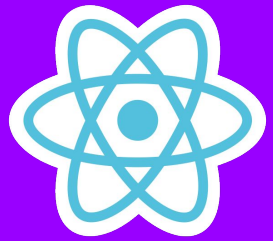

React - Conditional Rendering, Lists & Hooks

Introduction to ReactJS (Part II of a Series)

Instructors: Reuben Devanesan, Kanishk Singhal, Ananthu JP

Setting up a React App. Rendering Elements

- **Conditional Rendering**
- Lists map
- Hooks - Intro
- useState
- useEffect
- Custom hook

Conditional Rendering

It is the technique of rendering different content or components based on certain conditions.

Setting up a React App. Rendering Elements

- **Conditional Rendering**
- Lists map
- Hooks - Intro
- useState
- useEffect
- Custom hook

Following code will render Component 1 or 2 based on the condition A less than or greater than 5

```
const Component1 = () => {
  return <div>Number is greater than 5</div>;
};
const Component2 = () => {
  return <div>Number is less than 5</div>;
};

function ConditionalRender() {
  const A = prompt("Enter a number");
  return (
    <div className="w-screen h-screen flex flex-col bg-yellow-100">
      <h1 className="text-xl font-bold ">Conditional Render</h1>
      {A > 5 ? <Component1 /> : <Component2 />}
    </div>
  );
}
export default ConditionalRender;
```

Setting up a React App. Rendering Elements

- Conditional Rendering
- **Lists map**
- Hooks - Intro
- useState
- useEffect
- Custom hook

Map function

```
array.map(callback(currentValue, index, array), thisArg);
```

The **map()** function is a JavaScript method used for iterating over arrays and rendering a component for each element of the array. It is often used to transform and manipulate data before rendering it in the user interface.

Setting up a React App. Rendering Elements

- Conditional Rendering
- **Lists map**
- Hooks - Intro
- useState
- useEffect
- Custom hook

Map function (example)

Following code will render contents of the array using map function.

```
export default function Lists() {
  const fruits = ["Apple", "Orange", "Mango", "Water Melon"];
  return (
    <div className=" w-screen h-screen flex flex-col">
      <h1 className="text-xl font-bold">Lists</h1>
      <ol>
        {fruits.map((item) => (
          <li>{item}</li>
        ))}
        {/* <li>{fruits[0]}</li>
        <li>{fruits[1]}</li>
        <li>{fruits[2]}</li>
        <li>{fruits[3]}</li> */}
      </ol>
    </div>
  );
}
```

Setting up a React App. Rendering Elements

- Conditional Rendering
- Lists map
- **Hooks - Intro**
- useState
- useEffect
- Custom hook

Hooks - Introduction

- In React.js, hooks are functions that let developers "**hook into**" React **state and lifecycle features** from function components.
- Before the introduction of **hooks in React 16.8**, state and other React features were only available in class components.
- **Hooks** provide a way to use state and other React features **without writing a class**.
- React comes with several built-in hooks that allow developers to manage **state, perform side effects**, and handle other aspects of component behavior. Some of the most commonly used hooks are:

    1. useState
    2. useEffect

Setting up a React App. Rendering Elements

useState Hook

The **useState hook** allows functional components to manage local state. It returns an array with two elements: the **current state value and a function** to update it.

```
const [state, setState] = useState(initialState);
```

state: This is the current state value, and it's similar to the this.state in class components.

setState: This is a function that allows you to update the state value. It's similar to this.setState in class components.

initialState: This is the initial value of the state.

Setting up a React App. Rendering Elements

- Conditional Rendering
- Lists map
- Hooks - Intro
- **useState**
- useEffect
- Custom hook

useState Hook (example)

Simple counter component using the useState hook in React

```jsx
import React, { useState } from "react";

export default function StateHook() {
  const [count, setCount] = useState(0);

  return (
    <div className="w-screen h-screen flex flex-col">
      <h1 className="text-xl font-bold">useState Hook - Counter example</h1>
      <h1 className="text-5xl font-bold">{count}</h1>
      <div className="flex flex-row gap-3">
        <button
          onClick={() => setCount(count - 1)}
          className="w-8 h-8 bg-gradient-to-br from-red-300 to-red-500"
        >
          -
        </button>
        <button
          onClick={() => setCount(count + 1)}
          className="w-8 h-8 bg-gradient-to-br from-green-300 to-green-500"
        >
          +
        </button>
      </div>
    </div>
  );
}
```

Setting up a React App. Rendering Elements

- Conditional Rendering
- Lists map
- Hooks - Intro
- useState
- **useEffect**
- Custom hook

useEffect Hook

The **useEffect** hook allows you to perform side effects in function components. It runs after every render and can be used for **data fetching, subscriptions, manually changing the DOM,** and more.

```
useEffect(() => {
  // Side effect code or effect function
  // This code runs after every render

  // Clean-up code (optional)
  // This code runs when the component is unmounted or before the next effect
runs

  return () => {
    // Clean-up code
  };
}, [dependencies]);
```

useEffect takes two arguments: the first one is the effect function, and the second one is an optional array of dependencies.

The effect function contains the code that you want to run after the component renders. It may contain asynchronous operations, subscriptions, or other side effects.

The optional dependencies array is an array of values (**typically props or state variables**) that the effect depends on. If any of the values in the dependencies array change between renders, the effect function will run again. If the dependencies array is empty, the effect will only run after the initial render.

Setting up a React App. Rendering Elements

useEffect - (componentDidUpdate) example

Adding the following useEffect snippet to the counter example problem, it will **log the count value to console** whenever the **count value is updated**

```
useEffect(() => {
  console.log(count);
}, [count]);
```

Setting up a React App. Rendering Elements

useEffect - Component

```
useEffect(() => {
  //Component Did Mount?
  console.log("Component is mounted");

  return () => {
    //Component Did Un-Mount?
    console.log("Component is un-mounted");
  };
}, []);
```

componentDidMount (Component Did Mount) : In class components, componentDidMount is called after the component is rendered for the first time. Similarly, you can achieve this behavior in functional components using useEffect with an empty dependency array:

componentDidUpdate (Component Did Update): In class components, componentDidUpdate is called whenever the component updates, except for the initial render. You can achieve this behavior in functional components by using useEffect with a dependency array that includes the values you want to watch for changes:

componentWillUnmount (Component Will Unmount): In class components, componentWillUnmount is called before a component is removed from the DOM. You can achieve this behavior in functional components by returning a cleanup function from the useEffect:

Setting up a React App. Rendering Elements

Custom Hook

Custom hooks allow you to **extract component logic into reusable functions**. These functions can contain stateful logic, side effects, or any other features provided by React hooks. Custom hooks help in keeping your components clean and organized by separating concerns and promoting reusability.

Setting up a React App. Rendering Elements

Conditional Rendering

Lists map

Hooks - Intro

useState

useEffect

Custom hook

Custom Hook - Example

```
const { useState } = require("react");

function useLocalStorage(key, initialValue) {
  // Retrieve data from localStorage on component mount
  const storedValue = localStorage.getItem(key);
  const initial = storedValue ? JSON.parse(storedValue) : initialValue;

  // State to hold the current value
  const [value, setValue] = useState(initial);

  // Update localStorage when the value changes
  const setStoredValue = (newValue) => {
    setValue(newValue);
    localStorage.setItem(key, JSON.stringify(newValue));
  };

  return [value, setStoredValue];
}
export default useLocalStorage;
```

In this example, the **useLocalStorage** custom hook allows you to **store and retrieve data from localStorage**. It takes a key (string) and an **initialValue**. The hook initializes the state with the value retrieved from localStorage (if available) or the provided initialValue. The **setStoredValue** function updates both the state and localStorage whenever the value changes.