**Module:Lists , Hooks , Localstorage , Api Project**

**Question 1: How Do You Render a List of Items in React?**

**Ans -**

In **React**, you can render a list of items using the **.map()** method. This method iterates over an array and returns **React elements** for each item.

**Example: Rendering a List in React**

import React from 'react';

const ItemList = () => {

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

return (

<ul>

{items.map((item, index) => (

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

))}

</ul>

);

};

export default ItemList;

**Explanation:**

1. **items** – An array of data.
2. **.map()** – Iterates over the array.
3. **<li>** – Each item is rendered as a list element.
4. **key={index}** – Provides a **unique key** for each item (explained below).

**Why Is It Important to Use Keys When Rendering Lists?**

Keys help **React identify** which items:

1. **Changed** (updated content).
2. **Added** (new items).
3. **Removed** (deleted items).

Without **keys**, React cannot efficiently update the **DOM**, leading to:

* Performance issues.
* Incorrect UI updates.

**Keys** make React's **reconciliation process** faster and more efficient.

**Question 2: What Are Keys in React?**

Ans - A **key** is a special **attribute** used to give each element in a list a **unique identifier**.

**Example with Unique Keys:**

const users = [

{ id: 1, name: 'John' },

{ id: 2, name: 'Jane' },

];

return (

<ul>

{users.map((user) => (

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

))}

</ul>

);

Here, user.id is used as the **key**, ensuring React can track each item.

**What Happens If You Don’t Provide a Unique Key?**

If you don’t provide a **key** or use **non-unique keys**:

1. **Performance Issues:** React will re-render the **entire list** instead of just changed items.
2. **Incorrect UI Updates:** React may mix up **state** between components.
3. **Warnings:** React will log this error:
4. Warning: Each child in a list should have a unique "key" prop.

**Best Practices for Using Keys in React:**

1. **Use Unique IDs**: If available, prefer a **unique identifier** (e.g., id from a database).
2. **Avoid Using Index as Key**: Only use index as a key if the list will **never change** (not recommended for dynamic data).
3. **Stable Keys**: Ensure keys are **consistent** across renders.

**Question 3: How Do You Handle Forms in React?**

Ans - In **React**, forms are handled using **state** to manage input values. This allows you to capture user input, validate data, and control the form behavior dynamically.

**Controlled Components Concept:**

A **controlled component** in React is a form element (**input, textarea, select, etc.**) where the **React state** manages the input’s value.

* **Input value** is stored in the **component’s state**.
* Any change triggers the **onChange** event, updating the state.
* React controls the entire form’s behavior.
* **Why Use Controlled Components?**

1. **Data Synchronization:** Keeps form data in sync with the **React state**.
2. **Validation:** Allows **real-time** input validation.
3. **Consistency:** Ensures a **single source of truth** for the form’s data.

**Question 4: Difference Between Controlled and Uncontrolled Components in React**

Ans -

| **Feature** | **Controlled Component** | **Uncontrolled Component** |
| --- | --- | --- |
| **Definition** | Form inputs controlled by **React state**. | Form inputs managed by the **DOM itself**. |
| **Data Handling** | Data is stored in **React state**. | Data is stored in the **DOM** (not React). |
| **Value Control** | Controlled via the **value** prop. | Accessed via **Refs** (e.g., useRef()). |
| **Example Usage** | When you need **form validation** and real-time updates. | For **simple forms** where control isn’t required. |
| **Performance** | Slightly **slower** due to continuous state updates. | More **performant** for large forms as no state updates are needed. |
| **Best For** | **Complex** forms requiring dynamic input handling. | **Basic** forms where manual control isn’t necessary. |

**In Summary:**

* **Controlled Components**: React is the **single source of truth** for form data.
* **Uncontrolled Components**: Data is handled directly by the **DOM**.

**Question 5: What Are React Hooks? How Do useState() and useEffect() Work in Functional Components?**

**Ans - React Hooks** are special functions introduced in **React 16.8** that allow you to use **state** and other **React features** in **functional components** (without writing class components).

**useState() Hook:**

* **Purpose:** Allows you to **add and manage state** in functional components.
* **How It Works:**
  + Takes an **initial value** and returns an **array** with:
    1. **Current state value**
    2. **Function to update the state**
  + **State changes** trigger a **re-render** of the component.

**useEffect() Hook:**

* **Purpose:** Performs **side effects** (e.g., data fetching, subscriptions, DOM updates).
* **How It Works:**
  + Accepts two arguments:
    1. **Callback function** – Runs after rendering.
    2. **Dependency array** – Controls when the effect runs.

**Question 6: What Problems Did Hooks Solve in React Development? Why Are Hooks Considered Important?**

**Ans -**

**Problems Hooks Solved:**

1. **Complex State Logic:** Hooks like useState() and useReducer() simplify managing **complex state** without switching to class components.
2. **Reusability:** Hooks enable **reusing** logic via **custom hooks**, reducing code duplication.
3. **Component Bloat:** Hooks make **functional components** powerful, reducing reliance on **class components**.
4. **Lifecycle Issues:** useEffect() replaces lifecycle methods (e.g., componentDidMount, componentDidUpdate, and componentWillUnmount).

**Why Hooks Are Important:**

1. **Simplified Code:** Cleaner and easier to understand.
2. **Improved Performance:** Optimize with hooks like useMemo() and useCallback().
3. **Better Composition:** Logic can be split into **smaller, reusable** pieces.
4. **Functional First:** Encourages the use of **functional programming** in React.

**Question 7: What Is useReducer()? How Is It Used in a React App?**

**Ans -**

**useReducer()** is a **React hook** used to manage **complex state logic** in functional components.

**Purpose:**

* Alternative to **useState()** for managing **complex** state transitions.
* Useful when the **next state** depends on the **previous state**.

**How It Works:**

* Accepts two parameters:
  1. **Reducer function** – Defines how the state changes.
  2. **Initial state** – The starting state value.
* Returns an **array** with:
  1. **Current state**.
  2. **Dispatch function** – Triggers state updates.

**Question 8: What Is the Purpose of useCallback() & useMemo() Hooks?**

**Ans -**

**useCallback() Hook:**

* **Purpose:** **Memoizes** a function to prevent **re-creation** on each render.
* **When to Use:** When passing functions as **props** to **child components**.

**useMemo() Hook:**

* **Purpose:** **Memoizes** a **value** (or computed result) to optimize performance.
* **When to Use:** For **expensive computations** that should **not** re-run on every render.

**Question 9: What’s the Difference Between useCallback() & useMemo() Hooks?**

**Ans -**

| **Feature** | **useCallback()** | **useMemo()** |
| --- | --- | --- |
| **Purpose** | Memoizes a **function** | Memoizes a **computed value** |
| **Return Value** | **Memoized function** | **Memoized result** (e.g., object, array) |
| **Use Case** | Avoid function re-creation on re-renders | Optimize heavy computations |
| **Example Usage** | Passing **callback** functions to children | Caching **derived** values (e.g., sums) |

**Question 10: What Is useRef()? How Does It Work in a React App?**

**Ans -**

**Purpose:**

* Provides a way to **reference DOM elements** or store **mutable values** without causing re-renders.

**How It Works:**

* Returns an object: { current: initialValue }.
* **Does not** trigger re-renders when updated.

**Common Use Cases:**

1. **Accessing DOM Elements:** Directly interact with HTML elements (e.g., focus an input).
2. **Persistent Variables:** Store values that persist across renders (e.g., timers).
3. **Avoid Re-Renders:** Keep mutable values without causing re-renders.

**Question 11: What is the Context API in React? How Is It Used to Manage Global State Across Multiple Components?**

**Ans -**

The **Context API** in **React** is a built-in system for **sharing state** (data) between multiple components **without** passing props through every level of the component tree (known as **prop drilling**).

**Purpose of Context API:**

* **Global State Management:** Share **common data** (e.g., user authentication, themes) across many components.
* **Avoid Prop Drilling:** Pass data **directly** to any child component, **skipping** intermediate levels.
* **Simplify State Sharing:** Ideal for **medium-sized** applications where **Redux** may be too complex.

**How Context API Works:**

1. **Create Context** – Use **createContext()** to define a **data container**.
2. **Provide Context** – Use the **Provider** component to **share** data.
3. **Consume Context** – Access the shared data with the **useContext()** hook.

**Common Use Cases for Context API:**

* **Themes:** Light mode / Dark mode.
* **User Authentication:** Logged-in user data.
* **Localization:** Language and region settings.
* **Global State:** Cart items in an e-commerce site.

**Question 12: Explain How createContext() and useContext() Are Used in React for Sharing State.**

**createContext() Function:**

* **Purpose:** Creates a new **Context object** for managing global state.
* **Returns:**
  + **Provider**: Supplies data to the component tree.
  + **Consumer**: (Optional) Legacy method to consume data.

**useContext() Hook:**

* **Purpose:** Accesses the **context value** inside functional components.
* **Returns:** The **current context value** provided by the nearest **Context Provider**.

**Workflow of Using Context API:**

1. **Create Context:** Define a context object using createContext().
2. **Provide Context:** Wrap the required components with a **Provider**.
3. **Consume Context:** Use **useContext()** to access the **shared state**.
4. **When to Use Context API:**

* Suitable for **moderate global state**.
* For **large** and **complex** applications, **Redux** or **React Query** might be better.