1. `useState` Hook

The `useState` hook is used to manage state in a functional component. It returns an array with two elements: the current state value and a function to update it.

Ex =

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


function Counter() {
  // Declare a state variable named 'count', initialized to 0
  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;
```

In this example, `useState(0)` initializes the `count` state variable to 0. The `setCount` function updates the `count` state.

2. `useEffect` Hook

The `useEffect` hook allows you to perform side effects in function components. It can be used for tasks like data fetching, subscriptions, or manually changing the DOM.

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

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

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

    // Cleanup interval on component unmount
    return () => clearInterval(interval);
  }, []); // Empty dependency array ensures this runs once on mount

  return (
    <div>
      <p>{seconds} seconds have elapsed.</p>
    </div>
  );
}

export default Timer;
```

In this example, `useEffect` sets up an interval that increments the `seconds` state every second. The cleanup function `clearInterval(interval)` ensures the interval is cleared when the component unmounts.

3. `useReducer` Hook

The `useReducer` hook is used for managing complex state logic in a functional component. It's an alternative to `useState` and is usually preferred when you have multiple pieces of state or complex state transitions.

```
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' })}>
        Increment
      </button>
      <button onClick={() => dispatch({ type: 'decrement' })}>
        Decrement
```

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

export default Counter;
```

In this example, `useReducer` is used to manage the `count` state. The `reducer` function takes the current state and an action, and returns the new state based on the action type. The `dispatch` function is used to send actions to the reducer.

`useParams` Hook

The `useParams` hook is used in React Router to extract parameters from the URL. It's useful when you need to access dynamic segments of your route.

Example:

Assume you have a route defined like `/user/:id` and you want to access the `id` parameter in your component.

Setup React Router:

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import UserProfile from './UserProfile';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/user/:id" element={<UserProfile />} />
      </Routes>
    </Router>
  );
}

export default App;
```

Using `useParams` in UserProfile component:

```
// UserProfile.js
import React from 'react';
import { useParams } from 'react-router-dom';
```

```
function UserProfile() {
  const { id } = useParams();

  return (
    <div>
      <h1>User Profile</h1>
      <p>User ID: {id}</p>
    </div>
  );
}

export default UserProfile;
```

In this example, the `useParams` hook extracts the `id` parameter from the URL and makes it available in the `UserProfile` component.

`useRef` Hook

The `useRef` hook creates a mutable object which holds a `.current` property. This is useful for accessing DOM elements directly, storing mutable values that do not cause re-renders when updated, or persisting values across renders.

Example:

1. **Accessing a DOM Element:**

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


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


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


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


export default TextInputFocus;
```

In this example, `useRef` is used to create a reference to the input element. The `focusInput` function calls `inputRef.current.focus()`, which sets the focus on the input element when the button is clicked.

2. **Storing Mutable Values:**

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


function Stopwatch() {
```

```
const [time, setTime] = useState(0);
const timerId = useRef(null);

useEffect(() => {
  return () => clearInterval(timerId.current);
}, []);

const start = () => {
  if (timerId.current) return; // Prevent multiple intervals
  timerId.current = setInterval(() => {
    setTime(prevTime => prevTime + 1);
  }, 1000);
};

const stop = () => {
  clearInterval(timerId.current);
  timerId.current = null;
};

const reset = () => {
  clearInterval(timerId.current);
  timerId.current = null;
  setTime(0);
};

return (
  <div>
    <p>{time} seconds</p>
    <button onClick={start}>Start</button>
    <button onClick={stop}>Stop</button>
    <button onClick={reset}>Reset</button>
```

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

export default Stopwatch;

In this example, `useRef` is used to store the interval ID for the timer. This allows the interval to be cleared when the component unmounts or when the stop/reset buttons are clicked.

Summary

- **`useState`**: Simplest way to manage state in a functional component.
- **`useEffect`**: Handles side effects like data fetching, subscriptions, or changing the DOM.
- **`useReducer`**: Manages complex state logic, preferred for multiple related state values or complex state transitions.
- **`useParams`**: Used in React Router to extract URL parameters.
- **`useRef`**: Creates a mutable reference object to access DOM elements or store mutable values that persist across renders without causing re-renders.

React Router DOM is a popular library for handling routing in React applications. It allows developers to create single-page applications (SPAs) with dynamic navigation and rendering of different components based on the URL. React Router DOM helps manage the navigation state of the application, making it possible to navigate between different views without refreshing the page.

Key Concepts of React Router DOM

1. **Router**: The main component that enables routing in your application. It uses the HTML5 history API to keep your UI in sync with the URL.
2. **Routes**: Define the mapping between URL paths and the components to be rendered. Each `Route` component specifies a path and the component to render when the path matches.
3. **Link**: A component used to create navigation links. It allows users to navigate to different routes in the application without reloading the page.
4. **Switch**: Renders the first child `Route` that matches the current location. It ensures only one route is rendered at a time.

5. **useParams**: A hook that returns an object of key/value pairs of URL parameters. It is used to access route parameters.
6. **useHistory**: A hook that provides access to the history instance used by React Router. It allows navigation programmatically.
7. **useLocation**: A hook that returns the current location object, representing where the app is now.

The end

go through this notes and ask chatgpt for more but go more deeper into it because I want u should understand it properly

Thankyou so much

Your feedbacks -

@Vengat – Greate work [improving regularly]

@Shubhash – Active and learning but less concentrated be focused

@Neeraj – You Have knowledge just be consistent good work

@Prakash – Consistent but give more time for tasks

@Raghuveer – Trying well - Focus on Your logics sometime oversmart things can hurt us very badly

@RajaShekhar – Learning but I want more hard work then only u can get things quickly okay

@Shalija , @Bhavya – inconsistent not learning