Hyperiondev

TASK

React - Hooks

Visit our website

Introduction

WELCOME TO THE REACT - HOOKS TASK!

In earlier versions, React primarily used class components, which were a hassle as you always had to switch between classes, higher-order components, and render props. Now all of this can be done without switching, using function components with React hooks.

WHAT ARE REACT HOOKS?

Hooks are JavaScript functions that manage the state's behaviour and side effects by isolating them from a component. Before React 16.8.0, the most common way of handling lifecycle events required ES6 class-based components. With React Hooks, it is possible to use state and other features in a function component without the need to write a class or define the render method.

There are several types of hooks used in React.

- **State Hooks** - allow a component to "remember" information such as user input.
- **Context Hooks** - allow a component to receive information from distant parents without passing it as props. For example, your app's top-level component can pass the current UI theme to all components below, no matter how deep.
- **Ref Hooks** - allow a component to hold some information that isn't used for rendering, like a Document Object Model (DOM) node or a timeout ID.
- **Effect Hooks** - allow a component to connect to and synchronise with external systems.
- **Performance Hooks** - a common method to optimise re-rendering performance is to skip unnecessary work.
- **Additional Hooks** - some hooks are mostly used by library authors and aren't commonly used in application code.
- **Custom Hooks** - modern React allows you to write custom Hooks for your application's needs, for example, to fetch data, to keep track of whether the user is online.

In this task, we will look at state, effect and ref hooks in more depth, which are some of the most commonly used hooks.

STATE HOOKS

State management is done using the State hook. Below is a quick recap of Components and the State Hook.

Function Components

Functional components are JavaScript functions that use React hooks to create reusable pieces of UI. We call them components because the functions are constructed with single props object arguments and return React elements.

Since functional components are not objects; you must use **React Hooks** to manage state and lifecycle events.

Function components are more concise than the older class components, leading to cleaner, less complex code. They don't include lifecycle methods or inherited members required for code functionality:

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

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

  useEffect(() => {
    console.log("Counter has been initialised or updated.");
  }, [count])

  function handleIncrement() {
    setCount(prevState => prevState + 1);
  }

  return (
    <div>
        <h1>Count: {count}</h1>
        <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}

export default DisplayCount;
```

State hook

The `useState` hook declares a state variable. State variables are preserved between function calls. The hook accepts the initial state of the variable as its argument and returns a pair of values, the current state, and a function that updates it:

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

  return (
    <div>
        <h1>Count: {count}</h1>
    </div>
  );
}
```

In the above example, we declare a state variable called count and set it to zero. It is also possible to declare multiple state variables of different data types:

```
const [clicks, setClick] = useState(0);
const [age, setAge] = useState(27);
const [colour, setColour] = useState('orange');
const [bookList, setBookList] = useState([{ text : 'The Chronicles of
Narnia'}])
```

However, when there are multiple values that you need to keep track of, it is better to declare a single object state as opposed to numerous state variables. A good use case example would be a form with many inputs that are saved or updated through a single API:

```
const [user, setUser] = useState({name: 'Robert', age: 27})
```

Updating state

In a functional component using hooks, we use the update function passed into `useState()`. In the above example, `setUser` is the set function

State variables **should be immutable**. Meaning they cannot have the value changed by using the assignment operator '=', instead, the set function will replace the variable value. If we do not use the state setting function, React has no idea that the object has changed, so a re-render may not be triggered, which would affect the user interface (UI).

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

  return (
    <div>
        <h1>Count: {count}</h1>
        <button onClick={() => setCount(prevState => prevState + 1)}>
          Increment
        </button>
    </div>
  );
}
```

EFFECT HOOK

The **useEffect** hook is like a helper that lets your components do something after they've been rendered on the screen. It's useful for tasks like fetching data from a server, setting up event listeners, or changing something in response to certain conditions:

```
   useEffect(() => {

     //First argument: This block will run as soon as the component is loaded

   },[ /* Dependencies array */ ])
    /* Any variable that is used in this Dependencies Array will be monitored for changes and if any change is detected in this array, the first argument will be executed again.*/
```

The way the useEffect hook triggers re-renders of elements is through its **dependencies array**. The dependencies array is a crucial concept within the useEffect hook, as it determines when the effect should execute or re-execute. This array serves as an optional second argument that you provide to the useEffect function. If this array is left empty, the useEffect will only execute its first argument when the component is loaded.

Imagine that you're building a "To-do List" app. You want to *display the number of tasks you have whenever the list changes*. You can use useEffect to automatically update the count whenever the list of tasks changes!

Let's look at useEffect in code a bit more closely:

```
function FunctionComponent() {

  useEffect(() => {
    console.log("Hello from function component");
  }, []);

  return <h1>Hello from function component</h1>
}
```

In the above example, we have used the `useEffect` hook and an empty dependency array as its second argument meaning that it is only invoked once on mounting.

By default, effects run after every completed render, but you can choose to invoke them only when specific values have changed. For instance, the `useEffect` hook is invoked when the variable "title" is updated in the following example:

```
 const [title, setTitle] = useState('Title');

 useEffect(() => {
     document.title = title;
     console.log("Web page title changed")
 }, [title]);
```

Fetching data from an API

In the code snippet below, we demonstrate how to fetch a to-do from **JSONPlaceholder** using `useEffect` and the JavaScript fetch API to make an asynchronous HTTP request.

The default value of the to-do variable is set to null so that the to-do from JSONPlaceholder is only rendered once the callback function has been executed successfully and the to-do is updated with the URL response:

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

function App() {
  let [todo, setTodo] = useState(null);

  useEffect(() => {
    async function fetchData() {
      let response = await
```

```
fetch("https://jsonplaceholder.typicode.com/todos/1");
      let data = await response.json();
      console.log(data)
      setTodo(data);
    }
    fetchData();
  },[])


  return (
    <h1>{todo.title}</h1>
  )
}
```

API keys

Often when you want to make use of a third-party API, you will be granted an API key. An API key is a string of seemingly jumbled letters and numbers. It's actually a password that gives your app access to the API.

In the lab below, you are going to obtain an API key to use a third-party API. For this task, you can hardcode the key into your program where you make the `fetch()` call.

Here is an example of what the URL for the Open Weather API in a `fetch()` call looks like when hardcoded:

```
`http://api.openweathermap.org/data/2.5/weather?q=${city},${country}&appid
=this_is_an_api_key`
```

Note that this **isn't** secure. If you publish your code on GitHub, anyone will be able to see the key and use the API as if they were you. To help alleviate this, API keys and calls to third-party APIs are usually handled by the back-end of your web app. You will learn to do this soon.

It is good practice if you want to hide the key from your public code, to do the following:

1. Add a file called '.env' in your root folder with key/pairs entries. For instance: `WEATHER_API_KEY=<yourKey>`
2. Now you access the key stored in '.env' from anywhere in your React code by using the process.env variable.

Below is an example of how the same Open Weather API URL mentioned before would look if you use an environment variable instead of hardcoding the API key:

```
`http://api.openweathermap.org/data/2.5/weather?q=${city},${country}&ap
pid=${process.env.WEATHER_API_KEY}`
```

3. If you are using GitHub, add .env to your .gitignore file so that the .env file that stores your API key isn't pushed to GitHub.
4. Backup the .env file somewhere private.

Unsubscribing from listeners

Sometimes the `useEffect` hook uses resources such as a subscription or timer that needs to be terminated once its purpose has been fulfilled. If this isn't handled, the code may attempt to update a state variable that no longer exists, resulting in a memory leak. To avoid this, we implement an effect cleanup function within the `useEffect` hook when the component is unmounted.

Cleanup is only required if we need to terminate a repeated effect when a component unmounts:

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

function App() {
  useEffect(() => {
    const clicked = () => console.log('window clicked')
    window.addEventListener('click', clicked)

    // return a clean-up function
    return () => {
      window.removeEventListener('click', clicked)
    }
  }, [])

  return (
    <div>When you click the window you'll find a
          message logged to the console</div>
  )
}
```

In the snippet above, the clean up function removes the event listener after the user triggers the event. Please note that if you remove the clean-up function, the event gets triggered twice because React would mount, unmount and then mount your component again with the old state.

REF HOOK

The `useRef` hook lets you directly reference the function component's DOM and store mutable values that won't trigger a re-render when updated. We can use the hook to track state changes with the `useEffect` and `useState` hooks.

refs	state
`useRef(initialValue)` returns `{ current: initialValue }` function that accepts props as an argument and returns a React element(JSX).	`useState(initialValue)` returns the current value of a state variable and a state setter function `( [value, setValue])`
Doesn't trigger re-render when you change it.	Triggers re-render when you change it.
Mutable—you can modify and update the current value outside of the rendering process.	"Immutable"—you must use the state setting function to modify state variables to queue a re-render.
You shouldn't read (or write) the current value during rendering.	You can read the state at any time. However, each render has its own snapshot of state which does not change.

In the code snippet below, we use the `useRef` hook to keep track of the application renders:

```
import { useState, useRef, useEffect } from "react";

function CountRender() {
  const [inputValue, setInputValue] = useState("");

  const count = useRef(0);

  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
    <>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />
```

```
      <h1>Render Count: {count.current}</h1>
    </>
  );
}
```

The **useRef()** hook returns a mutable ref object. The **current** property of the ref is initialised to the passed argument in the **useRef(initialValue)** hook.

In the code snippet below, we use the **useRef** hook to auto-focus on an input field.

```
import { useState, useRef, useEffect } from "react";

function AutoFocusInput() {
  const [username, setUsername] = useState("");
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, [])

  return(
    <div>
      <input ref={inputRef} value={username} onChange={(e) =>
setUsername(e.target.value)} />
    </div>
  )
}
```

Compulsory Task 1

- Create a React app that will predict the nationality of a person given their name.
- You will need an auto-focused input field, a button that will trigger a function that will fetch data from the **nationalize.io** API: (https://api.nationalize.io?name=<Enter name here>).

  Here is an example of fetching the results for "Micheal": https://api.nationalize.io?name=michael. After the fetch, please display the details of the first object in the country array.
- Please ensure to **only** use function components and use the `useState`, `useEffect`, and `useRef` hooks.

Compulsory Task 2

- Create a React app that will display the current weather in a particular city.
- Use the Weather API: **https://www.weatherapi.com/**
- Consult the **documentation** of the API in order to get an understanding of how to use the API. You will need to, among other things, obtain an API **key**.
- Allow the user to enter the name of a city in an input field. The App should then retrieve the weather data from the API and display it in a user-friendly manner.
- As an optional challenge, try using a **Geolocation library** to get the user's location and use the data to retrieve the local weather. **This article** gives some pointers to get you started with retrieving a user's location.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.