**List**

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

* To render a list of items in React, we typically use the map() function to iterate over an array of data and return a JSX element for each item.
* Ex :

const fruits = ['apple', 'banana', 'orange'];

 return

  ( <ul>

  {fruits.map(fruit => (

  <li key={fruit}>{fruit}</li>

  ))}

</ul>

 );

* **Why keys are important:**
* **Efficient Rendering:** Keys help React identify which items have changed, added, or removed. This allows React to perform efficient updates to the DOM, minimizing unnecessary re-renders.
* **Consistent UI:** Keys ensure that React can correctly track the identity of each item in the list, even if the order of the items changes. This helps maintain the correct state and behavior of each item.
* **Correct Event Handling:** Keys are crucial for event handlers to work correctly, especially when dealing with dynamic lists. Without keys, React might not be able to associate events with the correct items, leading to unexpected behavior.

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

* Keys are unique identifiers assigned to each item in a list. They should be stable and unique within the list. React uses keys to:
* **Identify items:** React uses keys to track the identity of each item, even if the order or content of the items changes.
* **Optimize updates:** React can efficiently update the DOM by comparing the keys of the new and old lists.
* **Handle events:** Keys help React correctly associate event handlers with the correct items.
* **What happens without unique keys:**
* **Incorrect updates:** React may not be able to correctly identify which items have changed, leading to incorrect updates and unexpected behavior.
* **Performance issues:** Without keys, React might have to re-render the entire list, even if only a few items have changed, which can impact performance.
* **Event handling issues:** Event handlers might not work as expected, especially when dealing with dynamic lists.

**Hooks**

* 1. What are React hooks? How do useState() and useEffect() hooks work in functional components?
* React Hooks are special functions that let you use state and other React features in functional components. Before hooks, we had to use class components to manage state and side effects. Hooks make functional components more powerful and flexible.
* **useState Hook**
* **Purpose:** To manage state within a functional component.
* **How it works:**
  1. You call useState with an initial value.
  2. It returns an array with two values:
     + The current state value.
     + A function to update the state.

import { useState } from 'react';

function Counter() {

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

  return (

    <div>

      <p>Count: {count}</p>

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

    </div>

  );

}

* **useEffect Hook**
* **Purpose:** To perform side effects in functional components.
* **How it works:**
  1. You call useEffect with a function that contains the side effect.
  2. The effect will run after the component renders.
  3. You can optionally provide a dependency array to control when the effect runs.

import { useState, useEffect } from 'react';

function DataFetcher() {

  const [data, setData] = useState(null);

useEffect(() => {

    fetch('https://api.example.com/data')

      .then(response => response.json())

      .then(data => setData(data));

  }, []); // Empty dependency array: runs only once on mount

  return (

    <div>

      {data ? (

        <p>Data: {data}</p>

      ) : (

        <p>Loading...</p>

      )}

    </div>

  );

}

* 1. What problems did hooks solve in React development? Why are hooks considered an important addition to React?
* Before hooks, React primarily relied on class components to manage state and side effects. While this approach worked, it had some drawbacks:
* **Complex Component Structure:** Class components often became bloated and difficult to understand, especially for larger components with multiple state variables and side effects.
* **Code Reusability:** Sharing stateful logic between components was challenging, leading to code duplication and less maintainable codebases.
* **Steep Learning Curve:** Understanding class components, their lifecycle methods, and the intricacies of this binding could be daunting for newcomers.
* **Why Hooks are Important:**
* **Simplified Component Structure:** Hooks allow you to manage state and side effects directly within functional components, making them more concise and easier to reason about.
* **Enhanced Code Reusability:** Custom hooks enable you to extract reusable stateful logic into independent functions, promoting code modularity and reducing boilerplate.
* **Improved Readability:** Hooks often lead to more readable and maintainable code by breaking down complex logic into smaller, focused functions.
* **Steeper Learning Curve:** Hooks provide a more intuitive and beginner-friendly way to learn React, as they align more closely with functional programming principles.

* 1. What is useReducer ? How we use in react app?
* The useReducer hook is a powerful tool in React for managing complex state logic, especially when you have multiple state values that are interdependent or when you need to handle complex state transitions.

import { useReducer } from 'react';

function Counter() {

  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:

        return state;

    }

  };

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

  return (

    <div>

      <p>Count: {state.count}</p>

      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>

      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>

    </div>

  );

}

* 1. What is the purpose of useCallback & useMemo Hooks?
* **useCallback and useMemo** are React Hooks that help optimize performance by preventing unnecessary re-renders.
* **useCallback** memoizes functions, ensuring that child components only re-render when the function's dependencies change. This is useful for passing callback functions as props to child components.
* **useMemo** memoizes values, avoiding redundant calculations. It's ideal for expensive computations or data transformations that don't need to be recalculated unless their dependencies change.
  1. What’s the Difference between the useCallback & useMemo Hooks?

|  |  |
| --- | --- |
| useCallback | useMemo |
| Returns a memorized callback | Returns a memorized value |
| it returns referential equality between renders for functions | it returns referential equality between renders for value |
| It returns function when the dependencies change | It calls the function when the value changes and return result |

* 1. What is useRef ? How to work in react app?
* The useRef hook is a useful tool in React for creating a mutable reference object that persists across renders. It's often used for:
* **Persisting values across renders:**
* **Accessing DOM elements:**
* **Creating custom hooks:**

import { useRef, useEffect } from 'react';

function UseRefExample() {

  const inputRef = useRef(null);

  useEffect(() => {

    inputRef.current.focus();

  }, []);

  return (

    <div>

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

    </div>

  );

}