

How Control Props Work

In this lesson, we'll make the Expandable component work so that the elements expand one at a time

WE'LL COVER THE FOLLOWING ^

- Controlled vs. Uncontrolled Elements
- Implementing in **Expandable**
 - How it Works
 - Actual Implementation
- The Final Result
- Quick Quiz!

Controlled vs. Uncontrolled Elements

Consider a simple input field.

```
<input />
```



In React, a controlled input element will be defined like this:

```
<input value={someValue} onChange={someHandler} />
```



Hmm, that's different.

A controlled input component has its **value** and **onChange** handler managed **externally**.

This is exactly what we aim for when implementing the control props pattern. We want the internally-managed state to be manageable from the outside via props!

Implementing in **Expandable**

How it Works

The implementation is quite simple, but we need to understand how it works before we delve into writing some code.

The default usage of the `Expandable` component is as shown below:

```
<Expandable>
  <Expandable.Header>
    {someHeader}
  </Expandable.Header>
  <Expandable.Icon />
  <Expandable.Body>{someContent}</Expandable.Body>
</Expandable>
```

Within the `Expandable` component, the `expanded` value is internally managed and communicated to the children.

If we want to make `Expandable` a controlled component, then we need to accept a prop which defines the `expanded` value externally.

For example:

```
<Expandable shouldExpand={someUserDefinedBooleanValue}>
  <Expandable.Header>
    {someHeader}
  </Expandable.Header>
  <Expandable.Icon />
  <Expandable.Body>{someContent}</Expandable.Body>
</Expandable>
```

Internally, if the `shouldExpand` prop is passed, we will give up control of the state update to the user, i.e., let the `expand` value return whatever the user defines in the `shouldExpand` prop.


Actual Implementation

Now, let's move on to the actual implementation.

First, we will check if the control prop `shouldExpand` exists. If it does, then the component is a controlled component, i.e., `expanded` managed externally.

```
// Expandable.js
const Expandable = ({
  shouldExpand, // take in the control prop
```

```

...otherProps
})) => {
  // see check 

  const isExpandControlled = shouldExpand !== undefined
  ...
}

```

Not everyone would pass in the control prop. It's possible that a lot of users just go for the default use case of letting us handle the state internally. This is why we check to see if the control prop is defined.

Now, the `isExpandControlled` variable is going to define how we handle a couple of other things internally.

It all begins with the value that the `Expandable` component communicates to the child elements.

```

// before
const value = useMemo(() => ({ expanded, toggle}), [
  expanded, toggle
])

```



Before, we passed along the `expanded` and `toggle` values from the internals of the `Expandable` component.

But we can't do this now.

Remember, if the component is controlled, we want `expanded` to return the `shouldExpand` prop passed in by the user.

We can extract this behavior into a `getState` variable defined below;

```

// Expanded.js
...
const getState = isExpandControlled ? shouldExpand : expanded
...

```



where `shouldExpand` is the user's control prop and `expanded` is our internal state.

Be sure to remember that a controlled input element is passed the value prop and **onChange** prop. A controlled component receives both the state value and the function handler as well.

Before now, we had an internal `toggle` function.

```
// before. See toggle below
const toggle = useCallback(
  () => setExpanded(prevExpanded => !prevExpanded),
  []
)
```

Now, we can introduce a separate variable `getToggle` defined below:

```
// Expandable.js
...
const getToggle = isExpandControlled ? onExpand : toggle
...
```

If the component is controlled, we return the `onExpand` prop — a user defined callback. If not, we return our internal implementation, `toggle`.

With these done, here's the value communicated to the child elements:

```
...
const value = useMemo(() => ({ expanded: getState, toggle: getToggle }), [
  getState,
  getToggle
])
```

Note how `getState` and `getToggle` are both used to get the desired `expanded` state and `toggle` function depending on whether the component is controlled or not.

We're pretty much done implementing the control prop pattern here.

There's one more cleanup to be performed.

When the component wasn't controlled, the `onExpand` prop was a user-defined function invoked after every state change. That's not going to be relevant if the component is controlled. Why? The user handles the toggle function themselves when the component is controlled. They can choose to do anything in the callback with the already controlled state value too.

Based on the explanation above, we need to make sure the `useEffect` call doesn't run when the component is controlled.

```
// Expandable.js
...
useEffect(
  () => {
    // run only when component is controlled.
    // see !isExpandControlled
    if (!componentJustMounted && !isExpandControlled) {
      onExpand(expanded)
      componentJustMounted.current = false
    }
  },
  [expanded, onExpand, isExpandControlled]
)
...
```

And that is it!

We've implemented the control props pattern.

The Final Result

Here's the final result.

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

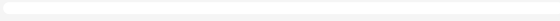
I have yet to show you how it solves the user's problem and how to use it in the user's app. We'll learn that next! But before that, let's take a quick quiz.

Quick Quiz!

1

What makes a component a *controlled component*?

COMPLETED 0%



1 of 3



Hope you got those right! Catch you in the next lesson.