Example: Building an Expandable Component

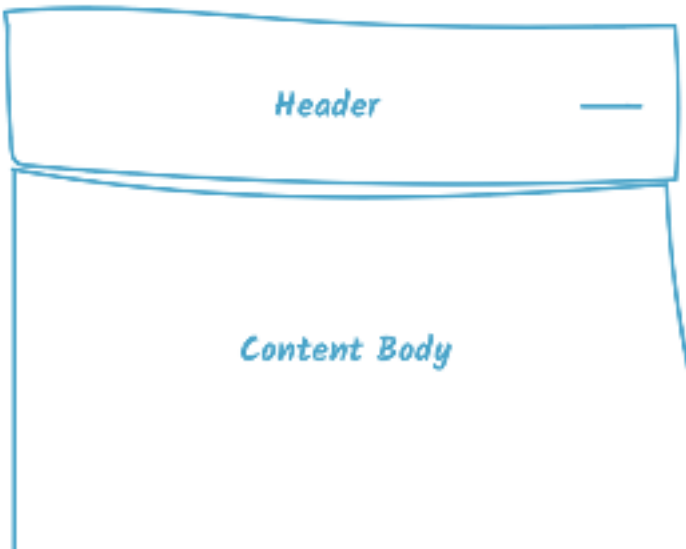
In this lesson, we will set up an expandable component that is based on the compound component pattern

WE'LL COVER THE FOLLOWING

What is an Expandable Component?

We'll be building an `Expandable` component. It will have a clickable header that toggles the display of an associated body of content.

In the unexpanded state, the component would look like this



And this, when expanded

Designing the API

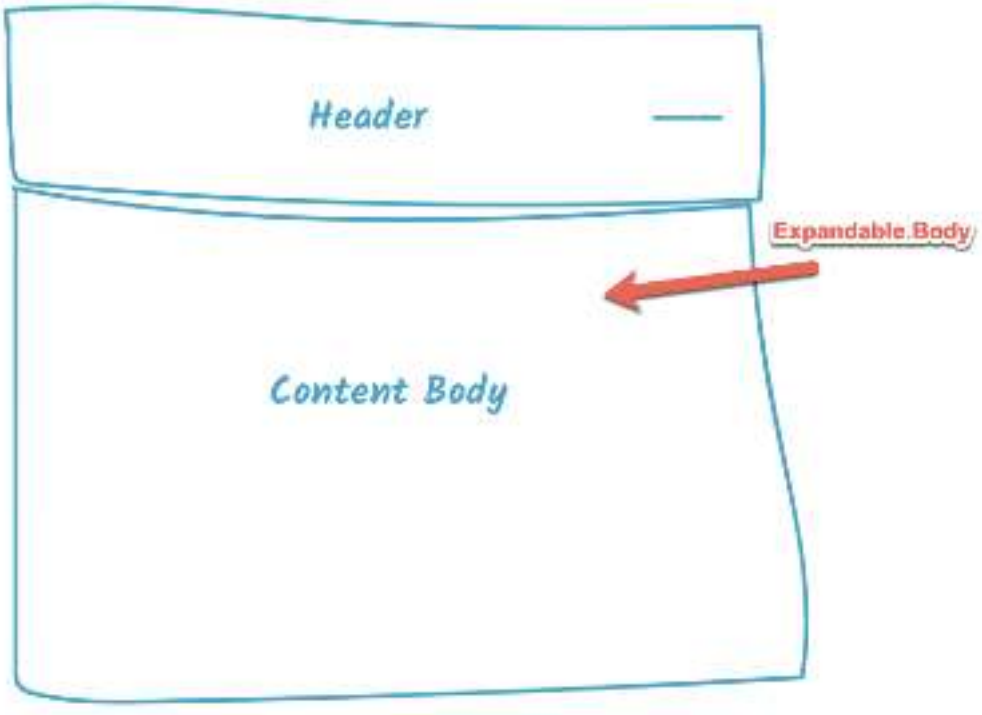
It's usually a good idea to write out what the exposed API of your component would look like before building it out.

In this case, here's what we're going for:

```
<Expandable>
        <Expandable.Header> Header </Expandable.Header>
        <Expandable.Icon/>
    <Expandable.Body> This is the content </Expandable.Body>
</Expandable>
```

exposed API of expandable component

Unexpanded



Expanded

In the code block above, you'll notice I have used expressions like this: `Expandable.Header`

You can do this as well:

```
<Expandable>
        <Header> Header </Header>
        <Icon/>
    <Body> This is the content </Body>
</Expandable>
```

It doesn't matter which way as either way works. I have chosen `Expandable.Header` over `Header` as a matter of personal preference. I find that it communicates dependency on the parent component well, but that's just my preference. A lot of people don't share the same preference and that's perfectly fine. Feel free to use whichever component looks good to you!

It's your component, use whatever API looks good to you ☺

Building the Expandable Component

The `Expandable` component as the parent component will keep track of the state, and it will do this via a boolean variable called `expanded`.

```
// state
{
  expanded: true || false
}
```

The `Expandable` component needs to communicate the state to every child component regardless of their position in the nested component tree.

Remember that the children are dependent on the parent compound component for the state.

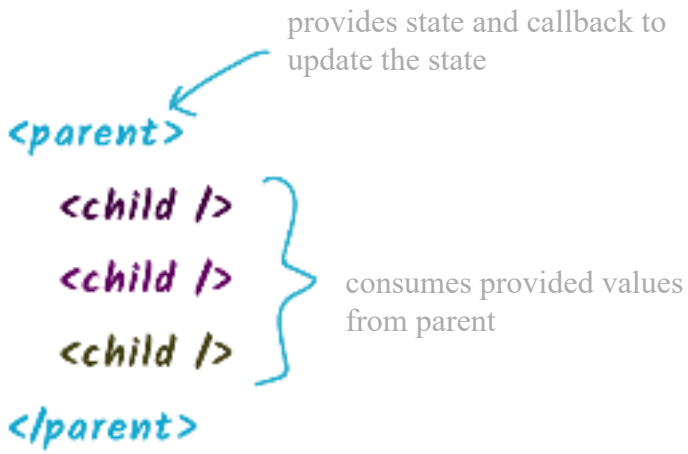
What is the best way to go about this?

Show Hint

Communicating the State to Child Components

We need to create a `context` object to hold the component state, and expose

the `expanded` property via the `Provider` component. Alongside the `expanded` property, we will also expose a function callback to toggle the value of this `expanded` state property.



The state relationship of the expandable component

If that makes sense to you, here's the starting point for the `Expandable` component. We'll be creating this in a file called `Expandable.js`

```
import React, { createContext } from 'react'

const ExpandableContext = createContext()
const { Provider } = ExpandableContext

const Expandable = ({children}) => {
  return <Provider>{children}</Provider>
}

export default Expandable
```

There's nothing spectacular going on in the code block above. There is no output yet, so we haven't put the code in one of our runnable widgets! Don't worry though, we're getting to that part soon.

A context object is created, and the `Provider` component is deconstructed. Then, we go on to create the `Expandable` component which renders the `Provider` and any `children`.

Got that?

With the basic setup out of the way, let's do a little more.

The context object was created with no initial value. However, we need the `Provider` to expose the state value `expanded` and a toggle function to update the state.

Let's create the `expanded` state value using `useState`.

```
import React, { createContext, useState } from 'react'

const ExpandableContext = createContext()
const { Provider } = ExpandableContext

const Expandable = ({children}) => {
  // look here 👇
  const [expanded, setExpanded] = useState(false)
  return <Provider>{children}</Provider>
}

export default Expandable
```

With the `expanded` state variable created, let's create the `toggle` updater function to toggle the value of `expanded` — whether `true` or `false`.

```
import React, { createContext, useState } from 'react'

const ExpandableContext = createContext()
const { Provider } = ExpandableContext

const Expandable = ({children}) => {
  const [expanded, setExpanded] = useState(false)
  // look here 👇
  const toggle = setExpanded(prevExpanded => !prevExpanded)
  return <Provider>{children}</Provider>
}

export default Expandable
```

The `toggle` function invokes `setExpanded`, the actual updater function returned from the `useState` call. Every updater function from the `useState` call can receive a function argument. This is similar to how you pass a function to `setState` e.g. `setState(prevState => !prevState.value)`.

This is the same as what I've done above. The function passed to `setExpanded` receives the previous value of `expanded`, i.e., `prevExpanded` and returns the

opposite of that, `!prevExpanded`

Quick Quiz!

Before we move on, let's take a quick quiz on what we've covered so far!

1

The first snippet is inherently worse than the second.

```
// First
<Expandable>
    <Header> Header </Header>
    <Icon/>
    <Body> This is the content </Body>
</Expandable>
```

```
// Second
<Expandable>
    <Expandable.Header> Header </Expandable.Header>
    <Expandable.Icon/>
    <Expandable.Body> This is the content </Expandable.Body>
</Expandable>
```

COMPLETED 0%

1 of 2

Let's make some optimizations now. Catch you in the next lesson!