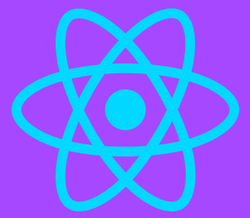
COMP6080

How React Works

Presented by
Nic Barker

What is  React ?

History

React is a Javascript library for building user interfaces.

Released in its recognisable form in 2013.

Main ideas are declarative rendering and components.

What is declarative rendering?

Contrasts with **imperative** rendering.

Imperative VS Declarative

Imperative

Specify the process, not the outcome

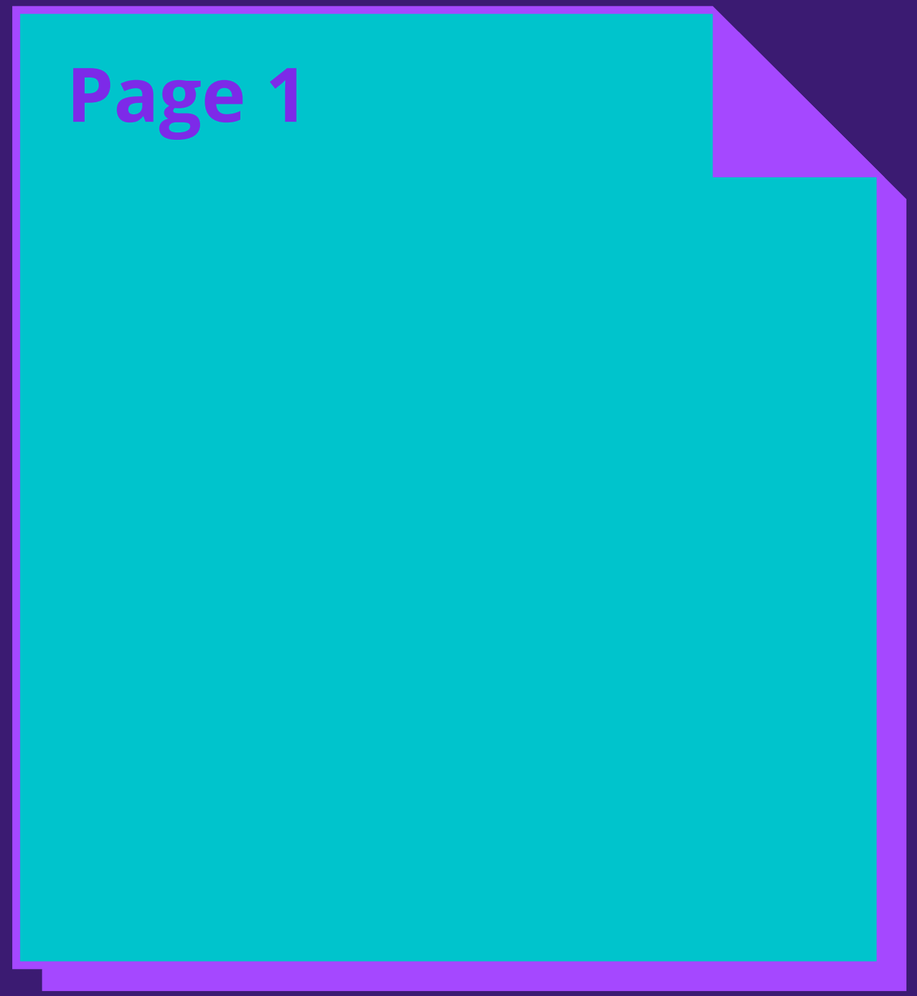
Declarative

Specify the outcome, not the process

Scenario

We have a simple web page that is blank with a green background, we are calling "Page 1"

Imperative VS Declarative



Imperative VS Declarative

How do we create this page using Javascript?

The Imperative Way

Execute the exact steps required to make the changes

```
document.body.style.backgroundColor = "green";
```

Imperative VS Declarative

Imperative:
Specify the process, not the outcome

The Declarative Way

Declare the expected UI and let react figure out how to make the changes

```
function App() {
  return (
    <body style={{ backgroundColor: "green" }} />
  );
};
```

Imperative VS Declarative

Declarative:
Specify the outcome, not the process

Imperative VS Declarative

Seems like a lot more code for the same outcome, why bother?

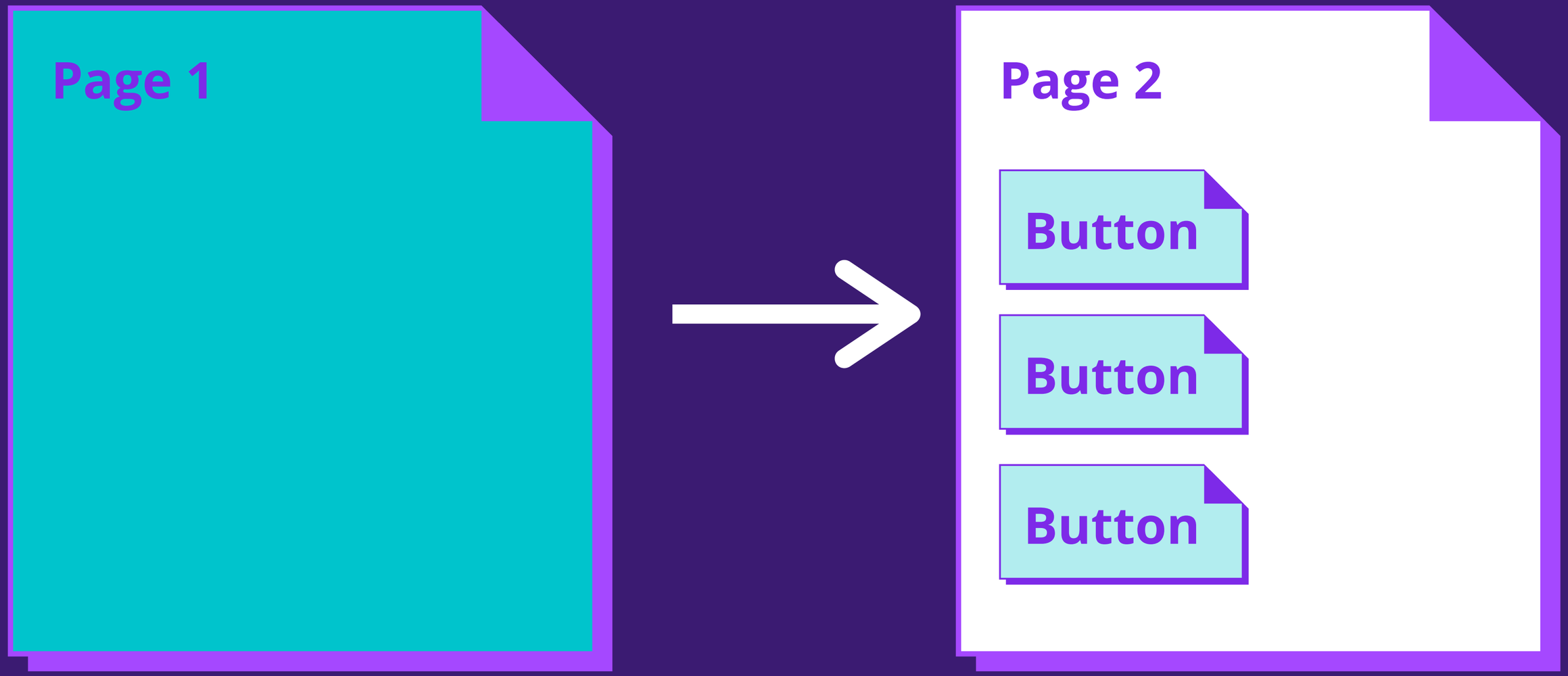
- `react` + `react-dom` is 109 kb (34.8 kb gzipped),

Scenario 2

We now have a second page - "Page 2" - which has a white background and three buttons.

We want to transition from Page 1 to Page 2.

Imperative VS Declarative



Imperative VS Declarative

- The imperative way - execute the exact steps for making the changes

```
document.body.style.backgroundColor = "white"
const buttonOne = document.createElement("button");
const buttonTwo = document.createElement("button");
const buttonThree = document.createElement("button");

document.body.appendChild(buttonOne);
document.body.appendChild(buttonTwo);
document.body.appendChild(buttonThree);
```

Imperative VS Declarative

The Declarative Way

Declare the expected states and let react figure out how to make the changes

```
function App(props) {
  if (props.pageType === 'one') {
    return (
      <body style={{ backgroundColor: 'green' }} />
    );
  } else {
    return (
      <body style={{ backgroundColor: 'white' }}>
        <button />
        <button />
        <button />
      </body>
    );
  };
};
```

Imperative VS Declarative

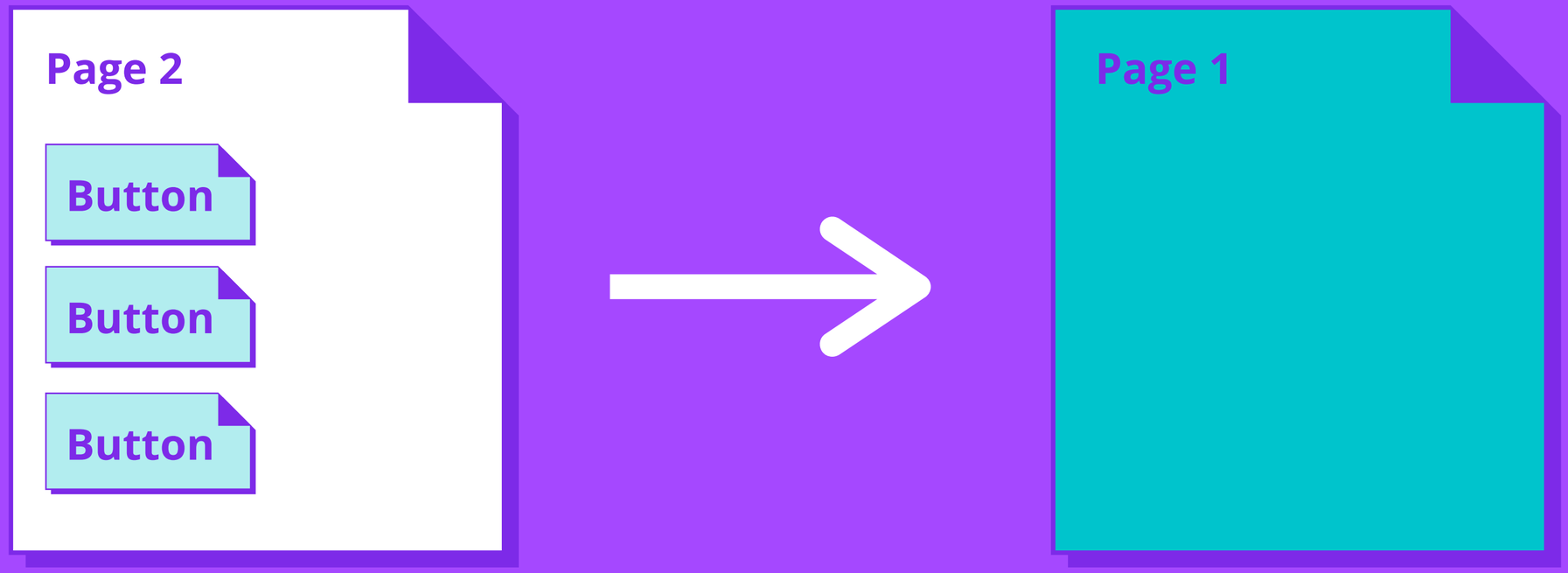
Still seems like a lot more code for the same outcome, why bother?

What have we forgotten?

State machines have a combinatorial amount of **state transitions**. 3 states = 6 transitions.

- Empty -> Page 1
- Empty -> Page 2
- Page 1 -> Page 2
- Page 2 -> Page 1
- Page 1 -> Empty (We can ignore this)
- Page 2 -> Empty (We can ignore this)

Scenario - transition from Page 2 -> Page 1



Imperative VS Declarative

The Imperative Way

```
document.body.style.backgroundColor = "white"
// Now need to keep a mutable reference to our
buttons
let buttonOne, buttonTwo, buttonThree;

// Empty -> Page 2
buttonOne = document.createElement("button");
buttonTwo = document.createElement("button");
buttonThree = document.createElement("button");

document.body.appendChild(buttonOne);
document.body.appendChild(buttonTwo);
document.body.appendChild(buttonThree);
```

```
// Page 2 -> Page 1
if (buttonOne) {
  buttonOne.remove();
  buttonOne = null;
}

if (buttonTwo) {
  buttonTwo.remove();
  buttonTwo = null;
}

// Etc
```

Imperative VS Declarative

The Declarative Way

Declare the expected states and let react figure out how to make the changes.

As a result, the declarative code for handling these cases doesn't change. It **includes all the state transitions.**

```
function App(props) {
  if (props.pageType === 'one') {
    return (
      <body style={{ backgroundColor: 'green' }} />
    );
  } else {
    return (
      <body style={{ backgroundColor: 'white' }}>
        <button />
        <button />
        <button />
      </body>
    );
  };
};
```

Problem 1.

As our application becomes more complex, the total number of possible state **transitions** expands rapidly.

Imperative VS Declarative

50 pages = a minimum of 2450 possible transitions.

When we use a **declarative** approach, we only need to describe the complete UI in 50 states. React manages the transitions.

Problem 2.

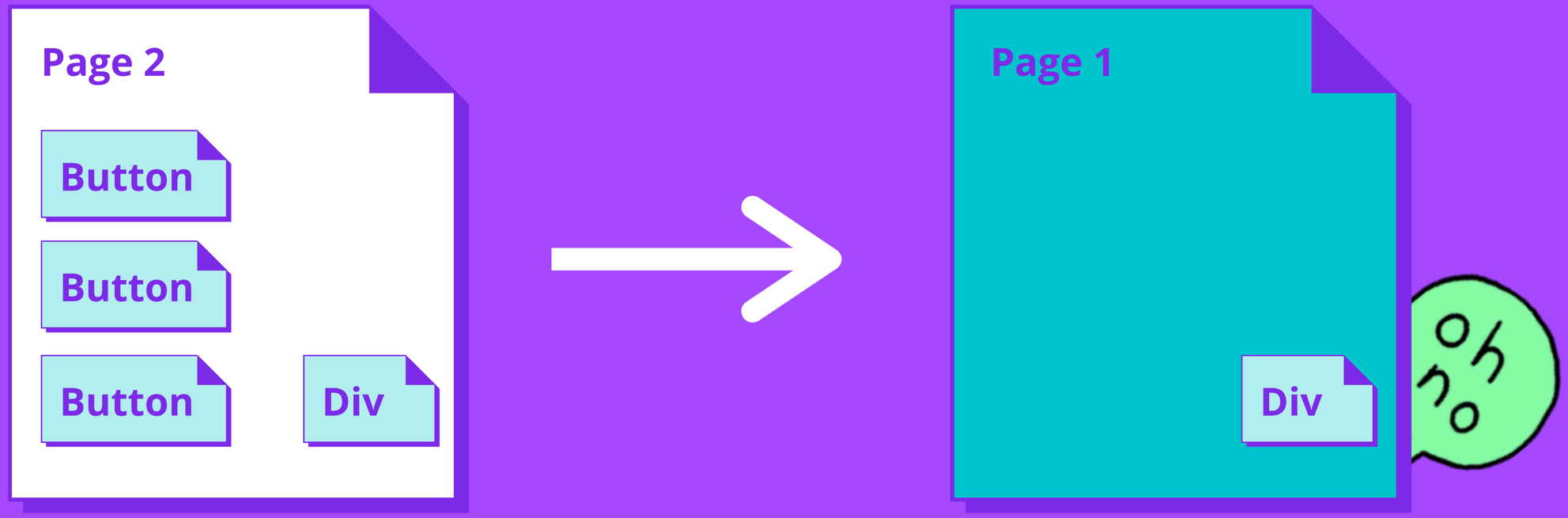
As our team expands, the chance of an untracked UI mutation rises rapidly.

Imperative VS Declarative

Other team: "It would be great to have a ** ** in the bottom right corner of Page 2!"

"This is simple enough that we don't need to tell the team that built Page 2 about it."

Imperative transition from Page 2 ->
Page 1 using the same code as before



Bonus Conundrum

```
<script async src="/script1.js" />
// script1.js contents
document.body.style.backgroundColor = "red";

<script async src="/script2.js" />
// script2.js contents
document.body.style.backgroundColor = "blue";


// What is the background color of body?
```

Imperative rendering

Imperative rendering is very convenient and ergonomic for doing simple things. You don't need to install any libraries, worry about package management or building.

It works great as long as you don't have any data that is external to the DOM.

Data in the DOM

```
<input type="checkbox" id="checkbox"></input>
<button id="button">Button</button>

document.getElementById("someOtherElement")
  .addEventListener('click', () => {
    if (document.getElementById("checkbox").checked) {
        // Do something with the checked box value
      }
    }
);
```

Imperative rendering

As soon as you have data stored in Javascript only (e.g an array of items from the server) you are now trying to keep two separate state machines (your javascript, and the DOM) **in sync** with each other.

You will be squashing edge cases until the heat death of the universe.

Declarative rendering

Let's use logic to build our own simple version of React.

Declarative rendering

Step 1.

We need a naive and dumb way of making sure that the DOM is always in sync with our render function result. This is the "contract" provided by declarative rendering.

Declarative rendering

Idea: Delete the whole DOM and re render the entire thing from scratch any time anything changes (it works)

Declarative rendering

We don't need to think about any state transitions if we always start from scratch.

Declarative rendering

Problem: It's too computationally expensive.

Why is updating the DOM so expensive?

One of the things that makes HTML and CSS awesome to work with is the layout engines. We can very easily make apps that can expand and contract to a range of sizes and orientations.

When we change something, these layouts often need to be recalculated. If we change a lot of things in a short amount of time, it adds up.

Problem Solving

Make our "delete the whole DOM" approach more efficient. Figure out how to only delete and recreate the parts that we actually need.

Declarative rendering

The trick: we don't actually care about the layout when we're making updates. Let the browser figure it out **after** we've made all our changes.

Solution: Keep our own copy of the HTML hierarchy in JS. Make changes against that one to avoid paying the layout costs. Diff the trees and use the diffs to surgically update the DOM.

This technique is known as the **virtual DOM.**

Declarative rendering

State 1

```
<div>
  <button>Button</button>
  <button>Button</button>
  <button>Button</button>
  <div>Div</div>
</div>
```

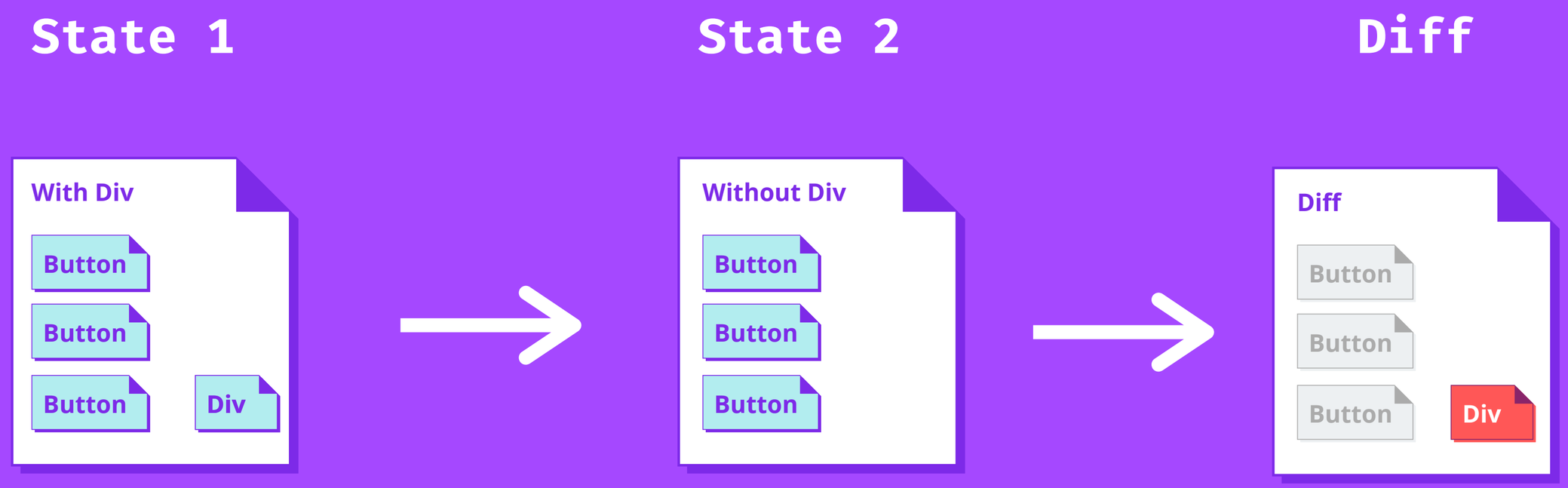
State 2

```
<div>
  <button>Button</button>
  <button>Button</button>
  <button>Button</button>
</div>
```

Diff

```
<div>
  <button>Button</button>
  <button>Button</button>
  <button>Button</button>
  <Div>Div</div>
</div>
```

Declarative rendering



Under The Hood

React looks at the diff and says "this div is being removed and nothing else has changed".

Then It simply calls **removeElement** to delete the single **div** from the DOM, and avoid having to pay the cost of deleting and rebuilding the whole DOM.

Declarative rendering

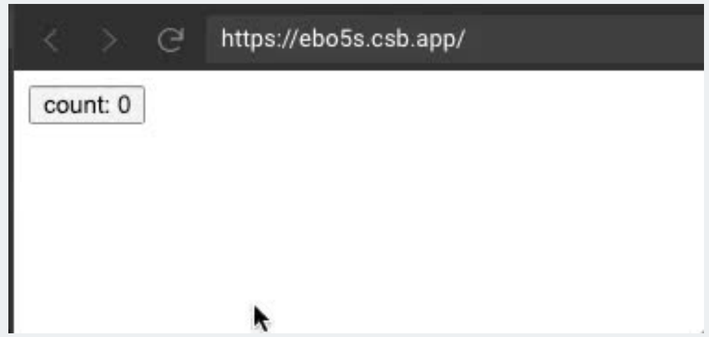
We now understand how to declare states, and how React takes changes in the virtual DOM structure and propagates them to the real DOM.

But how do we actually tell React that we're making changes?

React State

```
function App() {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);

  return (
    <button onClick={increment}>
      count: {count}
    </button>
  );
};
```



Declarative rendering

Run 1

```
<button>
  count: 1
</button>
```

Run 2

```
<button>
  count: 2
</button>
```

Diff

```
<button>
  count: 2
</button>
```

React State

When **setState** gets called:

- The state is updated asynchronously.
- After the state finishes updating, **render** is called again, with the new value of **state.**
- The result of the render function with the new state is what you're asking React to put on the screen, given the new state.

React State

```
function App() {
  const [dropdownVisible, setDropdownVisible] = useState(false);

  return (
    <div>
      <button onClick={() => setDropdownVisible(!dropdownVisible)}>
        Open Dropdown
      </button>
      {dropdownVisible &&
         <ul>
            <li>Dropdown Item 1</li>
            <li>Dropdown Item 2</li>
            <li>Dropdown Item 3</li>
         </ul>
      }
    </div>
  );
};
```

The best way to understand the render -> change state -> render cycle is to play around with some simple components yourself.

Have fun!