Advanced React Hooks: Creating custom reusable Hooks

December 21, 2020 · 4 min read

React Hooks, first introduced in the React 16.8.0 release, are new APIs that allow developers to supercharge functional components. `Hooks` make it possible for us to do with functional components things we could only do with classes.

Consequently, we can use states and other React features without writing classes.

Since their introduction, `Hooks` have had a seismic effect in the React ecosystem. And they have forever changed the way React apps are built.

In this article, we'll look at practical applications of the reusable `Hook` pattern. As we consider these, it's important to mention that `Hooks` are composable, meaning you can call another `Hook` inside your custom `Hook`.

Use cases of reusable `Hook` patterns

Let's consider some reusable `Hook` pattern below:

Using the `useIsMounted` Hook

In React, once a component is `unmounted`, it will never be `mounted` again, which is why we do not set state in an `unmounted` component. This is because it will never be re-rendered.

The image above features a small contrived example app with this issue.

In the app above, the `Dev component` is only rendered when the `showDev state` is true. And clicking the `stop button` toggles the value of the `showDev state`. Consequently, `unmounting` the `Dev component` happens when `showDev` is false.

Below is the implementation of the `Dev component`.

```
function Dev() {
  const [devProfile, setDevProfile] = useState("Fetching Dev...");
  const getDevProfile = () => {
    setTimeout(() => setDevProfile("Lawrence Eagles"), 4000);
  };
  useEffect(() => {
    getDevProfile();
  });
  return (
    <div className="mb-4 text-center">
      <p>{devProfile}</p>
    </div>
  );
}
```

From the code above, we can see that once the component `mounts`, the `getDevProfile` function is called. This takes `4000 milliseconds` to run and thereafter updates the `devProfile state` with `Lawrence Eagles`.

If we `unmount` the `Dev component` (by clicking the `stop button`) before `4000 milliseconds`, React displays the warning error seen in the image above.

Although this error does not break the UI, it is known to cause memory leaks, which hinders performance.

To avoid this issue, some developers do this:

```
if (this.isMounted()) { // This is bad.
  this.setState({...});
}
```

But the React team considers using the `isMounted function` an antipattern, so they recommend you track the `mounted status` yourself.

```
  useEffect(() => {
    let isMounted = true; // sets mounted flag true
    return () => {
      // simulate an api call and update state here
      isMounted = false;
    }; // use effect cleanup to set flag false, if unmounted
  }, []);
  return isMounted;
};
```

Our goal is to abstract the above logic into a custom `Hook` which we can reuse in our code; consequently, we keep our code `DRY`.

To do this, encapsulate all the boilerplate code above into a custom `Hook` (`useIsMounted Hook`), as seen below:

```
import { useEffect, useState } from "react";
const useIsMounted = () => {
  const [isMounted, setIsMouted] = useState(false);
  useEffect(() => {
    setIsMouted(true);
    return () => setIsMouted(false);
  }, []);
  return isMounted;
};
export default useIsMounted;
```

Now we can use it in our app like this:

```
function Dev() {
  const isMounted = useIsMounted();
  const [devProfile, setDevProfile] = useState("Fetching Dev...");
  useEffect(() => {
    function getDevProfile() {
      setTimeout(() => {
        if (isMounted) setDevProfile("Lawrence Eagles");
      }, 4000);
    }
    getDevProfile();
  });
  return (
    <div className="mb-4 text-center">
      <p>{devProfile}</p>
    </div>
  );
}
```

The above code ensures that the state is only updated when the component is still `mounted`.

The `useLoading Hook`

This is a well thought-out reusable `Hook` that can be a time saver in a scenario where you have a number of buttons that link to a resource that is loaded once the component `mounts`.

Typically, there's a `Loading..` spinner of some sort when the `async call` is running to get the resource.

The challenge is that the number of these buttons increases as the resource increases, which can clutter our component with different `loading states`.

Consider this code:

```
import "./styles.css";
import React, { useState, useEffect } from "react";
export default function App() {
  const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
  const [isLoadingDev, setIsLoadingDev] = useState(true);
  const [isLoadingStack, setIsLoadingStack] = useState(true);
  const fetchDevs = async () => {
    console.log("this might take some time....");
    await delay(4000);
    setIsLoadingDev(false);
    console.log("Done!");
  };
  const fetchStacks = async () => {
    console.log("this might take some time....");
    await delay(5000);
    setIsLoadingStack(false);
    console.log("Done!");
  };
  useEffect(() => {
    fetchDevs();
```

In the code above, the `fetchDev` and `fetchStacks` functions are contrived to simulate an `async request`. Once the component `mounts`, they are called and the message in the buttons changes when these function finishes.

The `loading status` of each button is handled by a `useState` initialization and would increase in number as we load more resources.

This code is not `DRY` and the repetition is a recipe for bugs.

We can refactor the code above by creating a reusable `useLoading Hook`, as shown here:

```
import { useState } from "react";
const useLoading = (action) => {
  const [loading, setLoading] = useState(false);
  const doAction = (...args) => {
    setLoading(true);
    return action(...args).finally(() => setLoading(false));
  };
  return [doAction, loading];
};
export default useLoading;
```

This hook takes an `async function` and returns an array containing that function and the `loading status`.

We have also been able to abstract our `useState` logic to this component and we only need one initialization.

We can use it in our code, like this:

```
import "./styles.css";
import React, { useEffect } from "react";
import useLoading from "./useLoading";
export default function App() {
  const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
  const fetchDevs = async () => {
    console.log("this might take some time....");
    await delay(4000);
    console.log("Done!");
  };
  const fetchStacks = async () => {
    console.log("this might take some time....");
    await delay(5000);
    console.log("Done!");
  };
  const [getDev, isLoadingDev] = useLoading(fetchDevs);
  const [getStacks, isLoadingStack] = useLoading(fetchStacks);
  useEffect(() => {
    getDev();
    getStacks();
```

Here, we used `array destructuring` to get the `async action function` and the `loading status`.

```
  const [getDev, isLoadingDev] = useLoading(fetchDevs);
  const [getStacks, isLoadingStack] = useLoading(fetchStacks);
```

These are then used in the `useEffect Hook` and in the view. The result is a clean code that is `DRY` and easier to maintain.

Also, after all `loading status` is completed, we can now easily do things like render a component, update state, etc.

```
if(isLoadingDev && isLoadingStack) {
  // do somthing
}

return {
  // normal component view
}
```

Conclusion

I hope after this discourse you appreciate the need to keep your codes `DRY` and are ready to start writing custom reusable `Hooks`.

These are just two examples of advanced patterns of creating reusable custom `Hooks`, now hopefully you can create your own advanced pattern.

You can read more on building your own `Hooks` here.

Full visibility into production React apps

Debugging React applications can be difficult, especially when users experience issues that are hard to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.

LogRocket is like a DVR for web apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

Lawrence Eagles Follow

Senior full-stack developer, writer, and instructor.

#react

3 Replies to "Advanced React Hooks: Creating custom reusable Hooks"

Koire Says:
December 22, 2020 at 6:16 am

Reply

I noticed in your isMounted hook you had 2 returns, I'm pretty sure the second one is unreachable.

Toby Says:
December 24, 2020 at 11:33 am

Reply

You are mistaken.. The first return statement is in the effect (function passed to useEffect) & not a directly under the main function like the second.

Richard Antao Says:
April 1, 2021 at 12:51 pm

Reply

Your useIsMounted is better off using useRef than useState, because useState causes your component to rerender

Leave a Reply

Enter your comment here...