Day 5

# React Components Management

kodemaven-portfolio.vercel.app

# Introduction

When you create a component, ReactJS creates a virtual representation of that component in memory. This virtual representation is known as the "**Virtual DOM**".

When the component's state or props change, ReactJS updates the Virtual DOM accordingly.

ReactJS applies the necessary patches to the actual DOM to reflect the changes. This process of updating the Virtual DOM and then applying patches to the actual DOM is called "**Reconciliation**".

Components go through various lifecycle methods, which occurs in similar manner but with different methods, for both class and functional components.

# Class Component Lifecycle Management

In React, a **class componen**t has a well-defined lifecycle, which refers to the series of methods that are automatically called during its creation, updating, and deletion.

These methods give you control over how your component behaves at different points, such as when it's first added to the DOM, when it's re-rendered with new data, or when it's removed.

The lifecycle of a class component can be broadly categorized into **three phases**:

- **Mounting:** When a component is created and inserted into the DOM.
- **Updating:** When a component is re-rendered due to changes in state or props.
- **Unmounting:** When a component is removed from the DOM.

# Key Lifecycle Methods in Class Components

## 1. constructor:

- Initializes the component's state and binds methods.
- Called **once** before the component is mounted.

## 2. ComponentDidMount:

- Executed once **after** the component is inserted into the DOM.
- Useful for side effects like data fetching, subscriptions, or setting timers.

## 3. shouldComponentUpdate:

- Determines whether the component should **re-render**.
- Returns a boolean (true by default).

## 4. componentDidUpdate:

- Called after a component has been **updated**.
- Ideal for responding to prop or state changes, like re-fetching data.

## 5. componentWillUnmount:

- Executed right before a component is **removed** from the DOM.
- Useful for cleaning up resources like removing event listeners or canceling network requests.

# Functional Component Management

Functional components are managed by using **React Hooks** for state management, side effects, context, and other React features.

# React Hooks

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

They are called "**hooks**" because they allow you to "**hook into**" React state and lifecycle methods from functional components.

The important hooks for managing functional components are:

- **useState**
- **useEffect**
- **useContext**
- **useReducer**
- **useCallback**
- **useMemo**
- **useRef**
- **Custom Hook**

# useState

The useState hook allows you to **add state** to a functional component.

It returns an array with two elements:

1. The **current state** value
2. A **function to update** that state.

Syntax:

```
const [state, setState] = useState(initialState);
```

Here, `**state**` is the current state value, and `**setState**` is the function to update the state value.

# Example:

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

function Counter() {

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

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

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

- **“useState(0)”** initializes the state count to 0.
- **“setCount”** is a function to update count.
- Clicking the **"Increment"** button calls increment function, which updates the count.

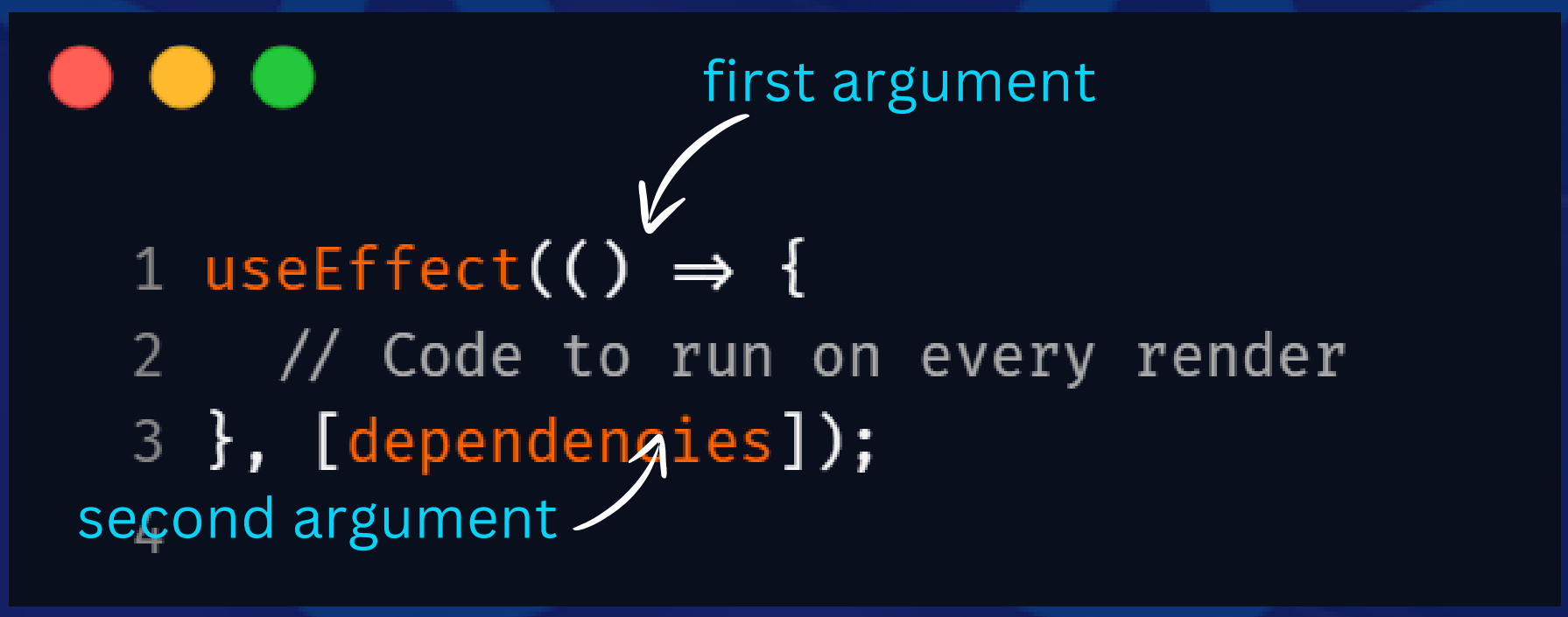
# useEffect

The useEffect hook allows you to perform side effects in function components, like data fetching, subscriptions, or manually changing the DOM.

It takes a function as an argument, which is called after every render.

It runs after every render, but you can control it by specifying dependencies.

Syntax:




Here, the **first argument** is the function to run on every render, and the **second argument** is an array of dependencies.

# Example:

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

function Timer() {
  const [time, setTime] = useState(0);

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

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

  return <div>Time: {time} seconds</div>;
}
```

sets a timer

cleanup function

dependency array

- The **useEffect** hook sets up a timer when the component mounts.
- The empty dependency array **[ ]** makes it run only once after the initial render.
- The **cleanup function** clears the timer when the component is unmounted.

# useContext

The **useContext** hook lets you consume a React **context** directly in a functional component. It avoids the need for nested **Consumer** components.

**Context** is a way to share data between components without having to pass props down manually. You can think of it as a global store that components can access and subscribe to (We will discuss this in our next topic).

Syntax:

```
const value = useContext(MyContext);
```

The **useContext(MyContext)** hook retrieves the nearest context value provided by **MyContext** and assigns it to **value**, enabling direct access to the context data within a component.

# Example:

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

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

const ThemedComponent = () => {
  const theme = useContext(ThemeContext);

  return <div className={`theme-${theme}`}>Current Theme: {theme}</div>;
};

export default ThemedComponent;
```

- This example sets up a context called **ThemeContext** with a default value of '**light**'.
- Inside the **ThemedComponent**, the **useContext** hook accesses this context value, allowing the component to use the theme ('light' in this case).
- The component uses this **theme value** to dynamically apply a class and display the current theme, making it easier to manage theme-related styling and data throughout the app.

# useReducer

The **useReducer** hook is a more powerful alternative to **useState**.

It that allows you to manage state by reducing the state to a single value.

It takes a **reducer function** and **an initial state** as arguments, and returns the **current state** and a dispatch function.

The **reducer function** takes the **current state** and an **action** as arguments, and returns a **new state**. The dispatch function is used to dispatch actions to the reducer.

Syntax:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

# Example:

```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>
  );
}
```

current state

new state for increment

new state for decrement

counter component

dispatch function

In this example,

- The **Counter component** uses the **useReducer** hook to manage the state of the **count**.

- The **reducer function** takes the **current state** and **an action** as arguments, and returns a new state based on the action type.
- The **dispatch function** is used to dispatch actions to the reducer.

The benefits of using useReducer include:

- Easy to manage complex state logic.
- Reducer function can be reused across multiple components.
- Dispatch function can be used to trigger state changes from anywhere in the component tree.

**Note that useReducer is a more advanced hook than useState, and is typically used when you need to manage more complex state logic.**

# useRef

The useRef hook returns a **mutable ref object** that persists for the full lifetime of the component.

It is commonly used to access or manipulate DOM elements or **store mutable values**.

It takes an **initial value** as an argument and returns a **mutable object** with a **current property** that can be used to access the referenced value.

Syntax

```
1 const refContainer = useRef(initialValue);
```

The **useRef** hook creates a **ref container** that takes an **initialValue** which can be any value, including a DOM node, a string, a number, or an object.

The **refContainer** declares a constant variable to store the **ref container**.

# useRef

The useRef hook returns a **mutable ref object** that persists for the full lifetime of the component.

It is commonly used to access or manipulate DOM elements or **store mutable values**.

It takes an **initial value** as an argument and returns a **mutable object** with a **current property** that can be used to access the referenced value.

Syntax

```
1 const refContainer = useRef(initialValue);
```

The **useRef** hook creates a **ref container** that takes an **initialValue** which can be any value, including a DOM node, a string, a number, or an object.

The **refContainer** declares a constant variable to store the **ref container**.

# Example:

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

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

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

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

The **useRef** hook is used to create a reference to an **input** element.

When the button is clicked, the **handleFocus** function is called, which uses the **reference (ref)** to focus the input element.

# useMemo

The **useMemo** hook **memoizes** a value to avoid recomputing it on every render.

It allows you to memoize (**cache**) the result of a function so that it's not recalculated on every render.

It takes a **function** as an argument and returns the **cached** result.

Syntax

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

- **computeExpensiveValue(a, b)** is the function that performs an expensive calculation.
- **[a, b]** is the dependency array, which tells React to recalculate the memoized value only when a or b changes.
- **memoizedValue** is the cached result of the function, which is returned by useMemo.

# Example:

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

function ExpensiveCalculation({ num }) {
  const calculate = (num) => {
    console.log('Computing ... ');
    return num * 2;
  };

  const result = useMemo(() => calculate(num), [num]);

  return <div>Result: {result}</div>;
}
```

- The **ExpensiveCalculation** component receives a **num** prop.
- The **calculate function** performs an expensive calculation.
- The useMemo hook is used to cache the result of the calculate function, so it's only recalculated when the num prop changes.
- The dependency array **[num]** tells React to recalculate the memoized value **only** when num changes.
- The **result variable** holds the cached result of the calculate function

# useCallback

Works like the useMemo hook, but instead of memoizing values, it memoizes function  to prevent it from being recreated on every render.

It is especially useful when passing callbacks to child components that rely on reference equality to prevent unnecessary re-renders.

Syntax

```
1 const memoizedFunction = useCallback(() ⇒ { /* Callback logic */ },
  [dependencies]);
```

**useCallback** takes a function and **an array of dependencies**, and returns a **memoized version** of the function that only changes when the dependencies change.

# Example:

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

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

  const increment = useCallback(() => {
    setCount((c) => c + 1);
  }, []);
                Empty dependency array means it will only be created once

  return <ChildComponent increment={increment} />;
}
```

- The **ParentComponent** has a state count initialized to 0.
- The increment function is defined using useCallback. It updates the count state by incrementing it by 1.
- The **useCallback** hook takes an **empty dependency array [ ]**, which means the increment function will only be created **once**, when the component mounts, and not on every render.

# Custom Hooks

A custom hook in React is a **reusable function** that uses other hooks to provide a specific functionality.

It's a way to extract component logic into a reusable function that can be used across **multiple** components.

They follow the **naming convention** of starting with **"use"** (e.g., useFetchData, useAuth). Example:

```
1  import { useState, useEffect } from 'react';
2
3  function useFetchData(url) {
4    const [data, setData] = useState(null);   // State to store the fetched data
5    const [loading, setLoading] = useState(true); // State to indicate whether
   data is being loaded
6
7    useEffect(() => {
8      async function fetchData() {
9        const response = await fetch(url); // Fetch data from the URL
10       const result = await response.json(); // Convert the response to JSON
   format
11       setData(result); // Update the state with the fetched data
12       setLoading(false); // Update the loading state to false since data has
   been loaded
13     }
14     fetchData(); // Call the fetch function
15   }, [url]); // Dependency array - the effect runs when `url` changes
16
17   return { data, loading }; // Return the fetched data and loading state
18 }
```

## Breakdown of the Custom Hook:

- The **useState** initializes a state variable data with an initial value of null and another state variable loading with an initial value of true.
- The **useEffect** hook is used to perform side effects in function components. In this case, the side effect is fetching data from the provided URL.
- An **asynchronous function (fetchData)** contains the logic for fetching the data, which is stored in the variable **response**.
- The **response** is then converted to a **JSON format** and stored in the variable **result** which is passed into the **setData** to update the **data** state.
- As soon as the data is fetched, the **loading** state is set to **false**. The **useEffect** hook is cleaned up by calling the **fetchData** function immediately.
- The **data** and **loading** state are returned which can now be used in other components using the **useFetchData** hook.

# Using the custom hook (useFetchData) in a component

```jsx
function App() {
  const { data, loading } = useFetchData('https://api.example.com/data'); // Using the custom hook

  if (loading) {
    return <div>Loading...</div>; // Show a loading message while data is being fetched
  }

  return (
    <div>
      <h1>Fetched Data:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre> {/* Display the fetched data */}
    </div>
  );
}
```

- useFetchData is called with the URL ('https://api.example.com/data'). It returns an object with two values: **data** (the fetched data) and **loading** (a boolean indicating the loading state).
- While **loading** is true, the component displays "Loading...".
- Once **loading** becomes **false** (i.e., data is fetched), it renders the data.

# Conclusion

- **Class Components** are managed using class syntax and lifecycle methods like componentDidMount and componentDidUpdate to handle state and side effects. They can be more verbose and are gradually being replaced by modern approaches.

- **Functional Components** leverage React hooks like useState and useEffect to manage state and side effects, making them more concise, easier to read, and test. They are the preferred choice for new React development due to their simplicity, flexibility, and better performance.

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

kodemaven-portfolio.vercel.app

# Hi There!

## Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi



kodemaven-portfolio.vercel.app