Module-18 Lists , Hooks , Localstorage , Api Project

1. Lists and Keys

THEORY EXERCISE

1: How do you render a list of items in React? Why is it important to use keys when rendering lists?

Ans: In React, you can render a list of items by using the map() function.

- **Example: import React from 'react';**

```
function Fruitlist() {
        const fruits = ["Banana", "Apple", "Orange", "Mango", "Grapce",
"Chikku"]

 return (
  <div>
            <h2>Fruit List</h2>
            <ul>
              {
                fruits.map((value,index) => (

                 <li key={index}>{value}</li>
                ))
              }
            </ul>
  </div>
 )
}

export default Fruitlist
```

- **Why is it Important to Use Keys When Rendering Lists?**
  - **Importance of Keys:**
    1. **Uniquely Identify Elements:** Keys help React identify which items have changed, been added, or removed.

2. **Improve Performance:** React uses keys to optimize rendering by reusing existing elements rather than re-rendering the entire list.
3. **Avoid Bugs**: Without keys, React may reorder DOM elements incorrectly, leading to UI glitches or unexpected behavior.

2: What are keys in React, and what happens if you do not provide a unique key?

Ans: In React, keys are special string attributes you must include when creating lists of elements using functions like map().

- A **key** is a unique identifier for each item in a list. It helps React identify which items have changed, been added, or removed.
- **<u>Why Are Keys Important?</u>**
  - React uses keys to optimize performance and ensure a smooth UI update. With unique keys,
  - Quickly determine which items in a list changed.
  - Avoid unnecessary re-renders
  - Preserve component state between renders.
- **<u>If You Don't Provide a Unique Key?</u>**
  - ✓ If you don't provide a key, or the keys are not unique, React will:
    - Show a **console warning:**
      - Warning: Each child in a list should have a unique 'key' prop.
    - Re-render the entire list unnecessarily.
    - Possibly cause **UI bugs**, such as**:**
      - Incorrect component state.
      - Input fields losing focus.
      - Unexpected behavior in animations.

2. Hooks(useState, useEffect, useReducer, useRef)

THEORY EXERCISE

1: What are React hooks? How do useState() and useEffect() hooks work in functional components?

Ans: React Hooks are special functions introduced in React 16.8 that allow functional components to use features previously only available in class components, such as state management, lifecycle methods, and context.

- **React Hooks** are functions that allow **functional components** in React to manage **state**, handle side effects, and access other React features without needing class components. They provide a simpler and more efficient way to manage component logic.

❖ useState() Hook

- useState() allows you to add state to a functional component.
- The useState hook is a function that allows you to add state to a functional component. It is an alternative to the useReducer hook that is preferred when we require the basic update. useState Hooks are used to add the state variables in the components.
- **Syntax: const [state, setState] = useState(initialState)**
- **Example: import React, { useState } from 'react';**

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

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

- **How It Works:**
  - **state:** It is the value of the current state.
  - **setState**: It is the function that is used to update the state.

- **initialState:** It is the initial value of the state.

❖ useEffect() Hook

- The **useEffect** in ReactJS is used to handle the side effects such as fetching data and updating DOM. This hook runs on every render but there is also a way of using a dependency array using which we can control the effect of rendering. It can be used to perform actions such as:
  - Fetching data from an API.
  - Setting up event listeners or subscriptions.
  - Manipulating the DOM directly (although React generally handles DOM manipulation for you).
  - Cleaning up resources when a component unmounts.
- **Syntax:**
```
useEffect(() => {
    // Code to run (side effect)
}, [dependencies]);
```
- **Example:**
```
import axios from 'axios'
import React, { useEffect, useState } from 'react'

function Card1() {

  const [data, setdata] = useState([])

  useEffect(() => {
    getCarddata()
  })
  const getCarddata = async () => {
      const res = await
      axios.get("https://dummyjson.com/recipes")
    // console.log(res.data)
    setdata(res.data.recipes)
  }
  return (
```

```
      <div className='container' style={{ display: 'flex',
justifyContent: 'start' }}>
         <div className='row g-3'>
           {
              data && data.map((value, index) => {
                 return (
                   <div className='col-md-3'>
                     <div className="card" style={{ width: '18rem'
}}>
                    <img src={value.image}  width="250px"
height="250px" className="card-img-top" alt="..." />
                 <div className="card-body">
                              <h5 className="card-title"
style={{fontSize:"25px",fontWeight:"600"}}>{value.id}</h5>
                            <h5 className="card-text"
style={{fontWeight:"600"}}>{value.name}</h5>
                              <p className='card-text'
style={{fontWeight:"500"}}>{value.instructions.slice(1,3)}</p>
                               <h5 className="card-text"
style={{fontWeight:"600"}}><i class="fa-solid fa-indian-rupee-
sign"></i> {value.caloriesPerServing}</h5>
                              <h5 className='card-text'
style={{fontWeight:"600"}}><i class="fa-solid fa-globe"></i>
{value.cuisine}</h5>
                              <h5 className="card-text"
style={{fontWeight:"600"}}><i class="fa-solid fa-star"></i><i
class="fa-solid fa-star"></i><i class="fa-solid fa-star"></i><i
class="fa-solid fa-star"></i><i class="fa-solid fa-star-half-
stroke"></i> {value.rating}</h5>
                              <button className='btn btn-danger mt-2
ms-5' style={{fontWeight:"600"}}>ORDER NOW</button>
                            </div>
                          </div>
                        </div>
                    )
```

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

export default Card1
```

- **How It Works:**
  - Runs after render — React runs useEffect *after* the component has rendered.
  - Dependencies control when it runs:
    - [] → Runs once after the first render (componentDidMount).
    - [someVar] → Runs when someVar changes.
    - *No array* → Runs after every render.

2: What problems did hooks solve in React development? Why are hooks considered an important addition to React?

Ans: React Hooks were introduced in React 16.8 to solve several key problems that developers faced with class components.

- **Problems Solved by Hooks in React**

1. **Complexity in Managing State and Side Effects in Class Components**
   - ✓ **Before Hooks**: You had to use class components (class MyComponent extends React.Component) to use state or lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.
   - ✓ **Problem:** Class components could become verbose and hard to read, especially with complex state logic split across multiple lifecycle methods.
   - ✓ **With Hooks:** You can manage state and side effects directly in functional components using useState, useEffect, etc.
2. **Reusing Logic Between Components Was Difficult**
   - ✓ **Before Hooks**: Reusing stateful logic required higher-order components (HOCs) or render props, which led to:

        - Wrapper hell (nested functions/components)
        - Less readable and harder to debug code
    - ✓ **With Hooks:** You can create **custom hooks** to reuse logic cleanly across components without changing the component hierarchy.
3. **Confusing this Keyword**
    - ✓ In **Class Components:** You had to bind methods to this, and beginners often faced issues with this not being set correctly.
    - ✓ **With Hooks:** Functional components with hooks avoid the this keyword entirely, simplifying the syntax and reducing bugs.
4. **Tight Coupling of Related Logic**
    - ✓ Lifecycle methods (like componentDidMount) often contained unrelated logic, and related logic was scattered across different lifecycle methods.
    - ✓ **With Hooks:** Related pieces of logic (like setting up a subscription and cleaning it up) can be grouped together in a single useEffect.

➢ Why Hooks Are an Important Addition to React

- **Simplified Codebase**
    - ✓ Hooks allow you to write less code, more readable and maintainable.
- **Functional Programming Style**
    - ✓ Encourages a functional approach, which aligns with JavaScript's trend toward functions-first programming.
- **Improved Code Reusability**
    - ✓ Custom hooks allow logic sharing without altering component structure.
- **Encouraged Best Practices**
    - ✓ Hooks promote separation of concerns and side-effect isolation.
- **Easier Learning Curve**
    - ✓ Once you understand JavaScript functions and closures, hooks feel more natural than learning the this context and lifecycle methods.

3: What is useReducer ? How we use in react app?

Ans: useReducer is a React Hook that lets you add a reducer to your component.

- The **useReducer** hook is an alternative to the useState hook that is preferred when you have complex state logic. It is useful when the state transitions depend on previous state values or when you need to handle actions that can update the state differently.
- **Syntax: const [state, dispatch] = useReducer(reducer, initialState);**
- **How It Work:**
  - **reducer:** A function that defines how the state should be updated based on the action. It takes two parameters: the current state and the action.
  - **initialState:** The initial value of the state.
  - **State** The current state returned from the useReducer hook.
  - **dispatch:** A function used to send an action to the reducer to update the state.
- **Example:**

```
import React, { useReducer } from 'react'
import Headers from '../layout/Comana/Headers';

let initialstate = 0
export const reducer=(state,action)=>{
   switch (action) {
      case 'increment':
         return state + 1;
         break;
      case 'decrement':
         return state - 1;
         break;
      default:
       return initialstate
         break;
   }
}
function Usered() {

   const [count,dispatch] = useReducer(reducer,initialstate)
```

```
    console.log(count)

    return (
    <div>
     <Headers />

     <h1>hello this Usereducer hooks</h1>
     <h1>Hello count : {count}</h1>
     <button
  onClick={()=>dispatch('increment')}>increment</button>
     <button
  onClick={()=>dispatch('decrement')}>decrement</button>
    </div>
   )
  }
export default Usered
```

4: What is useRef ? How to work in react app?

Ans: useRef is a React Hook that lets you reference a value that's not needed for rendering.

- The **useRef Hook** is a built-in React Hook that returns a mutable reference object (ref) that persists across renders. Unlike state variables, updating a ref does not trigger a component re-render.
- **Syntax: const refContainer = useRef(initialValue);**
- **How It Works:**
  - useRef returns an object { current: initialValue }.
  - The .current property can be updated without re-rendering the component.
- **Example:**

```
import { useRef } from 'react';
export default function Counter() {
 let ref = useRef(0);

 function handleClick() {
  ref.current = ref.current + 1;
```

```
    alert('You clicked ' + ref.current + ' times!');
  }

  return (
    <button onClick={handleClick}>
      Click me!
    </button>
  );
}
```