Resetting the State

WE'LL COVER THE FOLLOWING

While we've made it easier for the user to dictate the initial state within the custom hook, they should also be able to reset the state to the initial state at any point in time, i.e., a `reset` callback they can invoke to reset the state to the initial default state they provided.

This is useful in many different use cases.

Why Reset the State?

Let's consider a really trivial example.

Assume the terms and conditions content in the user app was so long that they changed the default expanded state to `false`, i.e., the expandable content isn't open by default.

Since the content was long, they decided to provide a button towards the end of the write-up. A reader could click to revert the expandable content to the initial closed state for which they may want to close the expandable content and perform some cleanups.

We could provide the user a `reset` callback for this, right?

Even though this particular example isn't the most realistic for the `reset` functionality, in larger applications you can solve the problem using the same method discussed here.

method discussed here.

How to Reset State?

So, here comes the solution.

All the `reset` function should do is set the “expanded” state back to the default provided, i.e., `initialExpanded`

Here’s a simple implementation:

```
export default function useExpanded (initialExpanded = false) {
  const [expanded, setExpanded] = useState(initialExpanded)
  const reset = useCallback(
    () => {
      // look here 👇
      setExpanded(initialExpanded)
    },
    [initialExpanded]
  )
  ...
}
```

The code’s simple enough. All the `reset` function does is call `setExpanded` with the `initialExpanded` value to reset the state back to the initial state supplied by the user.

Easy enough. However, there’s still something we need to do.

Remember, this consumer of our custom hook wants to close the terms and conditions body and also perform some cleanup/side effect.

How will this user perform the cleanup after a reset? We need to cater to this use case as well.

Let’s get some ideas from the implemented solution for the user to run custom code after every change to the internal `expanded` state.

```
// we made this possible 👇
useEffectAfterMount(
  () => {
    // user can perform any side effect here 👇
    console.log('Yay! button was clicked!!')
  },
  [expanded]
)
```

Now we need to make the same possible after a reset is made. Before I explain

the solution to that, have a look at the usage of `useEffectAfterMount` in the code block above.

`useEffectAfterMount` accepts a callback to be invoked and an array dependency that determines when the effect function is called except when the component just mounts.

Now, for the regular state update, the user just had to pass in the array dependency `[expanded]` to get the effect function to run after every expanded state update.

For a reset what do we do?

Ultimately, here's what we want the user to do.

```
// user's app
const { resetDep } = useExpanded(false)
useEffectAfterMount(
  () => {
    console.log('reset cleanup in progress!!!!')
  },
  [resetDep]
 )
...
```

We need to provide a reset dependency the user can pass into the `useEffectAfterMount` array dependency.

That's the end goal.

Can you think of a solution to this? What do we expose as a reset dependency?

Well, the first thing that comes to mind is a state value to be set whenever the user invokes the `reset` callback. The state value will keep track of how many times a reset has been made.

If we increment the counter variable every time a reset is made, we can expose this as a reset dependency as it only changes when an actual reset is carried out.

Here's the implementation of that:

```
// useExpanded.js
...
const [resetDep, setResetDep] = useState(0)
const reset = useCallback(
```

```
  () => {
    // perform actual reset
    setExpanded(initialExpanded)
    // increase reset count - call this resetDep
    setResetDep(resetDep => resetDep + 1)
   },
   [initialExpanded]
 )
...
```

We can then expose `resetDep` alongside other values.

```
// useExpanded.js
...
const value = useMemo(
    () => ({
      expanded,
      toggle,
      getTogglerProps,
      reset,
      resetDep
    }),
    [expanded, toggle, getTogglerProps, reset, resetDep]
 )
 return value
...
```

The Output

This works just as expected!

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

This is a decent solution. It works fine and is easy to reason about.

There's arguably one problem with the solution. Should we really be saving the reset count as a new state variable?

The `useExpanded` custom hook is mostly responsible for managing the `expanded` state. Introducing a new state variable feels like some inner conflict/pollution.

Quick Quiz!

Time for a quiz.

Q

Why is a `reset` callback important?

COMPLETED 0%

1 of 1

We'll have a look at this in the next lesson, however, note that this is just another case of personal preference. There's nothing technically wrong with the solution above.