**Lists And Hooks**

**THEORY EXERCISE**

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

**Answer:**

In React, you render a list by using the JavaScript map() function to iterate over an array and return a React element for each item.

**Example:**

const fruits = ['Apple', 'Banana', 'Mango'];

const listItems = fruits.map(fruit => <li>{fruit}</li>);

You use this listItems in JSX like:

<ul>{listItems}</ul>

**Why are keys important?**

Keys help React **identify which items have changed, been added, or removed**. This improves **performance** and prevents **unnecessary re-renders**.

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

**Answer:**

**Keys** are special string attributes that help React identify which elements in a list are changed, updated, or deleted.

**Good Key:**

* Must be **unique** among siblings.
* Should ideally be **stable** (not change over time).
* Common choice: a **unique ID** from the data.

**If you don’t provide a unique key:**

* React will fallback to using the index as a key.
* It may lead to **incorrect UI updates**, especially during item reordering or deletion.

**LAB EXERCISE**

**Task 1: o Create a React component that renders a list of items (e.g., a list of fruit names). Use the map() function to render each item in the list.**

import React from 'react';

function FruitList() {

  const fruits = ['Apple', 'Banana', 'Mango', 'Orange', 'Grapes'];

  return (

    <ul>

      {fruits.map((fruit, index) => (

        <li key={index}>{fruit}</li> // Note: index used only if no unique id

      ))}

    </ul>

  );

}

export default FruitList;

**Task 2: o Create a list of users where each user has a unique id. Render the user list using React and assign a unique key to each user.**

  import React from 'react';

function UserList() {

  const users = [

    { id: 101, name: 'Amit' },

    { id: 102, name: 'Sita' },

    { id: 103, name: 'Rahul' },

    { id: 104, name: 'Neha' }

  ];

  return (

    <ul>

      {users.map(user => (

        <li key={user.id}>{user.name}</li> // Unique key used here

      ))}

    </ul>

  );

}

export default UserList;

**Hooks**

**THEORY EXERCISE**

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

**Answer:**

**Hooks** are special functions in React that let you **use state and lifecycle features** in **functional components** without writing a class.

#### useState(): Allows you to add ****state**** to a functional component.

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

#### useEffect(): Handles ****side effects**** like data fetching, subscriptions, or manually changing the DOM.

useEffect(() => {

  // runs on mount and update

}, []);

### Question 2: What problems did hooks solve in React development? Why are hooks important?

**Answer:**

**Hooks solved:**

* Code **reuse issues** between components.
* Complexity of managing **stateful logic** across classes.
* Need for **lifecycle methods** in functional components.

**Importance of Hooks:**

* Enables **functional components to be stateful**.
* Makes code **cleaner**, **modular**, and **easier to test**.
* **No need for class components** for state and lifecycle.

### Question 3: What is useReducer? How is it used in a React app?

**Answer:**

useReducer() is a hook used for **complex state logic**, often preferred over useState() when:

* The next state depends on the **previous state**.
* State has **multiple sub-values**.

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

function reducer(state, action) {

  switch (action.type) {

    case 'increment':

      return { count: state.count + 1 };

    default:

      return state;

  }

}

### Question 4: What is the purpose of useCallback & useMemo hooks?

**Answer:**

| **Hook** | **Purpose** |
| --- | --- |
| useCallback | Returns a **memoized callback function**. Prevents function re-creation on every render. |
| useMemo | Returns a **memoized value**. Prevents expensive recalculations. |

### Question 5: What’s the Difference between useCallback & useMemo?

| **Feature** | **useCallback(fn, deps)** | **useMemo(fn, deps)** |
| --- | --- | --- |
| Returns | Memoized **function** | Memoized **value** |
| Use Case | Pass stable functions to children | Cache computed values |

### **Question 6: What is useRef? How does it work?**

**Answer:**

useRef() is a hook that:

* Creates a **mutable object** that persists across renders.
* Does **not cause re-renders** when the ref value changes.
* Can reference a DOM element or hold any mutable value.

const inputRef = useRef();

<input ref={inputRef} />

**LAB EXERCISE**

**Task 1: Create a functional component with a counter using the useState() hook. Include buttons to increment and decrement the counter.**

import React, { useState } from 'react';

function Counter() {

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

  return (

    <div>

      <h2>Count: {count}</h2>

      <button onClick={() => setCount(count + 1)}>Increment</button>

      <button onClick={() => setCount(count - 1)}>Decrement</button>

    </div>

  );

}

export default Counter;

**Task 2: Use the useEffect() hook to fetch and display data from an API when the component mounts.**

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

function DataFetcher() {

  const [users, setUsers] = useState([]);

  useEffect(() => {

    fetch('https://jsonplaceholder.typicode.com/users')

      .then(res => res.json())

      .then(data => setUsers(data));

  }, []); // Empty dependency = only on mount

  return (

    <ul>

      {users.map(user => <li key={user.id}>{user.name}</li>)}

    </ul>

  );

}

export default DataFetcher;

**Task 3: Create react app with use of useSelector & useDispatch.**

**Task 4: Create react app to avoid re-rendersin react application by useRef ?**

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

function RefComponent() {

  const renderCount = useRef(1);

  const [text, setText] = useState('');

  const handleChange = (e) => {

    setText(e.target.value);

    renderCount.current += 1;

  };

  return (

    <div>

      <input value={text} onChange={handleChange} />

      <p>Render Count (tracked by useRef): {renderCount.current}</p>

    </div>

  );

}

export default RefComponent;

Here, the **component doesn't re-render when** renderCount **changes** because useRef doesn't cause re-renders.