React Hooks

Introduction

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class component.

First released in October of 2018, the React hook APIs provide an alternative to writing class-based components, and offer an alternative approach to state management and lifecycle methods.

Hooks bring to functional components the things we once were only able to do with classes, like being able to work with React local state, effects and context through **useState, useEffect and useContext.**

Additional Hooks include: **useReducer, useCallback, useMemo, useRef, useImperativeHandle, useLayoutEffect and useDebugValue**.

Introduction

Five Important Rules for Hooks

- Never call Hooks from inside a loop, condition or nested function
- Hooks should sit at the top-level of your component
- Only call Hooks from React functional components
- Never call a Hook from a regular function
- Hooks can call other Hooks

React Hooks

React provides a bunch of standard in-built hooks:

useState: To manage states. Returns a stateful value and an updater function to update it.

useEffect: To manage side-effects like API calls, subscriptions, timers, mutations, and more.

useContext: To return the current value for a context.

useReducer: A useState alternative to help with complex state management.

useCallback: It returns a memorized version of a callback to help a child component not re-render unnecessarily.

useRef: It returns a ref object with a .current property. The ref object is mutable. It is mainly used to access a child component imperatively.

useState hook

```
import  { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useEffect hook

The data fetching, subscriptions, or manually changing the DOM operations from React components before.

We call these operations “side effects” (or “effects” for short) because they can affect other components and can’t be done during rendering.

The Effect Hook, useEffect, adds the ability to perform side effects from a function component.

It serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in React classes, but unified into a single API.

What are side-effects

A React side-effect occurs when we use something that is outside the scope of React.js in our React components e.g. Web APIs like localStorage.

- When we talk about side effects in the context of React.js, we are referring to anything that is outside the scope of React
- So calling any native Web APIs will be considered as a side effect as it’s not within the React universe
- Making a HTTPS request to an external API is another example of a side effect and the list goes on…
- We usually manage React side effects inside the useEffect hook (part of the React Hooks API)

useEffect

What does useEffect do? By using this Hook, you tell React that your component needs to do something after render.
React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates.

```
import { useState, useEffect } from "react";
export default function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  // useEffect(() => {
  //   setTimeout(() => {
  //     setCount((count) => count + 1);
  //   }, 1000);
  // },[]);

  return <h1>I've rendered {count} times!</h1>;
}
```

`useEffect` runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

useEffect

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

There are several ways to control when side effects run.

We should always include the second parameter which accepts an array. We can optionally pass dependencies to useEffect in this array.

1. No dependency passed:

```
useEffect(() => {
  //Runs on every render
});
```

2. An empty array:

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

3. Props or state values:

```
useEffect(() => {
  //Runs on the first render
  //And any time any dependency value
changes
}, [prop, state]);
```

useEffect

useEffect Hook that is dependent on a variable. If the count variable updates, the effect will run again:

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

function Counter() {
  const [count, setCount] = useState(0);
  const [calculation, setCalculation] = useState(0);

  useEffect(() => {
    setCalculation(() => count * 2);
  }, [count]); // add the count variable here

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount((c) => c + 1)}>+</button>
      <p>Calculation: {calculation}</p>
    </>
  );
}
```

Effect Cleanup

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed. We do this by including a return function at the end of the useEffect Hook.

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

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

  useEffect(() => {
    let timer = setTimeout(() => {
    setCount((count) => count + 1);
  }, 1000);

  return () => clearTimeout(timer)
  }, []);

  return <h1>I've rendered {count} times!</h1>;
}
```

Thank You!