REACT -3 State, Lifting the state and useEffect

State in React
Form Handling in React
useEffect Hook and how to use it
Lifting the state up in React

title:State and useState hook in React

What is a state ?

In React, a "state" is a built-in object that represents the current condition of a component. It's essentially a JavaScript object that holds data relevant to a component and can influence what is rendered on the user interface.

State allows components to manage dynamic data and update the UI accordingly based on user interactions, network responses, or any other event.

Unlike props, which are passed down from parent to child components and are immutable, state is managed internally by each component and can be changed. When the state of a component changes, React automatically re-renders the component, updating the UI to reflect the new state.

Hooks

In React, hooks are special functions that allow you to add extra capabilities to your functional components. They enable you to manage state, perform side effects, and reuse logic in a more organized and efficient way.

useState

useState is a hook in React that allows functional components to manage state. State in React is data that can change over time, and when state changes, React re-renders the component to reflect the updated state.

Let's create a simple React component that utilizes the useState hook.

Open the src/App.js file in your project directory, and replace its contents with the following code

```
import "./App.css";
import { useState } from "react";

function App() {
 // Define a state variable named "count" and a function to update
it, "setCount"
 const [count, setCount] = useState(0);

 // Define a function that increments the count
 const incrementCount = () => {
   setCount(count + 1); // Update the count state with the new value
 };
```

```
  // Define a function that decrements the count
  const decrementCount = () => {
    setCount(count - 1); // Update the count state with the new value
  };
  return (
    <div>
      <h1>Counter App</h1>
      <p>Count: {count}</p>
       {/* Button to increment count */}
       <button onClick={incrementCount}>Increment</button>
       {/* Button to decrement count */}
       <button onClick={decrementCount}>Decrement</button>
    </div>
  );
}

export default App;
```

Explanation

We import React and useState from the 'react' package. useState is the hook we'll use to manage state in our component.

Inside the App function component, we call the useState hook with an initial value of 0 (useState(0)). This initializes a state variable named count with an initial value of 0, and returns an array with two elements: the current state value (count) and a function to update the state (setCount).

We define two functions, incrementCount and decrementCount, which update the count state by calling setCount with the new value.

In the JSX returned by the component, we display the current value of count using curly braces {count}, and provide buttons to increment and decrement the count. These buttons call the incrementCount and decrementCount functions respectively when clicked.
If there is an error anytime, you can try restarting the dev server

npm run dev

title:State and Props Difference

In React, both state and props are used to manage data and pass information between components, but they serve different purposes and have distinct characteristics:

State:

1. Local to Component: State is internal data that belongs to a specific component. It is managed and updated within the component itself using the useState() hook (for functional components) or this.setState() method (for class components) as we will learn later.
2. Mutable: State can be changed using the setState() function, causing the component to re-render with the updated state.

3. Private to Component: State is not accessible outside of the component it belongs to, making it private and encapsulated within the component.
4. Initialized: State is typically initialized within the component's constructor (for class components) or using the useState() hook with initial values (for functional components).

Props:

1. External Data: Props (short for "properties") are passed from parent components to child components. They represent data that is received by a component from its parent and are immutable within the component receiving them.
2. Immutable: Props cannot be changed within the component that receives them. They are read-only and can only be modified by the parent component that passes them.
3. Accessed via Parameters: Props are accessed as parameters in functional components or via this.props in class components.
4. Optional: Props are optional and can be used to customize or configure a component based on the data provided by its parent.

In summary, state is used for managing internal component data that can change over time, while props are used for passing data from parent to child components to customize their behavior and appearance. Understanding the differences between state and props is essential for effective React component design and development.

title:Form Handling in React

Handling forms in React is an essential aspect of many applications, particularly those that interact with users. React provides a way to manage form inputs, handle events, and manage state effectively.

Basic Structure of a React Form

In a typical React application, forms are managed using state. This state stores the values of form inputs, which are updated using event handlers. Here's a step-by-step breakdown:

Creating a Form Component:
A form component can be created as a functional component or a class component. Here, let's use a functional component.

Using State to Manage Form Data:
We'll use the useState hook from React to store and manage form input values. For each input field, a state variable is created, and its value is updated through an event handler.

Handling Form Submission:
A form submission typically involves gathering all form input values, validating them, and then performing an action, such as sending them to a server or displaying them.

Here's how you can implement a simple form in React that collects a user's name and email:

Create a new file Form.jsx

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

const SimpleForm = () => {
 // Initializing state for form inputs
 const [name, setName] = useState("");
 const [email, setEmail] = useState("");

 // Handler for form submission
 const handleSubmit = (event) => {
   event.preventDefault(); // Prevent the default form submission

   // Here, you can handle the collected form data
   console.log(`Name: ${name}, Email: ${email}`);

   // Optionally, reset the form after submission
   setName("");
   setEmail("");
 };

 return (
   <form onSubmit={handleSubmit}>
     <div>
       <label htmlFor="name">Name:</label>
       <input
         type="text"
         id="name"
         value={name}
         onChange={(e) => setName(e.target.value)}
```

```
        />
      </div>

      <div>
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
      </div>

      <button type="submit">Submit</button>
    </form>
  );
};

export default SimpleForm;
```

Step-by-Step Explanation:

State Initialization: We initialize two state variables (name and email) to store the input values using the useState hook.

Input Fields: The form includes two input fields for name and email. These fields are bound to the state variables via the value attribute. This makes the form a "controlled component," as its value is controlled by React state.

Event Handling: The onChange attribute is set on each input field, which updates the respective state variables whenever the input changes.

Form Submission: The onSubmit attribute on the form triggers the handleSubmit function when the form is submitted. This function prevents the default form submission behavior using event.preventDefault() and then performs an action with the collected form data.

Optional Reset: After the form submission, the input fields can be cleared by resetting the state variables.

EXTRA notes

Advanced Form Handling

For more complex forms, you might consider:

Multiple State Variables: For forms with many fields, you can use a single state object to store all field values, and update them dynamically.

```
// Initializing state for form inputs
const [formData, setFormData] = useState({ name: '', email: '' });
```

```
const handleChange = (event) => {
  const { name, value } = event.target;
  setFormData((prevData) => ({ ...prevData, [name]: value }));
};
```

Validation: You can add validation logic within the handleSubmit function or create separate functions for validation.

```
const validateForm = () => {
  if (!formData.name || !formData.email) {
    return false;
  }
  // Additional validation logic...
  return true;
};
 const handleSubmit = (event) => {
  event.preventDefault();
   if (!validateForm()) {
    console.error('Form is invalid');
    return;
  }
   console.log(`Name: ${formData.name}, Email: ${formData.email}`);
};
```

Third-party Libraries: For complex form handling and validation, consider libraries like Formik or React Hook Form, which provide powerful utilities and built-in validation mechanisms.

Here's a brief example using Formik:

```
import React from 'react';
import { Formik, Field, Form, ErrorMessage } from 'formik';
import * as Yup from 'yup';
```

```
const MyForm = () => {
 return (
   <Formik
     initialValues={{ name: '', email: '' }}
     validationSchema={Yup.object({
       name: Yup.string().required('Name is required'),
       email: Yup.string().email('Invalid email address').required('Email is
required'),
     })}
     onSubmit={(values) => {
       console.log('Form data:', values);
     }}
   >
     <Form>
       <div>
         <label htmlFor="name">Name:</label>
         <Field name="name" type="text" />
         <ErrorMessage name="name" component="div" />
       </div>

       <div>
         <label htmlFor="email">Email:</label>
         <Field name="email" type="email" />
         <ErrorMessage name="email" component="div" />
       </div>

       <button type="submit">Submit</button>
     </Form>
   </Formik>
 );
};

export default MyForm;
```

title:Lifting the state up

Let's start explore the concept of "lifting state up" in React through a practical example involving two components: TemperatureInput for

entering temperature and TemperatureDisplay for showing that temperature in both Celsius and Fahrenheit.

Initially, let's consider a setup where each component manages its own state independently.

TemperatureInput Component

This component allows a user to input a temperature. It maintains its own state.

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

function TemperatureInput() {
 const [temperature, setTemperature] = useState('');

 return (
   <div>
     <label>
       Enter temperature:
       <input
         type="text"
         value={temperature}
         onChange={(e) => setTemperature(e.target.value)}
       />
     </label>
   </div>
 );
}
export default TemperatureInput;
```

TemperatureDisplay Component

This component displays the entered temperature in Celsius and Fahrenheit but also maintains its own state

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

function TemperatureDisplay() {
 const [temperature, setTemperature] = useState('');

 const fahrenheit = (temperature * 9 / 5) + 32;

 return (
   <div>
     <p>Temperature in Celsius: {temperature}°C</p>
     <p>Temperature in Fahrenheit: {fahrenheit.toFixed(1)}°F</p>
   </div>
 );
}
export default TemperatureDisplay;
```

Problems with Independent State Management here

1. Redundancy: Both components maintain a temperature state, leading to redundant data storage.
2. Lack of Synchronization: Changes in the temperature entered in TemperatureInput do not automatically reflect in TemperatureDisplay, leading to inconsistencies.
3. Difficulty in Propagation: If other parts of the application need to react to temperature changes, managing these changes becomes complex as updates are confined to local components.

Optimal Solution: Lifting the State Up

To solve these issues, we move the shared state to their closest common ancestor, which is the App component in this scenario. This approach ensures that both components are synchronized and simplifies the overall management of the state.

```
import { useState } from "react";
import TemperatureInput from "./components/TemperatureInput";
import TemperatureDisplay from "./components/TemperatureDisplay";

function App() {
 const [temperature, setTemperature] = useState("");

 const handleTemperatureChange = (newTemperature) => {
   setTemperature(newTemperature);
 };

 return (
   <div>
     <TemperatureInput
       temperature={temperature}
       onTemperatureChange={handleTemperatureChange}
     />
     <TemperatureDisplay temperature={temperature} />
   </div>
 );
}

export default App;
```

Modified Child Components

TemperatureInput now takes temperature and a function onTemperatureChange as props. It uses these props to display and update the temperature:

```
function TemperatureInput({ temperature, onTemperatureChange }) {
 return (
   <div>
     <label>
       Enter temperature:
       <input
         type="text"
         value={temperature}
         onChange={(e) => onTemperatureChange(e.target.value)}
       />
     </label>
   </div>
 );
}
export default TemperatureInput;
```

emperatureDisplay receives the temperature as a prop and uses it to display it in different units, thus remaining always up-to-date:

Benefits of Lifting the State Up

1. Single Source of Truth: Centralizing the state in the App component simplifies management and debugging.

2. Consistent Data: All components that depend on the temperature will automatically receive the updated state, ensuring consistency.
3. Easier Feature Expansion: Adding features or components that depend on the temperature becomes easier, as they can all tap into the same state via props.

This strategy is not only crucial for maintaining synchronization across components but also promotes a more scalable and maintainable structure in React applications.

title:The useEffect Hook

Before moving to useEffect we need to understand stages of a component, we call them as lifecycle of a component, In React a

Component can have three stages , mounting, updation and unmounting, these are the terminolgies which were often used with class based components and we had so many method there as well like

componentDidMount, componentDidUpdate, and componentWillUnmount in class components

Let's understand this lifecycle first and then we will see how useEffect can actually manage all these stages in an unified API

Components are mounted, they have updates and then they die or are unmounted

Mounting

Mounting is the stage where the component is being created and inserted into the UI. It's like setting up the stage before the actual work begins

In React, this is where the component is initially constructed. It sets up its initial state, fetches any necessary data if needed, and then renders the layout for the first time. This is typically where you would put code that needs to run when the component appears for the first time.

Updating

Updating occurs after the component has been mounted and involves any changes to the component’s state or props (the parameters passed into a component).

In React, a component updates in response to user interactions, data receptions, or prop changes. This might involve recalculating the state, or it might result in a re-rendering of the user interface.

For example, if a user fills out a form on a webpage, each keystroke might update the component's state, causing the component to render some visual feedback.

Counter button where you click on increment counter and the value changes on the UI

Unmounting

Unmounting is the final phase of the component lifecycle. The component is being removed from the screen, and this is where you clean up after your component: canceling any outgoing network requests, removing any event listeners, or clearing any caches or data that were set up for the component.

In React, when a component is no longer needed, it is unmounted to free up resources. This helps in preventing memory leaks and ensuring that components that aren't being used don't consume resources unnecessarily.

Each of these phases is crucial for managing resources efficiently and ensuring that the application behaves correctly and efficiently updates the user interface.

React manages each phase through a series of methods and hooks that allow developers to run code at specific points in a component's

lifecycle, adapting to the dynamic and interactive nature of modern web applications.

Now how does all this discussion tie to the useEffect

useEffect hooks will allow us to run code as part of all three phases - on mounting, when some updates happen and when component is unmounted

But generally what kind of code we would want to write inside a useEffect hook ?

The useEffect hook in React allows you to perform side effects and handles all these three phases in your function components.

Side effects are operations that help your components talk to or sync to any external system like an api or DOM

Side effects are operations that can affect other components or cannot be done during rendering, such as data fetching, subscriptions, or manually changing the DOM.

So what kind of code goes inside useEffect ?

1. useEffect is designed for side effects, operations that should not occur during the main rendering process to avoid blocking the UI.

2. These are typically operations that interact with external systems or require asynchronous execution. Examples include:
    a. Data Fetching: Making API calls to retrieve or update data outside the local state.
    b. Subscriptions: Setting up listeners or subscriptions, such as to a WebSocket service or an event listener.
    c. Manually Changing the DOM: Though React generally handles DOM updates, you might need to interact with the DOM directly for integrating with non-React libraries or for animations.
    d. Timers: Implementing intervals or timeouts, such as starting an auto-refresh or delay-display mechanism.
3. Key Concepts of Side Effects in React
4. Side effects are operations that:
    a. Interact with External Systems: Like databases, APIs, or other external data sources and services.
    b. Modify Observable Conditions Outside the Component: Such as changes in the global state, local storage, or even the URL.
    c. Cannot Be Done During Rendering: Because they might delay the rendering process, lead to inconsistent renders, or are not directly related to the output of the render method.

Basic Usage of useEffect

useEffect takes two arguments:

A function that contains the code for the side effect.
An optional array of dependencies that determines when the effect should rerun.

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

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

 console.log("Component render", count);

 useEffect(() => {
   console.log("useEffect run", count);
   document.title = `You clicked ${count} times`;
 }, [count]); // Dependency array

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

export default ExampleComponent;
```

In this example, the useEffect hook is used to update the document title every time the count state changes. The effect runs after the first render and after every update of count.

This React component, named ExampleComponent, demonstrates the use of the useState and useEffect hooks from React to manage state and side effects in a functional component

Code explanation:

useEffect is used to perform side effects in the component.
Inside the effect function (() => { document.title = You clicked ${count} times; }), it updates the document's title to display how many times the button has been clicked, incorporating the current count value.

[count]: This is the dependency array. It tells React to re-run the effect function whenever the value of count changes. If count does not change, the effect does not rerun.

Dependency Array

The dependency array is a crucial part of using useEffect effectively:

1. No dependency ([]): The effect runs only once after the initial render.
2. With dependencies ([dependencies]): The effect runs after the initial render and only if any dependency has changed since the last render.
3. No dependency array: The effect runs after every render.
   useEffect(() => {

```
    console.log('Component updated');
  });
```

So useEffect is like a timer. It reminds react that you need to run a piece of code. Some timers can be one time, some can be regular, etc

Cleaning up an Effect

Sometimes you need to clean up your effects to prevent memory leaks (e.g., removing event listeners or cancelling subscriptions). Here's how you can implement a cleanup in useEffect:

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Timer tick');
  }, 1000);
   // Cleanup function
  return () => {
    clearInterval(timer);
  };
}, []);  // Runs only once on mount and unmount
```

Practical Example: Fetching Data

useEffect is commonly used for data fetching:

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

function FetchDataComponent() {
 const [data, setData] = useState(null);
 console.log(data);
```

```
useEffect(() => {
  const fetchData = async () => {
    const response = await fetch('https://jsonplaceholder.typicode.com/users');
    const result = await response.json();
    setData(result);
  };

  fetchData();
}, []);  // Only runs once

return (
  <div>
    {data ? <p>Data loaded! with first user {data[0].name}</p> : <p>Loading...</p>}
  </div>
);
}
export default FetchDataComponent;
```

Code explanation

1. useState(null): Initializes a state variable called data with a default value of null. This state will hold the fetched data once it is retrieved.
2. useEffect(() => {...}, []): This is a React hook that performs side effects in function components. The function inside useEffect runs after the first render and not after any subsequent renders, due to the empty dependency array [].
3. fetchData Function: Inside useEffect, a function named fetchData is declared. This function is asynchronous (async), indicating it includes asynchronous operations, specifically a network request in this case.

4. fetch API: fetch('https://jsonplaceholder.typicode.com/users') is called to make a network request to the provided URL, which fetches user data from a placeholder API.
5. response.json(): The response from the fetch call is processed to extract JSON from it. This JSON is expected to be an array of user data.
6. setData(result): The fetched result (user data) is then stored in the state variable data using the setData function, updating the component's state and triggering a re-render.
7. Side Effect Trigger: The fetchData function is called immediately within the useEffect to execute the fetching operation.
8. Conditional Rendering: The component conditionally renders based on the state of data:
    a. Data loaded! : If data is not null (meaning the data has been successfully loaded), it displays this paragraph.
    b. Loading... : If data is still null (meaning the data is still being fetched or has not yet been fetched), it displays this paragraph indicating that the data is loading.

title:The Rules of using hooks

React hooks are a feature introduced in React 16.8 that allow you to use state and other React features without writing a class. Here are some fundamental rules for using hooks in React:

1. Call Hooks at the Top Level:

    a. Hooks should always be called at the top level of your React function components. Do not call Hooks inside loops, conditions, or nested functions.
    b. This ensures that hooks are called in the same order each time a component renders, which is necessary for React to correctly preserve the state of hooks between multiple useState and useEffect calls.
2. Call Hooks from React Functions:
    a. You can only call Hooks from React function components or from custom hooks.
    b. Do not call Hooks from regular JavaScript functions, classes, or any other non-React function.
3. Only Call Hooks from React Components or Custom Hooks:
    a. You should not call Hooks from any place other than a function component or a custom hook.
    b. This includes calling Hooks from regular JavaScript functions, event handlers, setTimeout or setInterval callbacks, and side effects within useEffect.
    c. Custom Hooks Must Follow the Same Rules as Built-in Hooks: When you create your own custom hooks, they must also follow the rules of Hooks. Custom hooks should have a name starting with "use" and should encapsulate hook logic. (we will see How to create custom hooks in later classes)

Check Rules with ESLint Plugin: To help follow these rules, you can use the ESLint plugin called eslint-plugin-react-hooks that enforces these rules when using hooks in your code.

These rules are designed to ensure that components behave consistently and that hooks work as expected. They help manage state and side effects in React's functional UI paradigm.