React JS

Module 15) Lists , Hooks , Localstorage , Api Project

**Lists and Keys**

THEORY EXERCISE

Here are the **answers** for your next **React.js theory exercise** on **lists and keys**:

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

**Rendering a List in React:**

To render a list in React, you usually use **JavaScript’s map() method** to loop over an array and return JSX for each item.

**Example:**

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

function FruitList() {

return (

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

))}

</ul>

);

}

**Why are keys important?**

* Keys **help React identify** which items have changed, are added, or removed.
* They improve the **performance** of rendering lists.
* Keys help React **avoid unnecessary re-renders**.

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

**What are Keys?**

* A **key** is a **special string attribute** you must include when creating lists of elements in React.
* It should be **unique** among siblings.
* It’s not visible in the UI — it’s used internally by React.

**Example:**

<li key={item.id}>{item.name}</li>

**What happens if you don’t provide a unique key?**

* React **can’t efficiently track** element changes.
* It may **re-render the wrong items**, leading to:
  + Unexpected behavior (e.g., incorrect form input values).
  + **Performance issues.**
* React might show a **warning** in the console.

**Best Practices for Keys:**

* Use a **unique ID** if available (e.g., user.id, product.code).
* **Avoid using index as a key**, especially if the list changes order or items are added/removed.

LAB EXERCISE

• Task 1:

• Create a React component that renders a list of items (e.g., a list of fruit names).

Usethe map()function to render each item in the list.

• Task 2:

• Create a list of users where each user has a unique id. Render the user list

usingReact and assign a unique keyto each user.

**Forms in React**

THEORY EXERCISE

**Question 1: How do you handle forms in React? Explain the concept of controlled components.**

In React, **forms are handled using state**, meaning you store form values in a component's state and update them via event handlers. This allows full control over form inputs.

**Controlled Components:**

A **controlled component** is an input element (like <input>, <textarea>, <select>) whose value is managed by React state.

**Example:**

import React, { useState } from "react";

function MyForm() {

const [name, setName] = useState("");

const handleChange = (e) => {

setName(e.target.value);

};

const handleSubmit = (e) => {

e.preventDefault();

alert("Submitted name: " + name);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" value={name} onChange={handleChange} />

<button type="submit">Submit</button>

</form>

);

}

**Advantages of Controlled Components:**

* Validation is easier
* Instant updates
* Single source of truth (React state)

**Question 2: What is the difference between controlled and uncontrolled components in React?**

| **Feature** | **Controlled Component** | **Uncontrolled Component** |
| --- | --- | --- |
| Data Source | React State | DOM (input’s own internal state) |
| Uses value & onChange | Yes | No |
| Access via | useState, this.state | useRef |
| Controlled by React? | Fully | No, React reads value when needed |
| Best For | Forms needing validation or dynamic behavior | Simple/legacy forms |

**📌 Example of Uncontrolled Component:**

import React, { useRef } from "react";

function MyForm() {

const nameInput = useRef();

const handleSubmit = (e) => {

e.preventDefault();

alert("Submitted name: " + nameInput.current.value);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" ref={nameInput} />

<button type="submit">Submit</button>

</form>

);

}

LAB EXERCISE

• Task 1:

• Create a form with inputs for name, email, and password. Use state to control the

form and display the form data when the user submits it.

• Task 2:

• Add validation to the form created above. For example, ensure that the emailinput

contains a valid email address.

**Lifecycle Methods (Class Components)**

THEORY EXERCISE

**Question 1: What are lifecycle methods in React class components? Describe the phases of a component’s lifecycle.**

**Lifecycle methods** in React class components are special methods that get called at specific stages of a component’s existence. They allow developers to run code at particular points during the component's **creation, update, and removal**.

**The 3 Main Phases of a Component's Lifecycle:**

| **Phase** | **Description** |
| --- | --- |
| **Mounting** | When the component is being inserted into the DOM for the first time. |
| **Updating** | When the component is re-rendered due to state or prop changes. |
| **Unmounting** | When the component is being removed from the DOM. |

**Question 2: Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().**

**🔹 componentDidMount()**

* **Phase:** Mounting
* **Purpose:** Runs once after the component is inserted into the DOM.
* **Use case:** Fetching data from APIs, setting up subscriptions, initializing third-party libraries.

componentDidMount() {

console.log("Component mounted");

// Example: fetch data

}

**🔹 componentDidUpdate(prevProps, prevState)**

* **Phase:** Updating
* **Purpose:** Runs after every update (except initial render).
* **Use case:** Reacting to changes in props or state (e.g., re-fetch data if a prop changes).

componentDidUpdate(prevProps, prevState) {

if (this.props.userId !== prevProps.userId) {

// Fetch new data

}

}

**🔹 componentWillUnmount()**

* **Phase:** Unmounting
* **Purpose:** Runs before the component is removed from the DOM.
* **Use case:** Clean up operations (like clearing timers, unsubscribing from events).

componentWillUnmount() {

console.log("Component is being removed");

// Cleanup code

}

LAB EXERCISE

• Task 1:

• Create a class component that fetches data from an API when the

componentmounts using componentDidMount(). Display the data in the

component.

• Task 2:

• Implement a component that logs a message to the console when it updates

using componentDidUpdate(). Log another message when the component

unmountsusing componentWillUnmount().

**Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)**

THEORY EXERCISE

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

**React Hooks** are special functions introduced in React 16.8 that allow **functional components** to use state and other React features that were previously only available in **class components**.

**🔹 useState()**

* Adds **state** to functional components.
* Returns an array: [currentValue, setValueFunction].

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

**🔹 useEffect()**

* Allows you to perform **side effects** (e.g., data fetching, subscriptions, DOM updates).
* Runs after rendering by default.

useEffect(() => {

console.log("Component rendered or updated");

}, [dependency]);

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

**🔹 Problems solved by Hooks:**

1. **Code Reuse**: Logic could not be reused between components easily in class components.
2. **Complex Components**: Stateful logic was spread across lifecycle methods.
3. **Class Confusion**: this keyword and binding created confusion for beginners.
4. **Better Organization**: Hooks allow separation of concerns by logic, not lifecycle methods.

**🔹 Why important?**

* Enables writing **cleaner**, **more modular**, and **readable** functional components.
* Encourages **composition** of logic using custom hooks.

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

* useReducer is an alternative to useState for managing **complex state logic**.
* It works like a Redux reducer: takes a **state** and **action**, returns a **new state**.

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: return state;

}

}

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

**Question 4: What is the purpose of useCallback() and useMemo() Hooks?**

**🔹 useCallback(fn, deps)**

* Returns a **memoized callback function**.
* Useful when passing functions to child components to **prevent unnecessary re-renders**.

const handleClick = useCallback(() => {

console.log("Clicked");

}, []);

**🔹 useMemo(factoryFn, deps)**

* Returns a **memoized value**.
* Useful for expensive calculations that shouldn't be repeated on every render.

const result = useMemo(() => expensiveFunction(input), [input]);

**Question 5: What’s the Difference Between useCallback() and useMemo()?**

| **Feature** | **useCallback()** | **useMemo()** |
| --- | --- | --- |
| Returns | Memoized **function** | Memoized **value** |
| Usage | Optimize function references | Optimize expensive calculations |
| Syntax | useCallback(fn, deps) | useMemo(() => compute, deps) |

**Question 6: What is useRef? How does it work in a React app?**

* useRef creates a **mutable reference** that persists across renders.
* Commonly used for:
  + Accessing DOM elements.
  + Storing mutable variables that don’t trigger re-renders.

const inputRef = useRef();

const focusInput = () => {

inputRef.current.focus();

};

return <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.

• Task 2:

• Use the useEffect()hook to fetch and display data from an API when the

component mounts.

• Task 3:

• Create react app with use of useSelector & useDispatch.

• Task 4:

• Create react app to avoid re-renders in react application by useRef ?