HyperionDev

React – Hooks

Task

Visit our website

Introduction

Welcome to the React – Hooks task! React used to rely on class components, which could be cumbersome for building complex UIs. They were a hassle because you always had to switch between classes, higher-order components, and render props. Now all of this can be done without switching by using function components with React hooks. This means cleaner, easier-to-read code and a more enjoyable development experience.

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 with the use of hooks.
- **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 or 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

`useState` hook

The `useState` hook in React allows you to add **state** to your functional components. In simple terms, state is a way for your component to remember things between renders. For example, if you want to track a counter or store some user input, you can use `useState` to store that value.

Here's an example of a simple counter using `useState`. This component increases the count by 1 every time you click a button. Create a new file called **Counter.jsx** in the **src** directory and add the following code:

```
// src/Counter.jsx
import { useState } from 'react';

function Counter() {
  // Initialise state with a value of 0
  const [count, setCount] = useState(0);

  // Function to handle button click
  const increment = () => {
    setCount(count + 1); // Update state to count + 1
  };

  return (
    <div>
      <h1>Count: {count}</h1> {/* Display the current count */}
      <button onClick={increment}>Increment</button> {/* Button to increase count
*/}
    </div>
  );
}

export default Counter;
```

Now, you need to use this `Counter` component in your main application file, usually **App.jsx**. Replace the default content with the following:

```
// src/App.jsx

import Counter from "./Counter"; // Import the Counter component

function App() {
  return (
    <div>
      <h1>Welcome to the Counter App!</h1>
      <Counter /> {/* Render the Counter component */}
    </div>
  );
}

export default App;
```

Now that you have everything set up, you can run your Vite application:

```
npm run dev
```

Updating state

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

State variables should be immutable, meaning that you cannot directly change the value of the state variable. If you try to change a state variable directly using the assignment operator (=), React won't be aware of the change. As a result, the component will not re-render, and the UI will not update. React provides the **state setter function** (e.g., `setCount`) to replace the value of the state variable. This setter tells React that the state has changed and triggers a re-render to update the UI.

In modern JavaScript, `const` declares a variable that cannot be reassigned, which helps to enforce immutability. When you declare state variables in React using `const`, you're essentially saying:

> "This variable (`count`) cannot be reassigned using the assignment operator (=)."

Instead, you use the `setState` function (like `setCount`) to update the state, because React internally keeps track of state changes and handles re-renders.

Let's explore how state management works in React functional components using hooks. Below is a simple example demonstrating state updates and UI synchronisation:

```
// src/Counter.jsx

import { useState } from "react";

function Counter() {
  // We use `const` to declare the state variable `count`.
  // This means we cannot directly reassign `count` with `=`.
  const [count, setCount] = useState(0);

  // Function to handle button click and update state using a functional
update
  const increment = () => {
    // We cannot do: count = count + 1; (this would break immutability)
    // Instead, we use the state setter “setCount” to update the state.
    setCount((prevCount) => prevCount + 1); // Functional update to increase
count
  };

  return (
    <div>
      <h1>Count: {count}</h1> {/* Display the current count */}
      <button onClick={increment}>Increment</button>{" "}
      {/* Button to increase count */}
    </div>
  );
}

export default Counter;
```

While `useState` helps manage local component state and triggers UI updates, state changes alone aren't always enough. To handle side effects, such as fetching data or updating the DOM outside of state updates, React provides the `useEffect` hook, which ensures your component stays in sync with external changes.

Effect hooks

`useEffect` hook

The `useEffect` hook in React lets your components **do something** (like fetch data, update the DOM, or set up a timer) **after they've rendered**. You can think of it as a way to handle **side effects** that happen after the UI has been updated. **Side effects** could include: Fetching data from an API, updating the page title, and setting up event listeners.

Below is a basic example:

```
useEffect(() => {
  // This block runs when the component is loaded or when dependencies change
}, [/* Dependencies array */]);

// The variables in the dependencies array are monitored for changes.
// If any variable changes, the effect will run again.
```

The first argument is a function. This is the code that will run after the component renders.

The second argument is the dependencies array. This array determines when the effect runs. If it's empty (`[]`), the effect runs once when the component mounts.

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.

Let's look at `useEffect` in a code example:

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

function FunctionComponent() {
  // useEffect runs after the component renders
  useEffect(() => {
    console.log("Hello from function component");
  }, []); // Empty dependencies array, runs only once when the component
loads

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

export default FunctionComponent;
```

In the above example, we have used the `useEffect` hook and an empty dependency array as its second argument, meaning that it's 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:

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

function TitleChanger() {
  // Create state for the title
  const [title, setTitle] = useState("Title");
  // useEffect to update the document title when “title” state changes
  useEffect(() => {
    // This will change the document title
    document.title = title;
    console.log("Web page title changed to:", title);
  }, [title]); // Dependency array with “title”, effect runs whenever “title”
changes

  return (
    <div>
      <h1>{title}</h1>
      {/* Input to change the title */}
      <input
        type="text"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
        placeholder="Change the title"
      />
    </div>
  );
}

export default TitleChanger;
```

In the **`TitleChanger`** component, the **`useState`** hook is used to create a state variable, **`title`**, which holds the value for the page's title. The **`useEffect`** hook is used to update the document's title whenever the **`title`** state changes. Inside the effect, **`document.title`** is updated, and a message is logged to the console to confirm the change. The dependencies array, **`[title]`**, ensures that the effect only runs when the **`title`** value changes, avoiding unnecessary updates on every render. The component also includes an input field, allowing users to change the title dynamically, with each change reflected immediately in the document's title.

Fetch 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. This is 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. See below:

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

```
// App component to fetch data from an API
function App() {
  // State to store the fetched data
  let [todo, setTodo] = useState(null);

  // Fetching data from an API on component mount
  useEffect(() => {
    // Async function to fetch data
    async function fetchData() {
      // Fetching data from an API
      let response = await fetch(
        "https://jsonplaceholder.typicode.com/todos/1"
      );
      // Parse the JSON data from the response
      let data = await response.json();

      // Log the data to the console
      console.log(data);

      // Update the state with the fetched data
      setTodo(data);
    }

    // Initial call to fetchData function
    fetchData();
  }, []);

  // Conditional rendering based on the state
  return <h1>{todo ? todo.title : "Loading..."}</h1>;
}

export default App;
```

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, but it's actually a password that gives your app access to the API.

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

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

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

Note that this **isn't** secure. If you publish your code on GitHub or deploy it publicly, anyone can inspect the code, see the key, and use the API as if they were you. To mitigate this, API keys and calls to third-party APIs are typically handled by the back-end of your web application. This way, the key is kept secure and not exposed to the user. You will soon learn how to implement this.

If you want to hide the key from your public code, it's good practice 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. A **.env** file is like a secret storage locker for your project's important information that you don't want to share with everyone, such as passwords, API keys, or configuration settings. Instead of putting these details directly in your code (where anyone could see them if they look), you place them in the **.env** file. This way, your project can still access the information it needs to run, but the sensitive data stays hidden and secure. Click **here** to learn more.

3. Now you can access the key stored in **.env** from anywhere in your React code by using the `import.meta.env` variable:

   ```
   const apiKey = import.meta.env.WEATHER_API_KEY;
   ```

   Below is an example of how the same 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}&appid=
   ${import.meta.env.WEATHER_API_KEY}`
   ```

4. 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. Click **here** for more information.

Unsubscribe 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 these resources are not properly handled, the component might attempt to update a state variable that no longer exists, leading to a memory leak. To avoid this, we will implement a cleanup function within the `useEffect` hook, which will run when the component is unmounted.

Cleanup is only required when we need to stop a repeating effect when a component is unmounted. See how this is done below:

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

function App() {
  useEffect(() => {
    // Define a callback function that logs to the console when the window is
clicked
    const clicked = () => console.log('window clicked');

    // Add the event listener for “click” events on the window
    window.addEventListener('click', clicked);

    // Cleanup function: remove the event listener when the component
unmounts
    return () => {
      window.removeEventListener('click', clicked);
    };
  }, []); // Empty dependency array to run effect only once (on mount and
unmount)

  return (
    <div>
      When you click anywhere in the window, you'll see a message logged to
the console.
    </div>
  );
}

export default App;
```

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

Ref hooks

`useRef` hook

The **`useRef`** hook allows you to directly reference a DOM element or store mutable values in a way that won't trigger a re-render when the value is updated. Unlike **`useState`**, which causes the component to re-render when the state changes, **`useRef`** keeps the value without re-rendering. This can be useful when you want to track values or reference DOM elements without affecting the component's lifecycle.

See the comparison between refs and state in the table below:

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 the state that does not change.

One of the most common uses of **`useRef`** is to get direct access to a DOM element (like an input field) in a functional component, as seen below:

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

function FocusInput() {
  // Create a ref object that will store the input DOM element
  const inputRef = useRef(null);

  // Function to focus on the input field
  const focusInput = () => {
    // Access the input field via inputRef.current and call the focus()
method
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Focus me!" />
      {/* Button to trigger the focus function */}
      <button onClick={focusInput}>Focus the Input</button>
    </div>
  );
}

export default FocusInput;
```

`useRef(null)` creates a reference that starts with a value of `null`. We assign this ref to the `input` element using the `ref` attribute `(ref={inputRef})`. When the button is clicked, `inputRef.current` points to the actual DOM element (`<input>`), and we call `focus()` to focus on the input field.

The important thing here is that changing the value of `inputRef.current` does NOT cause a re-render. It's just a way to directly interact with the DOM.

Storing values without causing re-renders

You can also use `useRef` to store values that persist across renders but don't trigger re-renders. This is different from `useState`, which does cause the component to re-render when the value changes. See below:

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

function CountRender() {
  // Initialise state to manage the input value
  const [inputValue, setInputValue] = useState("");

  // Initialise ref to track render count, starts at 0
  const count = useRef(0);

  // Increment the render count on every render
  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
    <>
      {/* Input field with state to handle user input */}
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
      />

      {/* Display the render count stored in the ref */}
      <h1>Render Count: {count.current}</h1>
    </>
  );
}

export default CountRender;
```

From the above, we can see that `CountRender` holds the number of times the component has rendered. This value is updated using `count.current`, but since `useRef` doesn't trigger re-renders, the value persists across renders without causing any new ones.

Each time the component re-renders (because `count` changes), `useEffect` runs and updates `count.current`.

Unlike `useState`, updating the `.current` property of a ref does not trigger a re-render.

Take note

The tasks below are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give the task your best attempt and submit it when you are ready.

After you click "Request review" on your student dashboard, you will receive a 100% pass grade if you've submitted the task.

When you submit the task, you will receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer. Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey for this task, which you can use as a self-assessment tool. Please take a moment to complete the survey.

Once you've done that, feel free to progress to the next task.

Auto-graded task 1

- Create a React app using React and Vite that will predict the nationality of a person given their name.
- You will need an auto-focused input field and 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 **`'Michael'`:**

  `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.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded task 2

- Create a React app using React and Vite that will display the current weather in a particular city.
- Use the **Weather API**.
- Consult the **documentation** of the API in order to get an understanding of how to use it. 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.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Share your thoughts

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

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

Share your thoughts anonymously using this **form**.