Handling State Change Callbacks

In this lesson, we'll learn how to use React hooks to handle state change callbacks.

WE'LL COVER THE FOLLOWING

Triggering Callbacks Upon State Change

Let's borrow a concept from your experience with React's class components. If you remember, it's possible to do this with class components:

```
this.setState({
  name: "value"
}, () => {
  this.props.onStateChange(this.state.name)
})
```

If you don't have experience with class components, this is how you trigger a **callback after a state change** in class components.

Usually, the callback, e.g., `this.props.onStateChange` on **line 4**, is always invoked with the current value of the updated state as shown below:

```
this.props.onStateChange(this.state.name)
```

Why is this important? This is good practice for creating reusable components because this way, the consumer of your component can attach any custom logic to be run after a state update.

For example:

For example:

```
const doSomethingPersonal = ({expanded}) => {
  // do something really important after being expanded
}
<Expandable onExpanded={doSomethingPersonal}>
 ...
</Expandable>
```

In this example, we assume that after the `Expandable` component's `expanded` state property is toggled, the `onExpanded` prop will be invoked — hence calling the user's callback, `doSomethingPersonal`.

We will add this functionality to the `Expanded` component.

With class components, this is pretty straightforward. With functional components, we need to do a little more work — not too much though :)

Using `useEffect`

In most cases, when you want to perform a side effect within a functional component, you should reach for `useEffect`.

The most natural solution might look like this:

```
useEffect(() => {
  props.onExpanded(expanded)
}, [expanded])
```

The problem with this, however, is that the `useEffect` function is called at least once — when the component is initially mounted.

So, even though there's a dependency array, `[expanded]`, the callback will also be invoked when the component mounts!

```
useEffect(() => {
  // this function will always be invoked on mount
})
```

Preventing Invocation Upon Mount

The functionality we seek requires that the callback passed by the user isn't invoked on mount.

How can we enforce this?

First, consider the naive solution below:

```
//faulty solution
...
let componentJustMounted = true
useEffect(
    () => {
      if(!componentJustMounted) {
        props.onExpand(expanded)
        componentJustMounted = false
      }
    },
    [expanded]
  )
...
```

What's wrong with the code above?

Loosely speaking, the thinking behind the code is correct. You keep track of a certain variable `componentJustMounted`, set it to `true`, and only call the user callback `onExpand` when `componentJustMounted` is false.

Finally, the `componentJustMounted` value is only set to `false` after the user callback has been invoked at least once.

Looks good.

However, the problem with this is that whenever the function component re-renders owing to a state or prop change, the `componentJustMounted` value will always be reset to `true`. Thus, the user callback `onExpand` will never be invoked as it is only invoked when `componentJustMounted` is false.

```
...
if (!componentJustMounted) {
        onExpand(expanded)
}
...
```

Quick Quiz!

Quiz yourself on what we've learned so far.

1

How do you update a component after a state change in traditional React class components?

COMPLETED 0%

1 of 2

We have to somehow keep the value of `componentJustMounted` constant.

The solution to this problem lies in the next lesson. Catch you there!