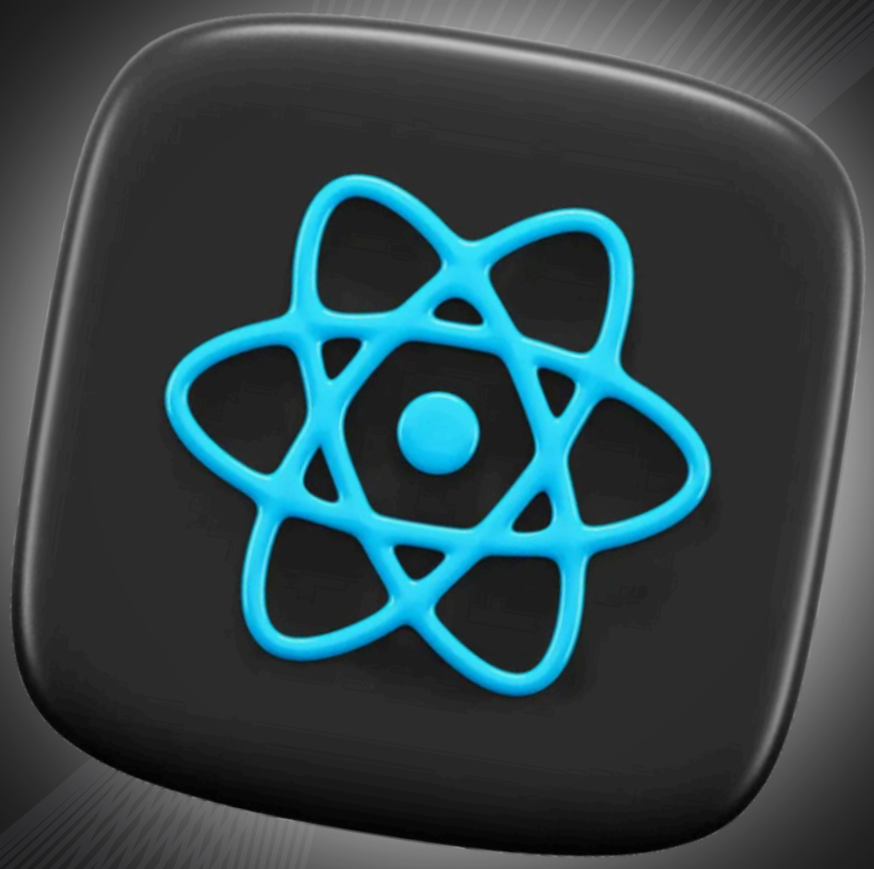
SHEKHOBABA.COM

Most Useful React Hooks

useState( )
useEffect( )
useContext( )
useReducer( )
useRef( )
useMemo( )
useCallback( )



Burhan Tahir
@iamshekhobaba

useState

- What It Does: useState adds state to functional components, managing data that changes over time.

- Why It's Important: It makes components dynamic, allowing them to react to user actions and changes.

- Example in Practice: Imagine a form where a user can type their name. You need to store that name somewhere, and useState provides a way to do that.

Burhan Tahir
@iamshekhobaba

useState

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

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

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

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

Burhan Tahir

@iamshekhobaba

useEffect

- What It Does: useEffect handles side effects like data fetching, DOM updates, and cleanup in functional components.

- Why It's Important: It enables functional components to perform essential tasks like interacting with APIs.

- Example in Practice: If you need to fetch user data from an API when the component first loads, useEffect is the perfect tool.

Burhan Tahir
@iamshekhobaba

useEffect

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

function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://api.example.com/user/${userId}`)
      .then(response => response.json())
      .then(data => setUser(data));

    return () => console.log('Cleanup if needed');
  }, [userId]);

  return (
    <div>
      {user ? <p>User name: {user.name}</p> : <p>Loading ...</p>}
    </div>
  );
}
```

Burhan Tahir

@iamshekhobaba

useContext

- What It Does: useContext allows components to access shared values from a context provider without passing props manually.

- Why It's Important: It simplifies sharing state across multiple components, avoiding prop drilling.

- Example in Practice: Consider a theme setting that needs to be accessible by many components. Instead of passing the theme prop through every component, you can use useContext.

Burhan Tahir
@iamshekhobaba

useContext

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

const ThemeContext = React.createContext('light');

function ThemedButton() {
  const theme = useContext(ThemeContext); // Access the current theme from context

  return <button className={`btn-${theme}`}>Themed Button</button>;
}

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton /> {/* All children of the provider can access the context */}
    </ThemeContext.Provider>
  );
}
```

Burhan Tahir

@iamshekhobaba

useReducer

- What It Does: useReducer manages complex state logic by dispatching actions to a reducer function, offering more control than useState.

- Why It's Important: It's ideal for handling complex state transitions, especially when the next state depends on the previous one.

- Example in Practice: Managing a shopping cart with actions like adding, removing, or updating items is a good use case for useReducer.

Burhan Tahir
@iamshekhobaba

useReducer

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

const initialState = { cart: [] };

function reducer(state, action) {
  switch (action.type) {
    case 'add':
      return { cart: [...state.cart, action.item] };
    case 'remove':
      return { cart: state.cart.filter(item => item.id !== action.id) };
    default:
      return state;
  }
}

function ShoppingCart() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <ul>
        {state.cart.map(item => (
          <li key={item.id}>
            {item.name} <button onClick={() => dispatch({ type: 'remove', id: item.id
})}>Remove</button>
          </li>
        ))}
      </ul>
      <button onClick={() => dispatch({ type: 'add', item: { id: 1, name: 'New Item' }
})}>Add Item</button>
    </div>
  );
}
```

Burhan Tahir

@iamshekhobaba

useRef

- What It Does: useRef gives you a way to persist values across renders without causing a re-render when the value changes. It also provides access to DOM elements.

- Why It's Important: Sometimes, you need to maintain a mutable value or directly interact with a DOM element (like focusing an input), and useRef is the best tool for that.

- Example in Practice: Automatically focusing an input field when a component mounts.

Burhan Tahir
@iamshekhobaba

useRef

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

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

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} type="text" />;
}
```

Burhan Tahir

@iamshekhobaba

useMemo

- What It Does: useMemo is used to memoize (cache) the result of an expensive computation, so it only recalculates when necessary (when dependencies change).

- Why It's Important: It helps in optimizing performance by avoiding unnecessary re-computations, which can be critical in large or complex applications.

- Example in Practice: Avoiding recalculating a filtered list every time a component renders.

Burhan Tahir
@iamshekhobaba

useMemo

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

function FilteredList({ items, filter }) {
  const filteredItems = useMemo(() => {
    console.log('Filtering items');
    return items.filter(item => item.includes(filter));
  }, [items, filter]);

  return (
    <ul>
      {filteredItems.map((item, index) => <li key={index}>{item}
</li>)}
    </ul>
  );
}
```

Burhan Tahir

@iamshekhobaba

useCallback

- What It Does: useCallback returns a memoized version of a callback function, which helps prevent unnecessary re-renders of components that rely on that function, especially when passing callbacks as props to child components

- Why It's Important: It keeps function references stable, avoiding re-renders in child components when passing callbacks.

- Example in Practice: Use useCallback when a parent component passes a function to a child that shouldn't re-render unless the function changes.

Burhan Tahir
@iamshekhobaba

useCallback

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

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

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

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

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

Burhan Tahir

@iamshekhobaba

SHEKHOBABA.COM

DO YOU WANT MOBILE APP OR WEBSITE FOR YOUR BUSINESS?

FOLLOW ME

FOLLOW ME