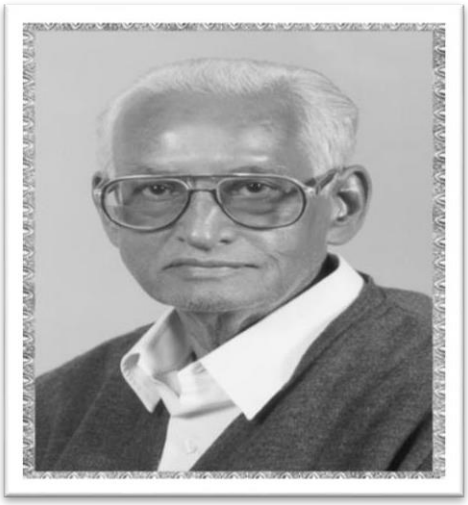
SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.
(AFFILIATED TO SAURASHTRA UNIVERSITY)



Lt. Shree Chimanbhai Shukla

MSCIT SEM-2 REACTJS

Shree H.N.Shukla college2
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Shree H.N.Shukla college3
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Unit :5

Introduction to Hooks and its implementation

- **Introduction:** React Hooks introduction, useState Hook, useState Previous state, useState with object, useState with array.
- **useEffect:** useEffect Hook, useEffect after render, Conditionally run effects, run effects only once,useEffect with cleanup, useEffect with incorrect dependency.
- **Fetching data:** Fetching data with useEffect, useContext Hook
- **useReducer Hook:** useReducer – simple state andaction, complex state and action, multiple useReducers
- **useContext:** useContext, useReducer, Fetching data with useReduer, useState vs useReducer

Introduction:

React Hooks introduction

React Hooks are functions that allow developers to use state and other React features in functional components. Before the introduction of Hooks, state management and other React features were primarily used in class components. With the introduction of Hooks in React 16.8, developers gained more flexibility and simplicity in managing state and side effects in functional components.

Here are some key points about React Hooks:

useState: This is a Hook that allows functional components to manage state. It returns a stateful value and a function to update it. Here's a basic example:

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

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useEffect: This Hook adds the ability to perform side effects in functional components. It's similar to componentDidMount, componentDidUpdate, and componentWillUnmount lifecycle methods combined. Here's an example:

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

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

React Hooks are functions that allow developers to use state and other React features in functional components. Before the introduction of Hooks, state management and other React features were primarily used in class components. With the introduction of Hooks in React 16.8, developers gained more flexibility and simplicity in managing state and side effects in functional components.

Here are some key points about React Hooks:

useState: This is a Hook that allows functional components to manage state. It returns a stateful value and a function to update it. Here's a basic example:

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

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

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

useEffect: This Hook adds the ability to perform side effects in functional components. It's similar to componentDidMount, componentDidUpdate, and componentWillUnmount lifecycle methods combined. Here's an example:

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

function Example() {
```

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

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

1.useContext: This Hook allows you to consume context within a functional component. It lets you access context values and subscribe to changes. Here's a basic example:

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

function MyComponent() {
  const value = useContext(MyContext);
  return <div>{value}</div>; }
```

1.useReducer: This Hook is a more powerful alternative to useState. It's used for more complex state logic. It accepts a reducer function and an initial state. Here's an example:

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

These are just a few examples of the many Hooks available in React. They provide a more concise and readable way to work with state and side effects in functional components, making React development more efficient and enjoyable.

useState Hook

In React.js, the useState hook is used to add state to functional components. It allows you to declare state variables and provides a function to update them. Here's how you can use the useState hook:

Example:

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

function Example() {
  // Declare a state variable named "count" and its updater function "setCount"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      {/* On click, call setCount to update the state */}
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Example;
```

in this example:
We import the useState hook from the react package.
Inside the Example functional component, we call the useState hook with an initial state value of 0, and it returns an array with two elements: the current state value (count) and a function (setCount) to update it.
We render the current state value (count) in a paragraph.
When the button is clicked, it calls the setCount function with the new value (count + 1), updating the state.
Remember that you can call useState multiple times in a single component to manage multiple pieces of state independently. Each call to useState maintains its own state and update function.

useEffect After Render

In React.js, the useEffect hook allows you to perform side effects in function components. Side effects could be anything from fetching data, subscribing to events, updating the DOM, etc. The useEffect hook runs after every render of the component, including the initial render and subsequent re-renders. Here's how you can use useEffect:

Javascript

Example:-

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  // This effect will run after every render
  useEffect(() => {
    // Update the document title using the count state
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default Example;
```

In this example:
We import the useEffect hook from the react package.
Inside the Example functional component, we call the useEffect hook. The first argument is a function that contains the side effect you want to perform (in this case, updating the document title), and the second argument is an array of dependencies. If any of the dependencies change between renders, the effect function will be called again.
Since we didn't provide a second argument to useEffect, it will run after every render.
Inside the effect function, we update the document title to reflect the current value of the count state.
Remember to handle cleanup in useEffect if necessary, and if you only want the effect to run once (similar to componentDidMount in class components), you can pass an empty array [] as the second argument to useEffect.

Conditionally run effects

To conditionally run effects in React using the useEffect hook, you can provide a dependency array as the second argument to useEffect. If any of the dependencies in the array change between renders, the effect function will be called again. If you want to conditionally run the effect based on certain conditions, you can include those conditions in the dependency array.

Here's an example of how you can conditionally run effects:

Example:-

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

function Example() {
  const [count, setCount] = useState(0);
  const [shouldRunEffect, setShouldRunEffect] = useState(true);

  useEffect(() => {
    if (shouldRunEffect) {
      // Run the effect only if shouldRunEffect is true
      console.log("Effect ran!");
    }
  }, [shouldRunEffect]); // Dependency array with shouldRunEffect as the dependency

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

    export default Example;
```

In this example:

We have a state variable shouldRunEffect to determine whether the effect should run or not.
Inside the useEffect hook, we check the value of shouldRunEffect. If it's true, we perform the side effect (in this case, logging "Effect ran!").
The dependency array of useEffect contains shouldRunEffect. This means that the effect will run whenever the value of shouldRunEffect changes.
We have a button to toggle the value of shouldRunEffect. Clicking this button will toggle whether the effect runs or not.
By changing the value of shouldRunEffect, you can control whether the effect should run or not based on your conditions.

run effects only once

To run effects only once in a React component, you can provide an empty dependency array [] as the second argument to the useEffect hook. This ensures that the effect runs only after the initial render of the component and doesn't run again for subsequent renders. Here's how you can achieve that:

Example:-

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

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

  useEffect(() => {
    // This effect will run only once after the initial render
    console.log("Effect ran!");
  }, []); // Empty dependency array ensures the effect runs only once

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

```
}

export default Example;
```

In this example:

We use the useEffect hook to perform a side effect (logging "Effect ran!") after the initial render of the component.
We provide an empty dependency array [] as the second argument to useEffect. This tells React to run the effect only once after the initial render, and not to re-run it when any dependencies change.
The effect runs only after the initial render, and subsequent updates to the component's state (such as clicking the button) won't trigger it again.
This pattern is useful for scenarios where you need to perform initialization tasks or subscribe to external data sources that you only want to do once when the component mounts.

useEffect with cleanup

In React, you can use the useEffect hook to perform side effects in function components. Sometimes, these side effects might require cleanup when the component unmounts or before the effect runs again due to a change in dependencies. You can achieve cleanup by returning a cleanup function from the effect.

Here's an example of how you can use useEffect with cleanup:

Example:-

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

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

  useEffect(() => {
    // This effect will run after every render
    console.log("Effect ran!");

    // Cleanup function
```

```
  return () => {
    console.log("Cleanup");
    // Perform cleanup tasks here (e.g., unsubscribe from subscriptions)
  };
 }, []); // Empty dependency array ensures the effect runs only once

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

   export default Example;
```

In this example:

We use the useEffect hook to perform a side effect (logging "Effect ran!") after the initial render of the component.
We provide an empty dependency array [] as the second argument to useEffect, so it runs only once after the initial render.
Inside the effect function, we return a cleanup function. This cleanup function will be called when the component unmounts or before the effect runs again due to changes in dependencies.
In the cleanup function, you can perform any necessary cleanup tasks, such as unsubscribing from subscriptions or clearing timers.
Cleanup functions are crucial for managing resources and preventing memory leaks in your application. They help ensure that your application is properly cleaned up when components are unmounted or when dependencies change.

useEffect with incorrect dependency.

Using the useEffect hook with incorrect dependencies can lead to unexpected behavior and bugs in your React application. When you specify dependencies in the dependency array of useEffect, React will execute the effect whenever any of those dependencies change. If you provide incorrect or missing dependencies, it can cause your effect to either run too frequently or not at all, which can lead to issues in your application.

Here are some common mistakes related to dependencies in the useEffect hook and how to avoid them:

1. **Empty Dependency Array ([]):**

If you pass an empty dependency array ([]) to useEffect, it will run the effect only once, similar to componentDidMount in class components. This is useful for effects that don't depend on any state or props. However, if your effect does depend on state or props, omitting them from the dependency array can cause the effect to not update when those values change.

```
useEffect(() => {
  // Effect code
    }, []); // Empty dependency array
```

To fix this, include the necessary dependencies in the array:

```
const [count, setCount] = useState(0);
useEffect(() => {
  // Effect code that depends on count
}, [count]); // Include count in the dependency array Missing Dependencies:
```

2. **Missing Dependencies:**

Forgetting to include all the dependencies your effect relies on can lead to bugs where the effect doesn't update when expected. React will only re-run the effect if the values in the dependency array change.

```
const [username, setUsername] = useState('');
const [userData, setUserData] = useState(null);
```

```
useEffect(() => {
  // Effect code that depends on username but doesn't include it in the dependencies
  fetchUserData(username).then(data => setUserData(data));
    }, []); // Missing username dependency
```

To fix this, add any missing dependencies to the array:

```
useEffect(() => {
  // Effect code that depends on username
  fetchUserData(username).then(data => setUserData(data));
    }, [username]); // Include username in the dependency array
```

Using Props Directly in Effect:

When using props inside an effect, make sure to include those props in the dependency array if the effect depends on their values. This ensures that the effect runs whenever those props change.

```
useEffect(() => {
  // Effect code that uses props directly but doesn't include them in dependencies
  console.log(props.someProp);
    }, []); // Missing props dependency
```

To fix this, add the props to the dependency array:

```
useEffect(() => {
  // Effect code that depends on props
  console.log(props.someProp);
    }, [props.someProp]); // Include props.someProp in the dependency array
```

By following these best practices and ensuring that your useEffect dependencies are correctly set, you can avoid many common issues related to the useEffect hook in React.

Fetching data with useEffect

In React, the useEffect hook is used to perform side effects in function components. One common use case for useEffect is fetching data from an API when the component mounts. Here's an example of how you can fetch data using useEffect:

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

const MyComponent = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/data');
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();

    // Cleanup function (optional)
    return () => {
      // Perform cleanup actions if needed
    };
  }, []); // Empty dependency array means this effect runs once on mount

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

```
  return (
    <div>
      {/* Render your data here */}
      {data && (
        <ul>
          {data.map(item => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      )}
    </div>
  );
};

export default MyComponent;
```

In this example:

We initialize state variables data, loading, and error using the useState hook.
Inside the useEffect hook, we define an asynchronous function fetchData that fetches data from the API using fetch and updates the state variables accordingly.
The useEffect hook runs once when the component mounts because we passed an empty dependency array ([]), indicating that there are no dependencies, so it doesn't need to run again unless those dependencies change.
The component renders different content based on the state of loading and error, showing a loading message while the data is being fetched and an error message if an error occurs.
If the data is fetched successfully (data is not null), we render the data in a list.
Remember to replace 'https://api.example.com/data' with the actual API endpoint you want to fetch data from.

useContext Hook

The useContext hook in React allows you to consume a context that has been created using the createContext function. Context provides a way to pass data through the component tree without having to pass props down manually at every level. Here's an example of how to use the useContext hook:

First, let's create a context and a provider for that context:

```
// MyContext.js
```

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

// Create a context
const MyContext = createContext();

// Create a provider for the context
const MyProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(prevCount => prevCount + 1);
  };

  return (
    <MyContext.Provider value={{ count, incrementCount }}>
      {children}
    </MyContext.Provider>
  );
};

export { MyContext, MyProvider };
```

In the code above:

We create a context called MyContext using createContext().
We create a provider component called MyProvider that wraps its children with MyContext.Provider and provides the context value { count, incrementCount }.
Inside the provider, we have a state variable count and a function incrementCount to update the count.
Now, let's use this context and provider in a component using the useContext hook:

```
import React, { useContext } from 'react';
import { MyContext } from './MyContext';

const MyComponent = () => {
  // Use the useContext hook to consume the context
  const { count, incrementCount } = useContext(MyContext);

  return (
    <div>
```

```
    <h1>Count: {count}</h1>
    <button onClick={incrementCount}>Increment Count</button>
   </div>
  );
};

export default MyComponent;
```

In the code above:

We create a context called MyContext using createContext().
We create a provider component called MyProvider that wraps its children with MyContext.Provider and provides the context value { count, incrementCount }.
Inside the provider, we have a state variable count and a function incrementCount to update the count.
Now, let's use this context and provider in a component using the useContext hook:

```
import React from 'react';
import MyComponent from './MyComponent';
import { MyProvider } from './MyContext';

const App = () => {
  return (
    <MyProvider>
      <MyComponent />
    </MyProvider>
  );
};

export default App;
```

Now, when you run your application, MyComponent will have access to the context values provided by MyProvider through the useContext hook.

useReducer – simple state andaction

useReducer is another React hook that is used for managing state, particularly when state logic involves complex updates or transitions. It's similar to Redux in terms of using reducers and actions. Let's create a simple example to demonstrate how to use useReducer with a simple state and action:

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

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'RESET':
      return { count: 0 };
    default:
      return state;
  }
};

const Counter = () => {
  // Initialize state using useReducer
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
      <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
    </div>
  );
};
```

```
export default Counter;
```

In this example:

We define a reducer function that takes state and action parameters and returns a new state based on the action type.
The Counter component uses useReducer to manage the state. It initializes the state with { count: 0 } and provides the reducer function.
Inside the component, we use the dispatch function returned by useReducer to dispatch actions with different types (INCREMENT, DECREMENT, RESET).
When a button is clicked, it dispatches the corresponding action to update the state based on the logic defined in the reducer.
To use this Counter component, you can import it into your main application file and render it:

```
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <h1>Counter App</h1>
      <Counter />
    </div>
  );
};

    export default App;
```

Now, when you run your application, you'll see the Counter component with buttons to increment, decrement, and reset the count. The state updates and re-renders based on the actions dispatched to the reducer.

useReducer – simple state and action

Certainly! Here's an example of using useReducer hook in React to manage a simple state and actions:

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

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'RESET':
      return { count: 0 };
    default:
      return state;
  }
};

const Counter = () => {
  // Initialize state using useReducer
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
      <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
    </div>
  );
};

    export default Counter;
```

In this example:

We define a reducer function that takes state and action parameters and returns a new state based on the action type.

The Counter component uses useReducer to manage the state. It initializes the state with { count: 0 } and provides the reducer function.

Inside the component, we use the dispatch function returned by useReducer to dispatch actions with different types (INCREMENT, DECREMENT, RESET).

When a button is clicked, it dispatches the corresponding action to update the state based on the logic defined in the reducer.

To use this Counter component, you can import it into your main application file and render it:

```
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <h1>Counter App</h1>
      <Counter />
    </div>
  );
};

    export default App;
```

In this example:

We define a reducer function that takes state and action parameters and returns a new state based on the action type.

The Counter component uses useReducer to manage the state. It initializes the state with { count: 0 } and provides the reducer function.

Inside the component, we use the dispatch function returned by useReducer to dispatch actions with different types (INCREMENT, DECREMENT, RESET).

When a button is clicked, it dispatches the corresponding action to update the state based on the logic defined in the reducer.

To use this Counter component, you can import it into your main application file and render it:

```
import React from 'react';
```

```
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <h1>Counter App</h1>
      <Counter />
    </div>
  );
};

    export default App;
```

When you run this code, you'll see the **Counter** component with buttons to increment, decrement, and reset the count. The state updates and re-renders based on the actions dispatched to the reducer.

multiple useReducers

Using multiple useReducers in a React component can be helpful for managing different parts of the state independently. Each useReducer manages a specific portion of the state and has its own reducer function. Here's an example of how you can use multiple useReducers in a React component:

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

// First reducer function
const reducer1 = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

// Second reducer function
```

```
const reducer2 = (state, action) => {
  switch (action.type) {
    case 'TOGGLE':
      return { isOn: !state.isOn };
    default:
      return state;
  }
};

const MultipleReducers = () => {
  // Initialize state using two useReducers
  const [state1, dispatch1] = useReducer(reducer1, { count: 0 });
  const [state2, dispatch2] = useReducer(reducer2, { isOn: false });

  return (
    <div>
      {/* Counter */}
      <h2>Counter</h2>
      <h3>Count: {state1.count}</h3>
      <button onClick={() => dispatch1({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch1({ type: 'DECREMENT' })}>Decrement</button>

      {/* Toggle */}
      <h2>Toggle</h2>
      <h3>State: {state2.isOn ? 'On' : 'Off'}</h3>
      <button onClick={() => dispatch2({ type: 'TOGGLE' })}>Toggle</button>
    </div>
  );
};

    export default MultipleReducers;
```

In this example:

We define two separate reducer functions (reducer1 and reducer2) to manage different parts of the state.

We use useReducer twice in the MultipleReducers component to create two separate state slices (state1 and state2), each with its own reducer and initial state.

The component renders two sections: one for the counter (state1) and another for the toggle state (state2).

Each section has its own set of actions (INCREMENT and DECREMENT for the counter, TOGGLE for the toggle state) that are dispatched to their respective reducers using dispatch1 and dispatch2.

To use the MultipleReducers component, you can import it into your main application file and render it:

```
import React from 'react';
import MultipleReducers from './MultipleReducers';

const App = () => {
  return (
    <div>
      <h1>Multiple Reducers App</h1>
      <MultipleReducers />
    </div>
  );
};

    export default App;
```

Now, when you run your application, you'll see the **MultipleReducers** component with separate state management for the counter and the toggle functionality.

useContext

The useContext hook in React allows you to consume a context that has been created using the createContext function. Context provides a way to pass data through the component tree without having to pass props down manually at every level. Here's an example of how to use the useContext hook:

First, let's create a context and a provider for that context:

```
// MyContext.js
import React, { createContext, useState } from 'react';
```

```
// Create a context
const MyContext = createContext();

// Create a provider for the context
const MyProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(prevCount => prevCount + 1);
  };

  return (
    <MyContext.Provider value={{ count, incrementCount }}>
      {children}
    </MyContext.Provider>
  );
};

    export { MyContext, MyProvider };
```

In the code above:

We create a context called MyContext using createContext().
We create a provider component called MyProvider that wraps its children with MyContext.Provider and provides the context value { count, incrementCount }.
Inside the provider, we have a state variable count and a function incrementCount to update the count.
Now, let's use this context and provider in a component using the useContext hook:

```
import React, { useContext } from 'react';
import { MyContext } from './MyContext';

const MyComponent = () => {
  // Use the useContext hook to consume the context
  const { count, incrementCount } = useContext(MyContext);

  return (
    <div>
      <h1>Count: {count}</h1>
```

```
    <button onClick={incrementCount}>Increment Count</button>
   </div>
  );
};

    export default MyComponent;
```

In the MyComponent:

We import the MyContext context from MyContext.js.
We use the useContext hook to access the context values (count and incrementCount).
We render the count and a button that triggers the incrementCount function when clicked.

Finally, to make this work, you need to wrap your component tree with the MyProvider component at the top level:

```
import React from 'react';
import MyComponent from './MyComponent';
import { MyProvider } from './MyContext';

const App = () => {
  return (
    <MyProvider>
      <MyComponent />
    </MyProvider>
  );
};

    export default App;
```

Now, when you run your application, MyComponent will have access to the context values provided by MyProvider through the useContext hook.

UseReducer

The useReducer hook in React is used for managing state in a more complex manner than what's possible with the useState hook alone. It's particularly useful when the state logic involves multiple actions and transitions. Here's a detailed example of how to use useReducer:

First, let's define a reducer function. This function takes the current state and an action, and returns a new state based on the action type:

```
const initialState = {
  count: 0,
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return initialState;
    default:
      throw new Error('Unhandled action');
  }
};
```

In this example, our state has a single property count, and the reducer handles three types of actions: increment, decrement, and reset.

Now, let's use the useReducer hook in a component:

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

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  );
};
```

```
export default Counter;
```

In the Counter component:

We initialize our state using useReducer, passing the reducer function and the initial state (initialState).
We destructure state and dispatch from the return value of useReducer.
We render the count from the state, and three buttons that dispatch actions (increment, decrement, reset) when clicked.
Finally, you can use the Counter component in your main application:

```
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <h1>Counter App</h1>
      <Counter />
    </div>
  );
};

export default App;
```

Now, when you run your application, you'll see the Counter component with buttons to increment, decrement, and reset the count. The state updates and re-renders based on the actions dispatched to the reducer.

Fetching data with useReduer

To fetch data using useReducer, you'll typically combine it with useEffect to perform asynchronous operations. Here's an example of fetching data using useReducer:

First, let's define the initial state and the reducer function for handling actions related to fetching data:

```
const initialState = {
```

```
  data: null,
  loading: true,
  error: null,
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'FETCH_SUCCESS':
      return { ...state, data: action.payload, loading: false, error: null };
    case 'FETCH_ERROR':
      return { ...state, data: null, loading: false, error: action.payload };
    default:
      throw new Error('Unhandled action');
  }
    };
```

In this example, initialState contains data, loading, and error properties. The reducer function handles two types of actions: FETCH_SUCCESS for successful data fetching and FETCH_ERROR for errors during fetching.

Next, let's use useReducer and useEffect to fetch data in a component:

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

const fetchDataReducer = (state, action) => {
  switch (action.type) {
    case 'FETCH_REQUEST':
      return { ...state, loading: true, error: null };
    case 'FETCH_SUCCESS':
      return { ...state, data: action.payload, loading: false, error: null };
    case 'FETCH_ERROR':
      return { ...state, loading: false, error: action.payload };
    default:
      throw new Error('Unhandled action');
  }
};

const FetchDataExample = () => {
  const [state, dispatch] = useReducer(fetchDataReducer, initialState);
```

```
  useEffect(() => {
    const fetchData = async () => {
      dispatch({ type: 'FETCH_REQUEST' });

      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
        if (!response.ok) {
          throw new Error('Failed to fetch data');
        }
        const data = await response.json();
        dispatch({ type: 'FETCH_SUCCESS', payload: data });
      } catch (error) {
        dispatch({ type: 'FETCH_ERROR', payload: error.message });
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      {state.loading ? (
        <div>Loading...</div>
      ) : state.error ? (
        <div>Error: {state.error}</div>
      ) : (
        <div>
          <h1>Data:</h1>
          <pre>{JSON.stringify(state.data, null, 2)}</pre>
        </div>
      )}
    </div>
  );
};

    export default FetchDataExample;
```

In this example:

We define a fetch Data Reducer function that handles actions related to fetching data.

We use useReducer to manage state based on this reducer function and initial State.
We use useEffect to trigger the data fetching operation when the component mounts ([] dependency array indicates this effect runs once on mount).
Inside useEffect, we define an asynchronous function fetch Data to fetch data from an API and dispatch appropriate actions based on success or failure.
When you render the Fetch Data Example component, it will fetch data from the API and display either the data, a loading message, or an error message based on the state managed by useReducer. Adjust the API URL and error handling logic as per your requirements.

useState vs useReducer

useState and useReducer are two React hooks used for managing state, but they serve different purposes and are suitable for different scenarios. Let's compare useState and useReducer based on their features and use cases:

useState:

Simple State Management: useState is used for managing simple state values, such as numbers, strings, booleans, or objects.

Single State Variable: It manages a single state variable per hook usage.

Update Function: It provides a function to update the state, which can be used to perform state transitions.

Usage: Ideal for managing independent or localized state within a component.

Example:

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

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

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <h1>Count: {count}</h1>
```

```
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};

export default Counter;
```

useReducer:

Complex State Management: useReducer is used for managing more complex state logic, especially when state transitions involve multiple actions.

Reducer Function: It uses a reducer function to handle state updates based on dispatched actions.

Multiple State Variables: It allows managing multiple state variables within a single state object.

Usage: Suitable for managing global or shared state, or when state updates involve complex logic or multiple actions.

Example:

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

const initialState = { count: 0 };

const reducer = (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');
  }
};

const Counter = () => {
```

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

  return (
    <div>
      <h1>Count: {state.count}</h1>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
    </div>
  );
};

export default Counter;
```

In summary, use useState for simple and localized state management within a component, and use useReducer for more complex state management involving multiple state variables and actions, especially when managing global or shared state. Choose the appropriate hook based on the complexity and nature of your state logic.