimizing performance by avoiding unnecessary renders of pure components. For example:



```jsx
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>
  );
};
```

39. WHAT ARE PURE COMPONENTS?

- A **PureComponent** in React is a special type of class component that helps optimize performance by preventing unnecessary re-renders.
- Unlike a regular component, a PureComponent automatically implements `shouldComponentUpdate()` with a **shallow comparison** of both props and state.
- Shallow comparison means React only checks whether the top-level values of props or state have changed, rather than doing a deep comparison of nested objects. For example, if a primitive value like a number, string, or boolean changes, the component will re-render, but if an object reference remains the same even though its internal properties are updated, the shallow comparison will not detect the change.
- If nothing has changed at this top level, the component will skip re-rendering, which improves performance by avoiding unnecessary renders. For example:



```
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>
    );
  }
}
```

- For functional based components, we can use `memo` and for class based components, we can use `pure`.
- Note: A **regular class component** (extending `React.Component`) will **always re-render** whenever its parent re-renders, even if the props and state are the same. It doesn't perform any check to see if the data has actually changed.

40. **EXPLAIN HIGHER ORDER COMPONENTS WITH EXAMPLE?**

A Higher Order Component is a function that takes a component and returns a new component with added functionality. They are used for reusing component

logic and enhancing components with additional behavior. For example:

```
import React from "react";

// HOC function
const withLogger = (WrappedComponent) => {
  return (props) => {
    console.log("Props received:", props);
    return <WrappedComponent {...props} />;
  };
};

// Normal Component
const Hello = ({ name }) => {
  return <h2>Hello, {name} 👋</h2>;
};

// Wrap Hello with HOC
const HelloWithLogger = withLogger(Hello);

export default function App() {
  return <HelloWithLogger name="Ankit" />;
}
```

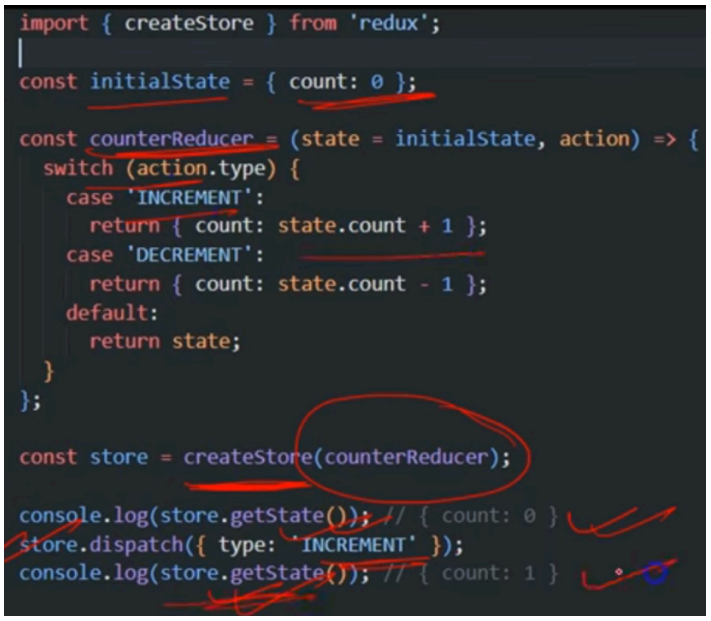
- `withLogger` is a **HOC** → it takes `Hello` as input.
- It returns a new component that **logs the props** before rendering.
- When you use `<HelloWithLogger name="Ankit" />`, it prints props in console and then renders `Hello`.

41. **WHAT IS REDUX? EXPLAIN CORE PRINCIPLES?**

Redux is a state management library for Javascript applications . It acts as a centralized store for state management in an application. It's core principles are:

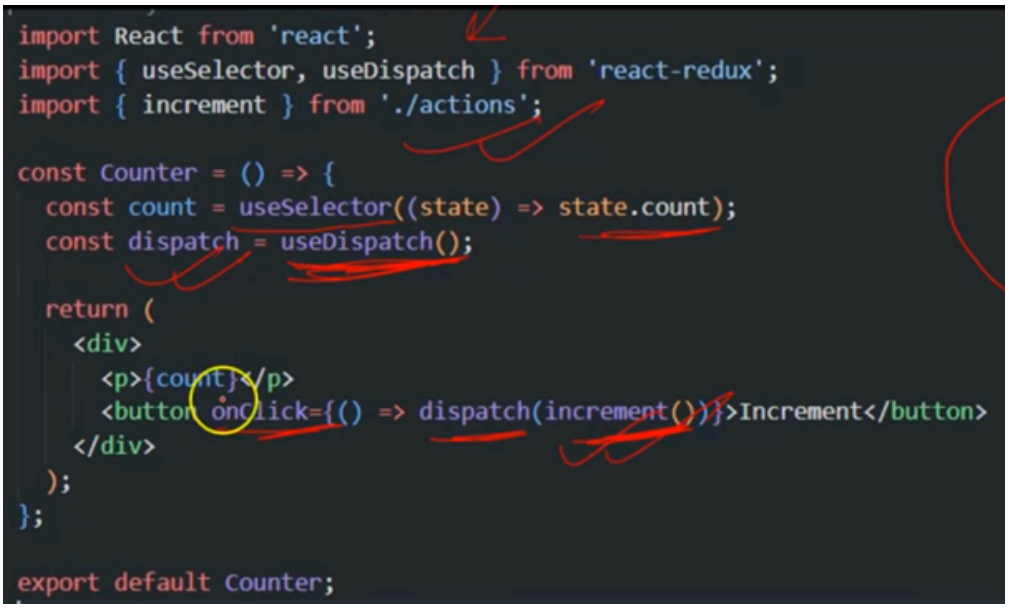
- *Single Source of Truth:* The state of the application is stored in a single object.
- (Optional: State is an object that represents the parts of the application that can change)
- *State is Read-Only:* The only way to change the state is to emit an action, an object describing what happened.
- What are `actions` in Redux?

    - Actions are plain JS objects that describe what happened in the application. They must have a `type` property that indicates the type of action being performed.
    - ex: `export const increment = () => ({ type: 'INCREMENT' })`
- What are `reducers` in Redux?
    - Reducers are pure functions that take the previous state and an action, and return the next state.
    - *A Pure function is a function that always returns the same output for the same input and does not cause any side effects.*
- What is the role of 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)`.
    - ex:



42. **HOW DO YOU USE `useSelector` & `useDispatch` HOOKS IN A FUNCTIONAL REACT COMPONENT?**

- `useSelector` is used to bring the updated value from the Redux store to the React component.
- `useDispatch` is used to dispatch an action (like if a button is pressed, increase count) towards Redux store.



```jsx
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;
```

43. WHAT IS REDUX TOOLKIT?

Redux Toolkit is the **official, recommended way** to write Redux code because it makes working with Redux much simpler and less error-prone. Normally, setting up Redux requires writing a lot of boilerplate code (actions, reducers, types, etc.), but Redux Toolkit reduces that by providing built-in utilities like `createSlice`, which lets you define state, actions, and reducers all in one place. In short, Redux Toolkit helps developers build scalable and maintainable state management with less code and better readability.

- **Configuring store in redux toolkit:**

```
import { configureStore } from '@reduxjs/toolkit'
const store = configureStore({
    reducer:{
        // pass all reducers
    },
})
export default store
```

44. EXPLAIN `createSlice` IN REDUX TOOLKIT WITH EXAMPLE?

- `createSlice` in Redux Toolkit is a function that makes managing state much easier by combining **state, actions, and reducers** into a single place. Instead of writing separate action types, action creators, and reducers, you can define

everything inside a slice. Each slice corresponds to a "slice of the global Redux state."

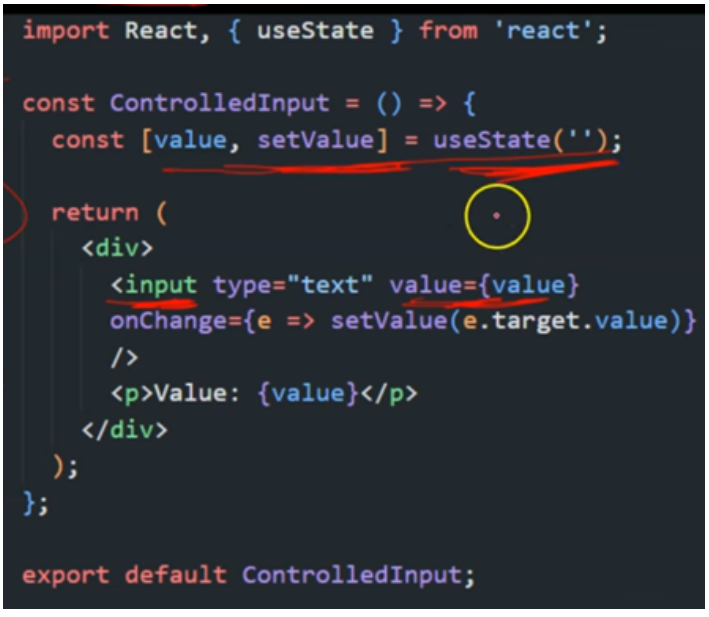
- When you create a slice, Redux Toolkit automatically generates action creators and a reducer for you, based on the logic you define.
  ex (of counter slice):

```
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
```

- You only **write the minimal reducer logic**, and Redux Toolkit takes care of **action creators, action types, and the final reducer wiring**.

45. **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. For example:



46. **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. For example:

```jsx
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;
```

47. HOW DO YOU OPTMIZE PERFORMANCE IN REACT APPLICATIONS?

- Using `useMemo` and `useCallback` to memoize expensive calculations and functions.
- Using `React.memo` for Pure Components.
- By using lazy loading and code splitting.

48. WHAT IS CODE SPLITTING IN REACT?

- So, code splitting in React basically means breaking our app into smaller bundles instead of shipping the entire JavaScript code in one big file.
- Why are we splitting the code into various bundles?
  - When we write a React app (or any modern JS app), we usually write the code across multiple files and components. But browsers don't really know how to handle all that modular stuff directly. So before our app runs in the browser, a **build tool** like Vite **combines** all our JavaScript, CSS, and other assets into one or more big files — called **bundles**, that can

become super heavy and affect performance, especially the initial load time.

- Now, the idea behind code splitting is simple: only load the code that's actually needed at the moment. For example, if I'm on the homepage, why should the code for the dashboard or the settings page be loaded? We can split that into separate bundles and load them **on demand**. (Optional: Which is exactly what tools like React.lazy and dynamic `import()` let us do.)
  ex:

```
import React, { lazy, Suspense } from 'react'
import { BrowserRouter, Route, Switch } from 'react-router-dom'
const Home = lazy(() => import ('./Home'))
const About = lazy(() => import ('./About'))
const App = () => {
    return (
        <BrowserRouter>
            <Suspense fallback={ <div> Loading... </div> } >
                <Switch>
                    <Route exact path='/' component={Home} />
                    <Route path='/about' component={About} />
                </Switch>
            </Suspense>
        </BrowserRouter>
    )
}
export default App
```

49. **WHAT ARE RENDER PROPS IN REACT? GIVE AN EXAMPLE?**

- So, **Render props** in React is a pattern where **we pass a function as a prop** to a component, and **that function tells the component what UI to render**.
- It's like "Hey component, you handle the logic, and I'll give you a function that decides what to show on the screen."
- ex:

```
    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
```

50. **WHAT ARE PORTALS IN REACT?**

In React, **Portals** provide a way to render a component's children into a different part of the DOM tree, outside of the parent component's DOM hierarchy. Normally, components render inside their parent, but with portals, you can render them anywhere in the DOM while still keeping their React component relationship intact. This is useful for things like modals, tooltips and overlays. Ex:

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

const Modal = ({ children }) => {
  return ReactDOM.createPortal(
    <div className="modal">
      {children}
    </div>,
    document.getElementById("modal-root")  // Target DOM node
  );
};

export default function App() {
  return (
    <div>
      <h1>Normal App Content</h1>
      <Modal>
        <p>This is inside a portal!</p>
      </Modal>
```

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

- In this example, we are creating a `Modal` component using React Portals. Normally, when we place `<Modal>` inside `<App>`, React would render its children inside the same DOM tree under `#root`. But since we used `ReactDOM.createPortal`, the modal's content is rendered into a completely different DOM node, `#modal-root`, which is defined separately in `index.html`. This means that although in our JSX it looks like the modal is nested inside `<App>`, in the actual DOM structure it is "teleported" outside the parent hierarchy. The big advantage of this is that the modal is no longer affected by parent styles like `overflow: hidden` or `z-index`, making it ideal for components such as dialogs, dropdowns, or tooltips. At the same time, React still treats the portal as part of the component tree, so event bubbling and state management continue to work just like normal.

51. **WHAT IS LAZY LOADING?**

- Lazy loading is basically a technique where components or resources are loaded **only when they are needed**, not before that.
- How do we implement Lazy Loading?:
  - In React, we use `React.lazy()` along with `Suspense` to achieve this.
  - ex:

```
const Profile = React.lazy(() => import(./Profile))
<Suspense fallback={<div> Loading... </div>}
    <Profile />
</Suspense>
```

  - Without lazy loading, the `Profile` component would be included in the **main bundle**, even if the user never visits the page where it's used. That makes the initial load slower.
  - But with `React.lazy()`, `Profile` gets its **own separate bundle**, which is only downloaded **on-demand** — i.e., when the user navigates to the part of the app where `<Profile />` is actually needed.

52. **DIFFERENCE BETWEEN TRANSPILER AND COMPILER IN REACT?**

Both transpilers and compilers are tools that convert code from one form to another, but they have different purposes.

- **Transpiler:**
  - A **transpiler** converts code from **one version of a language to another version of the same language**.
  - In React, we mostly use **Babel** as a transpiler. (Optional: `Vite` uses `esbuild` as it's transpiler.)
  - It takes modern JavaScript (like ES6+) or JSX and converts it to **older JavaScript** that browsers can understand.
    ex: `const greet = () => console.log("hi");`
    gets transpiled to:

```
var greet = function () {
    return console.log("hi");
};
```

  - This is what Babel does. So browsers don't freak out when they see arrow functions or JSX.
- **Compiler:**
  - A **compiler** usually means converting **high-level code** into **machine code or a lower-level language**.
  - For React apps, tools like **Vite** do this job. They compile our code (Optional: along with assets like CSS, images, etc.), make it optimized and prepare that code for execution.
  - **Vite uses ESBuild** under the hood for lightning-fast compilation during development. It compiles the code on-demand, meaning when the browser requests a module, it compiles and serves just that, instead of bundling everything upfront like Webpack used to do. In production, Vite switches to **Rollup** from **ESBuild**.

53. HOW BROWSER LOADS JSX ELEMENT IN FRONTEND ?

So JSX isn't something browsers understand directly. Browsers can only read regular JavaScript, HTML, and CSS.

- So when I write something like:
  `<h1>Hello World</h1>`

- Babel or whatever transpiler I'm using will convert this into:
  `React.createElement("h1", null, "Hello World")`
  This `React.createElement()` returns a **plain JavaScript object** — this object is what we call part of the **virtual DOM**. It's literally a JS representation of what we want the UI to look like. Now, React keeps this virtual DOM in memory. But the browser doesn't care about that. The browser only deals with the **actual DOM** (which are real HTML elements on the page). This is where rendering engine of React comes i.e. ReactDOM that converts that virtual DOM into actual DOM nodes and mounts them into the browser's DOM. So basically,
- We write JSX → React turns it into virtual DOM (JS objects).
- Then **ReactDOM** walks through that virtual DOM and builds real DOM nodes.
- Those real DOM nodes are finally mounted (added) into the browser's page. So at no point is the browser directly reading or executing JSX—it's always JavaScript by the time it reaches the browser.

54. **What's the difference between Vite and React? Why do we use Vite?**

- **React and Vite are very different things.** React is a **JavaScript library for building user interfaces**; it focuses only on the UI part, letting us create components, manage state, and handle rendering. But React by itself doesn't provide tooling for bundling, running a dev server, or optimizing code.
- That's where **Vite** comes in. Vite is a **modern build tool and dev server** that we use alongside React. In the past, React apps usually relied on Webpack, which bundled the entire project up front. The problem was that even small changes forced Webpack to re-bundle everything, which slowed down development.
- Vite solves this by leveraging **native ES modules** in the browser. Instead of bundling everything at the start, Vite serves files on demand and reloads only the part of the code that changed. This makes the dev server start almost instantly and updates appear immediately with **Hot Module Replacement (HMR)**. For production, Vite still bundles the app efficiently using Rollup.
- So to summarize: **React builds the UI, while Vite makes building and running that UI much faster and more efficient.** React is the "what," and Vite is the "how."

Practical

55. CREATE A CONTROLLED INPUT COMPONENT

```
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
```

56. 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 = {() => setItVisible(!isVisible)}>
                {isVisible ? 'Hide' : 'Show'} Component
            </button>
            {
                isVisible && <ChildComponent />
            }
```

```
        </div>
    )
}

export default ToggleComponent
```

57. **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)
    const [loading, setLoading] = useState(true)
    const [error, setError] = useState(null)

    useEffect(() => {
        const fetch = async () => {
            try{
                const response = await
fetch('https://jsonplaceholder.typicode.com/posts/1')
                if(!response.ok){
                    throw New Error('Network response was not ok')
                }
                const result = await response.json()
                setData(result)
            } catch(err) {
                setError(err.message)
            } finally {
                setLoading(false)
            }
        }

        fetchData()
    }, [])

    if(loading) return <div> Loading... </div>
    if(error) return <div> Error: {error}</div>

    return <div> data?.title </div>
}
```

```
export default DataFetcher
```

58. **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!")}
    />
}
```

59. **BUILD A COMPONENT THAT USES AN EFFECT TO PERFORM CLEANUP**

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

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

    useEffect(() => {
        const interval = setInterval(() => {
            setSeconds((prevSeconds) => prevSeconds + 1)
        }, 1000) // Runs every 1second = 1000ms

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

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

- `setTimeout` : - Runs **once** after a given delay. For example:

```
setTimeout(() => {
  console.log("Runs once after 2 seconds")
}, 2000) // Output: "Runs once after 2 seconds" → only once.
```

- `setInterval` : Runs **repeatedly** at the given interval (until stopped). For example:

```
const id = setInterval(() => {
  console.log("Runs every 2 seconds")
}, 2000)

// Stop after 7 seconds
setTimeout(() => clearInterval(id), 7000)

// Output:
- After 2s → `"Runs every 2 seconds"`
- After 4s → `"Runs every 2 seconds"`
- After 6s → `"Runs every 2 seconds"`
- Then it stops at 7s.
```

60. **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
}
```

- **Example Usage:**

```
function CounterButtons() {
  const { state, dispatch } = useCounter()

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}> +
</button>
      <button onClick={() => dispatch({ type: 'decrement' })}> -
</button>
    </div>
  )
}

// In App.jsx
function App() {
  return (
    <counterProvider>
      <CounterButtons />
    </counterProvider>
```

```
  )
}
```

61. 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
```

62. IMPLEMENT A SIMPLE FORM COMPONENT

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

export default 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>
    )
}
```