The Props Collection Pattern in Practice

Let's apply the props collection pattern to our expandable component!

WE'LL COVER THE FOLLOWING

The Props Collection

Take a look at the props collection for the `useExpanded` hook:

```
export default function useExpanded () {
  const [expanded, setExpanded] = useState(false)
  const toggle = useCallback(
    () => setExpanded(prevExpanded => !prevExpanded),
    []
  )
  // look here 👇
  const togglerProps = useMemo(
    () => ({
      onClick: toggle,
      'aria-expanded': expanded
    }),
    [toggle, expanded]
  )

  ...
  return value
}
```

Within `useExpanded` we've created a new memoized object, `togglerProps`, which includes the `onClick` and `aria-expanded` properties.

Note that we've called this props collection `togglerProps` because it is a prop collection for the toggler, i.e., the toggle element, regardless of which UI element it is.

We'll then expose the `togglerProps` from the custom hook via the returned

`value` variable.

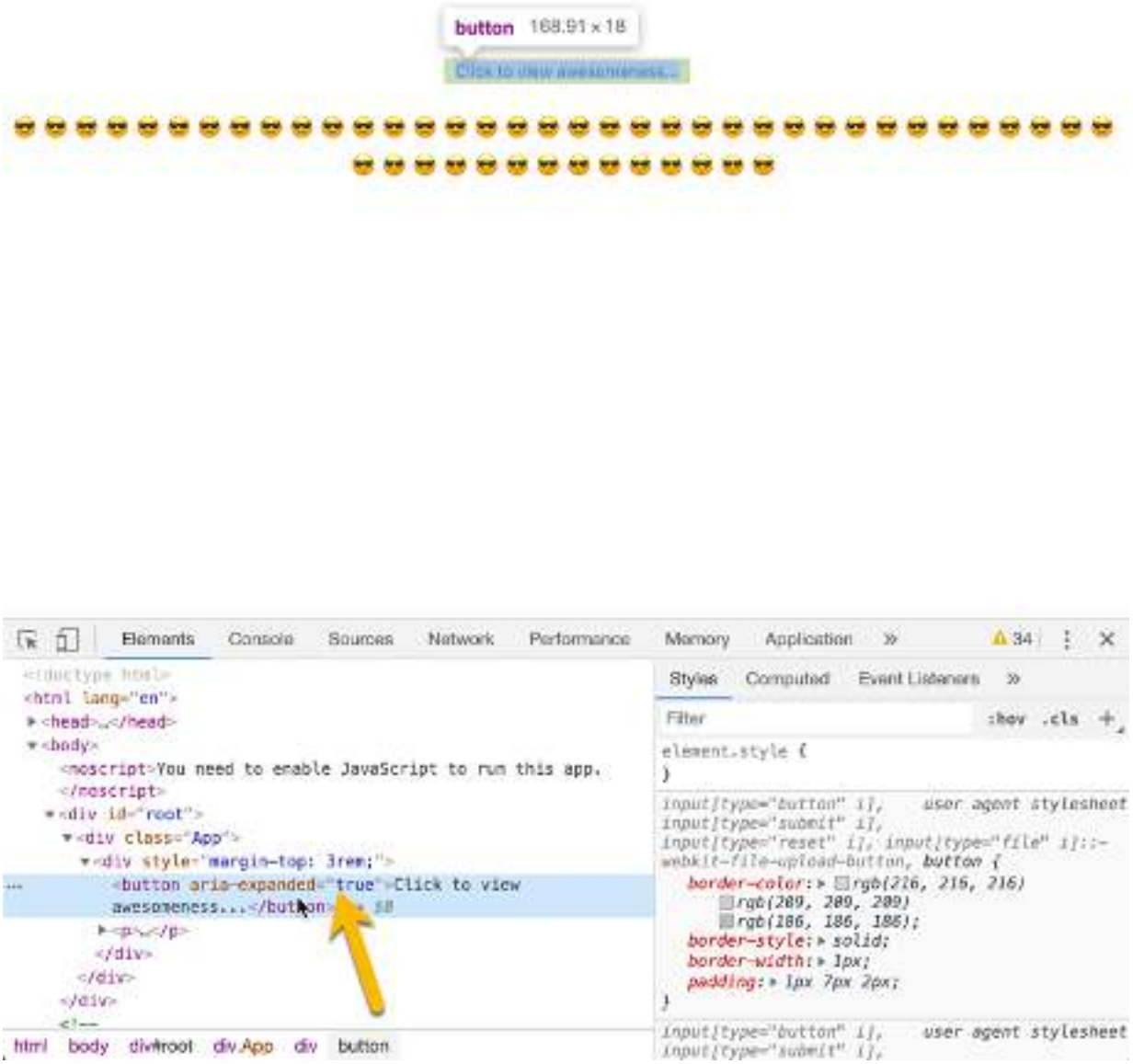
```
export default function useExpanded () {
  ...
  // look here - togglerProps has been included 👇
  const value = useMemo(() => ({ expanded, toggle, togglerProps }), [
    expanded,
    toggle,
    togglerProps
  ])

  return value
}
```

Final Output

With `togglerProps` now exposed, it can be consumed by any user as shown below. Have a look at `useExpanded.js`.

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

This works just like before, except that the user didn't have to manually write the "common props" for the toggle element.



Here is the expected output of the app. aria attribute also added to the DOM element.

In the next chapter, we will consider an alternative (perhaps more powerful) solution to the same problem the props collection pattern aims to solve.