**1. Introduction to Hooks**

* **What are Hooks?** Hooks are functions that let you use React features (like state and lifecycle) in functional components. Introduced in React 16.8, they allow developers to avoid class components while keeping the same functionality.
* **Why Hooks?**
  + Simplify code by reducing boilerplate.
  + Enhance readability and reusability of logic.
  + Eliminate the need for class components.
  + Improve testability of components.
* **Common Rules of Hooks:**
  + Only call Hooks at the top level (not inside loops, conditions, or nested functions).
  + Only call Hooks from React functional components or custom Hooks.

**2. useState: Managing Local State**

* **Purpose:** To manage local state in a functional component.
* **Syntax:**

jsx

const [state, setState] = useState(initialValue);

* **Example:**

jsx

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

<p>You clicked {count} times</p>

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

</div>

);

}

* **Key Points:**
  + useState returns an array with two elements: the current state and a function to update it.
  + State updates through setState trigger re-renders.

**3. useEffect: Managing Side-Effects**

* **Purpose:** To handle side-effects like data fetching, subscriptions, and DOM manipulations.
* **Syntax:**

jsx

useEffect(() => {

// Effect logic here

return () => {

// Cleanup logic here (optional)

};

}, [dependencies]);

* **Example:**

jsx

Copy code

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

function DataFetcher() {

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

useEffect(() => {

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

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

.then(data => setData(data));

return () => console.log('Cleanup on component unmount');

}, []);

return <ul>{data.map(item => <li key={item.id}>{item.name}</li>)}</ul>;

}

* **Key Points:**
  + The second argument (dependency array) determines when the effect runs.
  + An empty array ([]) makes the effect run only once (on mount).

**4. useRef: Accessing DOM Elements and Component Instances**

* **Purpose:** To access and manipulate DOM nodes or persist a value across renders without triggering re-renders.
* **Syntax:**

jsx

const ref = useRef(initialValue);

* **Example:**

jsx

import React, { useRef } from 'react';

function TextInput() {

const inputRef = useRef();

const focusInput = () => {

inputRef.current.focus();

};

return (

<div>

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

<button onClick={focusInput}>Focus Input</button>

</div>

);

}

* **Key Points:**
  + The ref object persists across renders.
  + useRef does not notify when its value changes.

**5. useReducer: Handling Complex State Logic**

* **Purpose:** To manage complex state logic, often as an alternative to useState.
* **Syntax:**

jsx

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

* **Example:**

jsx

Copy code

import React, { useReducer } from 'react';

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;

}

};

function Counter() {

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

return (

<div>

<p>Count: {state.count}</p>

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

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

</div>

);

}

* **Key Points:**
  + Use for complex state logic or when the next state depends on the previous state.

**6. useContext: Context API for State Sharing**

* **Purpose:** To share state or props globally without prop drilling.
* **Syntax:**

jsx

Copy code

const value = useContext(MyContext);

* **Example:**

jsx

Copy code

import React, { useContext, createContext } from 'react';

const ThemeContext = createContext();

function ThemedButton() {

const theme = useContext(ThemeContext);

return <button style={{ backgroundColor: theme }}>Click me</button>;

}

function App() {

return (

<ThemeContext.Provider value="blue">

<ThemedButton />

</ThemeContext.Provider>

);

}

* **Key Points:**
  + Simplifies consuming context values.
  + Works seamlessly with the Context API.

**7. Custom Hooks**

* **Purpose:** To encapsulate and reuse stateful logic across components.
* **Syntax:**

jsx

function useCustomHook() {

// Logic here

return value;

}

* **Example:**

jsx

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

function useFetch(url) {

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

const [loading, setLoading] = useState(true);

useEffect(() => {

fetch(url)

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

.then(data => {

setData(data);

setLoading(false);

});

}, [url]);

return { data, loading };

}

function DataDisplay() {

const { data, loading } = useFetch('https://api.example.com/data');

if (loading) return <p>Loading...</p>;

return <ul>{data.map(item => <li key={item.id}>{item.name}</li>)}</ul>;

}

* **Key Points:**
  + Start custom Hooks with the prefix use.
  + Combine multiple built-in Hooks to create reusable logic.

In React, hooks are functions that allow developers to "hook into" React features like state, context, and lifecycle methods without needing to write a class-based component. Below are explanations and examples for useRef, useReducer, useContext, and custom hooks.

**1. useRef: Accessing DOM Elements and Component Instances**

The useRef hook is used to persist values between renders. It can be used to directly interact with a DOM element or store a mutable value that doesn’t cause a re-render when updated.

**Example 1: Accessing DOM Elements**

If you need to directly interact with a DOM element (e.g., focus an input field when the component loads), useRef is useful.

jsx

Copy code

import React, { useRef, useEffect } from "react";

const InputFocus = () => {

const inputRef = useRef(null);

useEffect(() => {

// Focus the input element when the component mounts

inputRef.current.focus();

}, []);

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

};

export default InputFocus;

In this example, the useRef hook creates a reference to the input element, and inputRef.current.focus() focuses on the input field when the component is mounted.

**Example 2: Storing a Mutable Value**

You can also store values that need to persist between renders but don't require triggering a re-render, such as a timer ID or previous state values.

jsx

Copy code

import React, { useRef } from "react";

const Timer = () => {

const timerRef = useRef(null);

const startTimer = () => {

timerRef.current = setInterval(() => {

console.log("Timer is running");

}, 1000);

};

const stopTimer = () => {

clearInterval(timerRef.current);

};

return (

<div>

<button onClick={startTimer}>Start Timer</button>

<button onClick={stopTimer}>Stop Timer</button>

</div>

);

};

export default Timer;

Here, timerRef stores the timer ID and allows you to start and stop the timer without causing re-renders.

**2. useReducer: Handling Complex State Logic**

The useReducer hook is an alternative to useState, but it's more suitable for managing complex state logic. It's especially useful when your state is dependent on previous state values or when you have multiple state transitions that need to be handled in a predictable manner (like Redux).

**Example: Using useReducer**

jsx

Copy code

import React, { useReducer } from "react";

// Reducer function: defines how the state changes based on actions

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 Counter = () => {

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>

);

};

export default Counter;

* **useReducer** returns two values:
  + The current state.
  + The dispatch function, which is used to send an action to the reducer function.
* The reducer function is responsible for modifying the state based on the action type.

**3. useContext: Context API for State Sharing**

The useContext hook allows you to access context values (like global state) from a context provider without needing to pass props down the component tree manually.

**Example: Using useContext for Global State**

First, we create the context and a provider to wrap the components that need access to this global state.

jsx

Copy code

import React, { useContext, useState } from "react";

// Create a Context for the theme

const ThemeContext = React.createContext();

const ThemeProvider = ({ children }) => {

const [theme, setTheme] = useState("light");

const toggleTheme = () => {

setTheme(prevTheme => (prevTheme === "light" ? "dark" : "light"));

};

return (

<ThemeContext.Provider value={{ theme, toggleTheme }}>

{children}

</ThemeContext.Provider>

);

};

const ThemeButton = () => {

const { theme, toggleTheme } = useContext(ThemeContext);

return (

<div>

<p>Current theme: {theme}</p>

<button onClick={toggleTheme}>Toggle Theme</button>

</div>

);

};

const App = () => {

return (

<ThemeProvider>

<ThemeButton />

</ThemeProvider>

);

};

export default App;

* ThemeContext.Provider provides the theme state and toggleTheme function to all its child components.
* useContext(ThemeContext) is used in ThemeButton to access the theme and toggleTheme from the context.

This allows for easy state sharing across multiple components without prop drilling.

**4. Custom Hooks: Encapsulating Reusable Logic**

Custom hooks allow you to encapsulate and reuse logic between components. They follow the same rules as built-in hooks, but they let you create your own logic that can be shared across components.

**Example: Custom Hook for Form Handling**

Here's an example of a custom hook for managing form input states:

jsx

Copy code

import React, { useState } from "react";

// Custom hook to manage form inputs

function useForm(initialValues) {

const [values, setValues] = useState(initialValues);

const handleChange = (e) => {

const { name, value } = e.target;

setValues(prevValues => ({

...prevValues,

[name]: value

}));

};

return {

values,

handleChange,

};

}

const Form = () => {

const { values, handleChange } = useForm({ username: "", email: "" });

const handleSubmit = (e) => {

e.preventDefault();

console.log("Form Submitted:", values);

};

return (

<form onSubmit={handleSubmit}>

<div>

<label>Username</label>

<input

type="text"

name="username"

value={values.username}

onChange={handleChange}

/>

</div>

<div>

<label>Email</label>

<input

type="email"

name="email"

value={values.email}

onChange={handleChange}

/>

</div>

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

</form>

);

};

export default Form;

* The useForm hook handles the logic of managing form input state (username and email in this case).
* The hook encapsulates the handleChange function, so any component can reuse the form handling logic.

**Summary:**

* **useRef**: Used to reference DOM elements or store mutable values that don't trigger re-renders.
* **useReducer**: Best for handling complex state logic (e.g., multiple state transitions), similar to Redux but built into React.
* **useContext**: Allows access to global state (or context) without the need for prop drilling.
* **Custom Hooks**: Reusable functions that allow you to extract and share logic across components.

These hooks are powerful tools for managing state, sharing data across components, and creating reusable logic in React.