Building a Custom Hook

In this lesson, you'll learn how to create a custom hook!

WE'LL COVER THE FOLLOWING

- Introduction
- useExpanded Custom Hook
- useEffectAfterMount Custom Hook
- Using Custom Hooks
- Quick Quiz!

Introduction

Before we go into the other advanced patterns, it's important to understand the default way to share functionality with hooks — building a custom hook.

Only by building on this foundation can we take advantage of the other advanced patterns we will discuss.

So far, we've built a compound component that works great! Let's say you were the author of some open-source library, and you wanted to expose the "expand" functionality via a custom hook, how would you do this?

First, let's agree on the name of your open-source (OS) library. We'll leave it as Expandable — same as before.

Now, instead of having the logic for managing the expanded state in an Expandable component that returns a Provider, we can just export 2 custom hooks.



useExpanded Custom Hook

The useExpanded custom hook will now handle the logic for the expanded state variable, and useEffectAfterMount will handle the logic for invoking a callback only after mount.

So, shall we write these custom hooks?

Note that these custom hooks will pretty much use the same logic as the **Expandable** compound component we had written earlier. The difference here will be wrapping these in a custom hook.

Here's the useExpanded custom hook:

```
.Expandable-panel {
   margin: 0;
   padding: 1em 1.5em;
   border: 1px solid hsl(216, 94%, 94%);;
   min-height: 150px;
}
```

If you would like a refresher on the internal logic of this custom hook, feel free to look at the section on compound components.

What's important is that we've wrapped this functionality in a useExpanded function (aka custom hook) which returns the value we previously had in a context Provider.

useEffectAfterMount Custom Hook

We'll do something similar with the useEffectAfterMount custom hook as shown below:

```
.Expandable-panel {
   margin: 0;
   padding: 1em 1.5em;
   border: 1px solid hsl(216, 94%, 94%);;
   min-height: 150px;
}
```

The only difference here is that useEffectAfterMount doesn't return any value.

Rather, it have the userffect mook. To make this as generic as possible, the

custom hook takes in two arguments, the callback to be invoked after mount, and the array dependencies on which the useEffect function relies.

Also, note that **line 8** reads, **return cb()**. This is to handle unsubscriptions by returning whatever is returned by the callback.

Great!

Using Custom Hooks

Now that you've built these custom hooks, how would a typical consumer of your OS library use these hooks?

Here's one simple example. Have a look at App.js.

```
.Expandable-panel {
    margin: 0;
    padding: 1em 1.5em;
    border: 1px solid hsl(216, 94%, 94%);;
    min-height: 150px;
}
```

The consumer imports your useExpanded custom hook and invokes the function to retrieve expanded and toggle.

With these values, they can render whatever they want. We provide the logic and let the user render whatever UI they deem fit. By default, they render a button.

This button invokes the exposed toggle function from the custom hook. Based on the toggled expanded state, they render smileys.

Well, this is the user's version of an expandable container.

Do you realize what we've done here?

Like the render props API, we've given control over to the user to render whatever UI they want while we handle whatever logic is required for them to do so.

To use the useEffectAfterMount hook, the user needs to do something like the

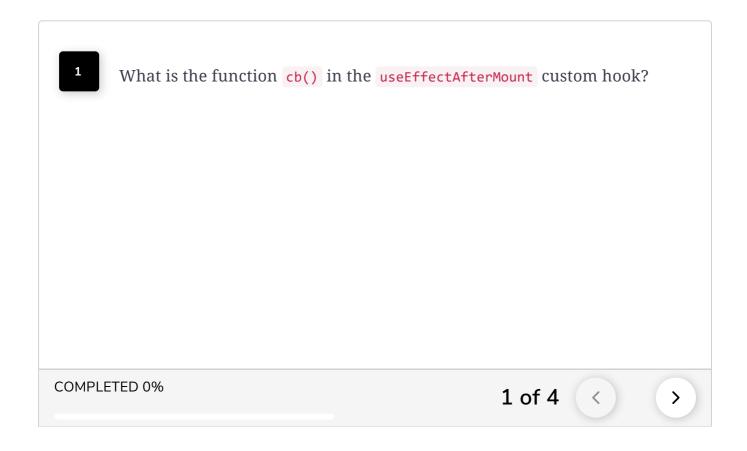
following; have a look at App. js.

```
.Expandable-panel {
   margin: 0;
   padding: 1em 1.5em;
   border: 1px solid hsl(216, 94%, 94%);;
   min-height: 150px;
}
```

Now, whenever the button is clicked, they'll get the logs in the console!

Quick Quiz!

Let's take a quiz. You know the drill.



Now that we know how to build and use custom hooks, let's make the user's life even easier and provide them with some default UI elements.