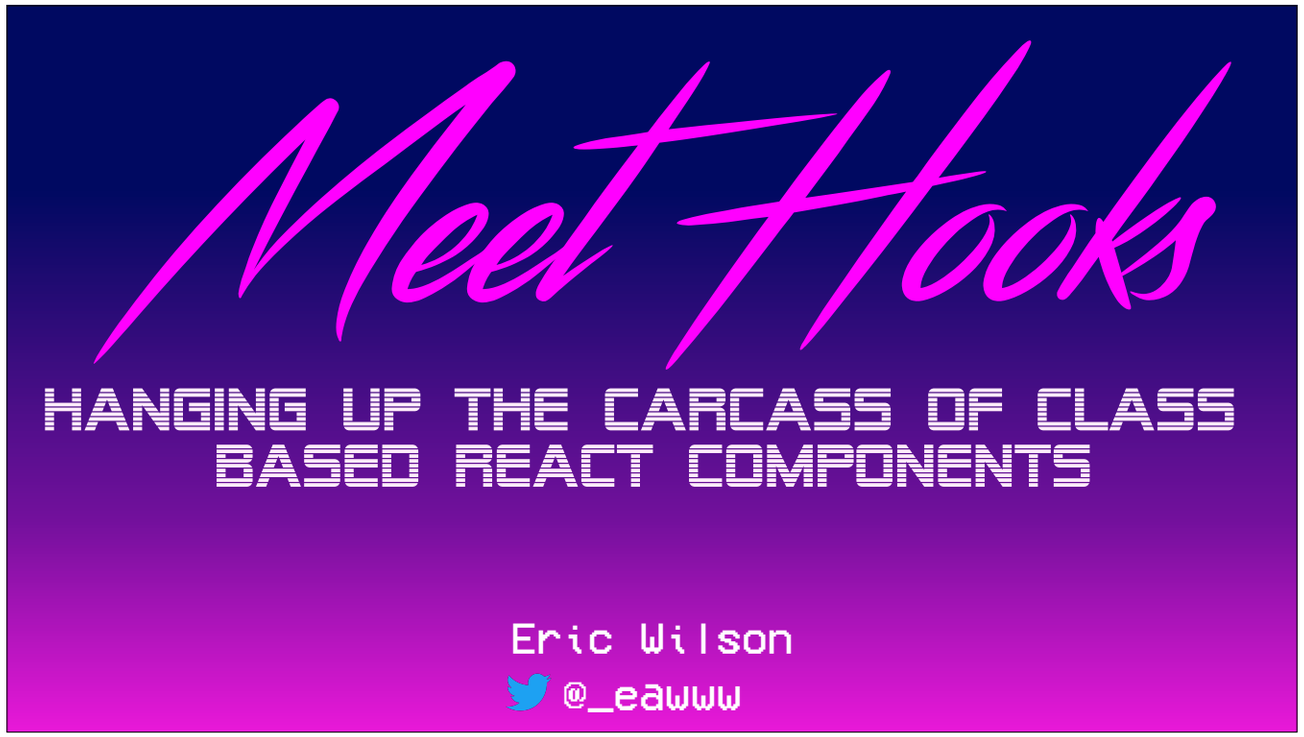


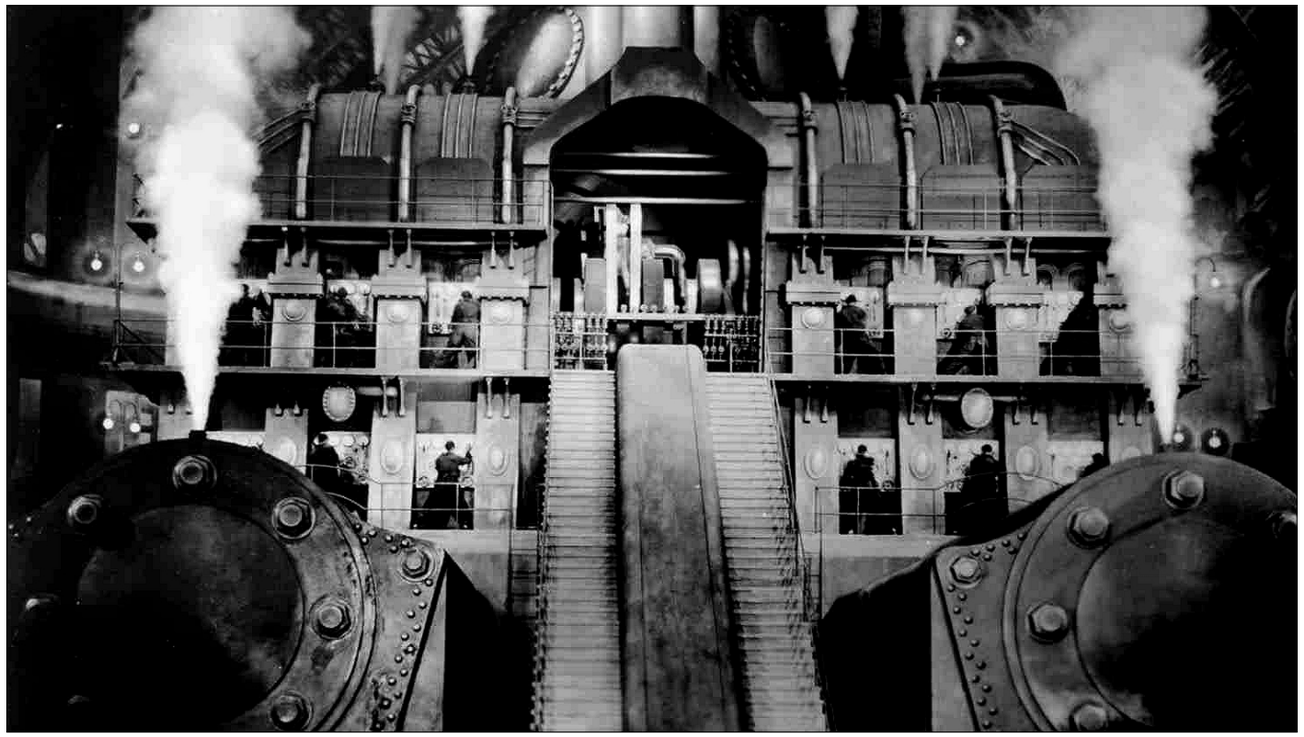
⚠️ This talk assumes a **basic understanding of JavaScript/HTML** fundamentals.

Working knowledge of React is recommended but not required for the brave 👩🏽‍🚀.

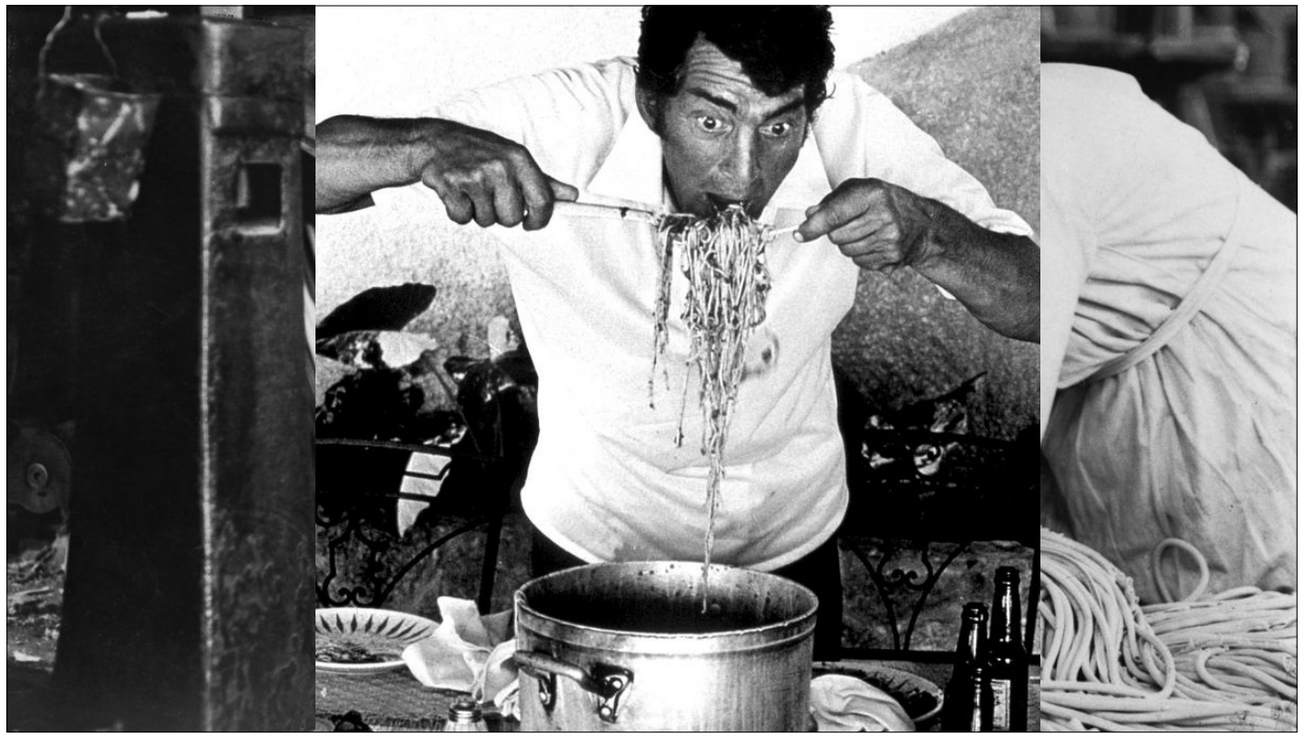
All **materials will be available** online after the talk.

The `class` model for writing React components is a long and storied tradition. For four years (0.2 score), classes have been the de facto solution for implementing stateful logic within React components…

(2015-03-10, React v0.13)



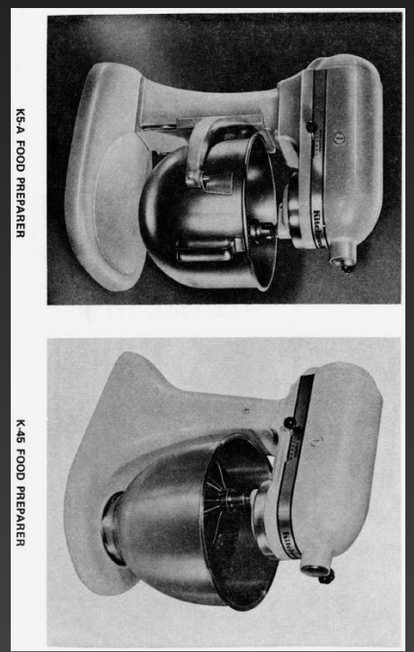
but they can quickly become large and confusing when related code has to be spread across multiple lifecycle methods, each with bits and pieces of unrelated logic within them.



- By themselves, classes have created a marvelous way for us to cook up big ol' bowls of spaghetti code,
-🌟 which sounds delicious...

🌟but not when you're the one who has to eat it. 🍝

- There have been several solutions used to give components the power of stateful logic:



Mixins

- Before classes
- Made a mess of everything

- Mixins were one of the first.
- 🌟They even predate classes.
- 🌟They muddied up the component scope, and simply did not scale well at all.

- That's why their use has been strongly discouraged for at since at least the release of ES6 classes. (2015-03-13)

Higher order Components

2015 - 2019

"export default InjectSadness(DoesntMatter)"

- With ES6 classes, came *Render props* and *Higher Order Components* and grand promises of composability🌟.
- For those who are new to this, Higher Order Components are an extra component that you wrap around the component you're actually using to implement some desired logic and inject props
- **Higher Order Components** worked and, along with classes, they solved a lot of the problems of mixins.🌟

- As this pattern became more popular, and developers began to rely on them more and more, component hierarchies began to turn into what has become affectionately known as🌟 🔥wrapper hell🔥…
- so called because a component hierarchy should be an easily consumable view of how an app is organized, but patterns like these make them overwhelming, unpleasant…
- The most recent and, in my opinion *best* attempt at creating cohesive, reusable stateful logic in React is the new…

HOOKS

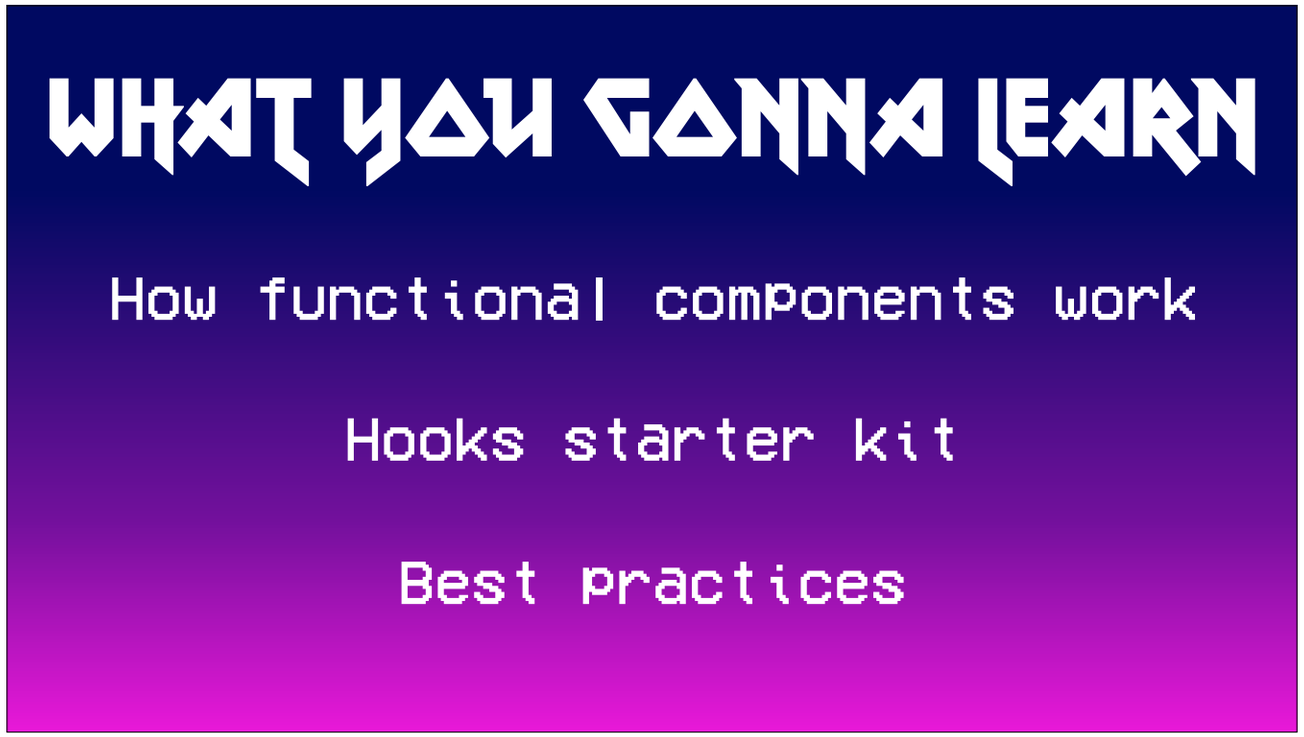
- …**Hooks API**.

- 🌟Hooks were initially proposed in an RFC on October 25th, 2018 and announced at React Conf 2018 the same day in back-to-back talks by these fine folks🌟 on behalf of the ReactJS team.

https://www.youtube.com/watch?v=dpw9EHDh2bM

- After months of enthusiastic feedback from developers, and hard work by contributors, exactly 4 months ago to the day, 🌟Hooks became an official part of React on Feb 6, 2019 in React 16.8.0

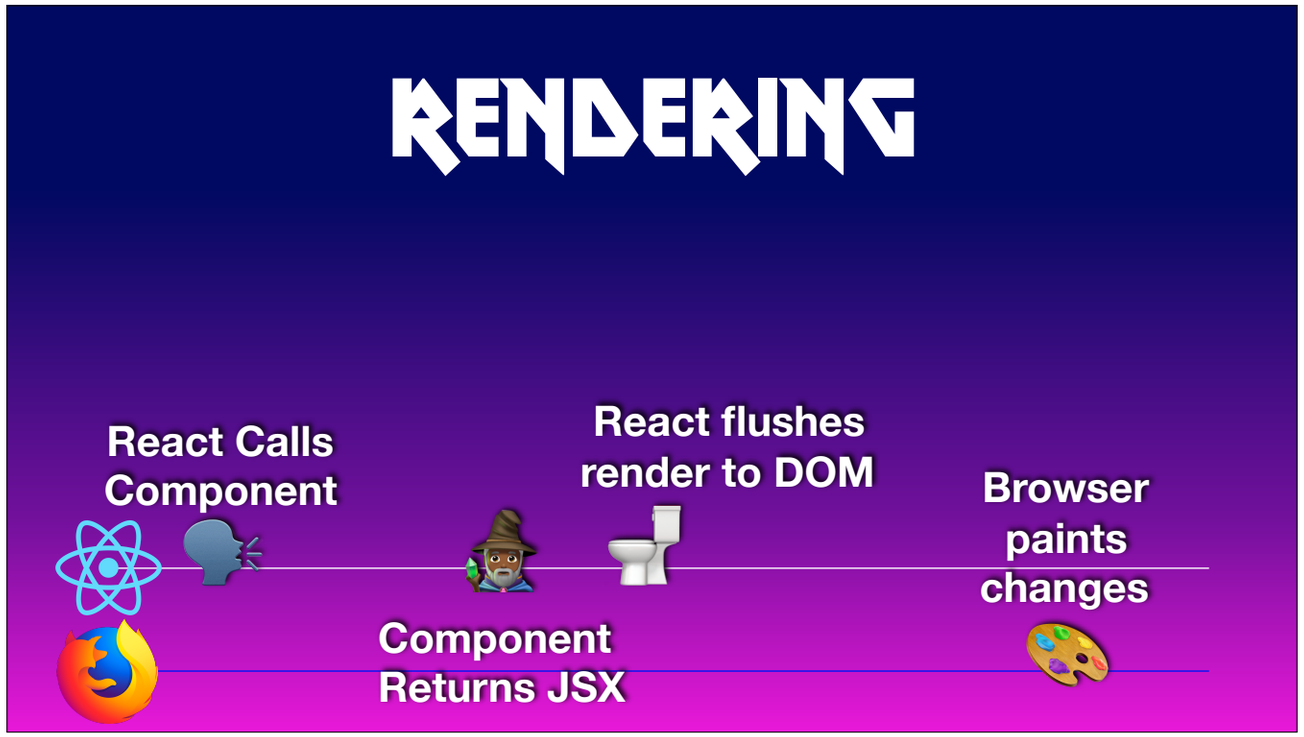
- Hooks are still new so little bugs, best practices, and support from some third party libraries are still being worked out.🌟
- Despite this, since their release, Hooks have become incredibly popular and have lived up to the hype.🌟
- There's already countless blog posts, instructional videos,and talks on the subject but *this* is the only one at Cerner DevCon KC 2019.🌟



- Thanks to Hooks, almost everything you need to know to develop an entire react application can be broken down into three basic categories:
    - How functional components work🌟
    - The most important Hooks🌟
    - Best practices🌟
- Let’s start with how functional components work…

- A functional React component has three fundamental characteristics: 🌟
  - It's defined as a function (that's why they call them *functional*)🌟
  - Accepts a `props` object as an argument🌟
  - Returns JSX which is a syntax extension to JavaScript but you can think of it conceptually as magic HTML 🌟 that *lives* in JavaScript.

- Furthermore, just like any function, a functional component can 🌟do stuff before it returns a value.
- Functions that return JSX can be used as JSX elements in 🌟 other components. Just like in HTML.

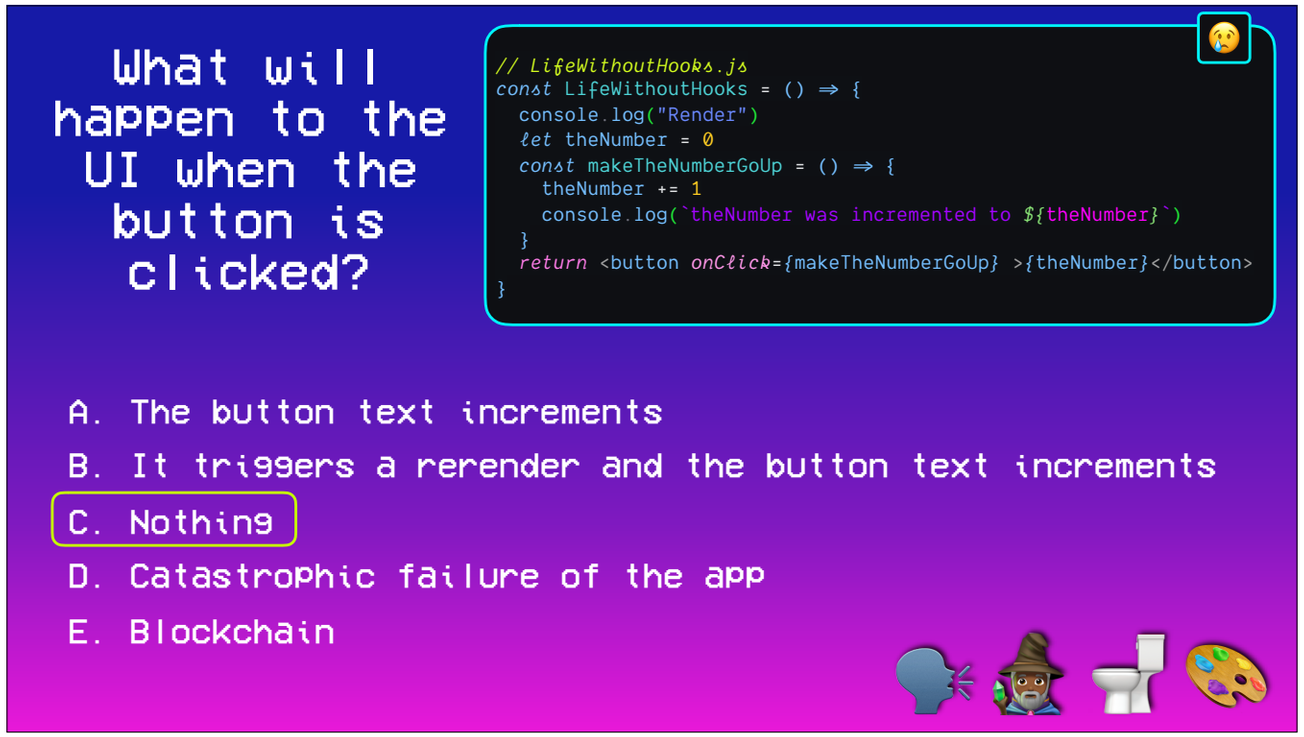


- When a component is rendered or re-rendered, its function is called 🌟along with the functions for all of its child components all the way down the tree.
- The 🌟component returns its JSX and
- Any and all changes that the render causes will be reflected in React's Virtual DOM
- which React will then use to decide which changes should be 🌟🚽 flushed to the actual DOM to be 🌟🎨 painted by the browser.
- A very important thing to know going forward is that each time a component is rendered and its function is called, everything defined within the function, belongs to that function call. Each render has its own state, props, definitions, etc…

```
// LifeWithoutHooks.js
const LifeWithoutHooks = () => {
  console.log("Render")
  let theNumber = 0
  const makeTheNumberGoUp = () => {
    theNumber += 1
    console.log(`theNumber was incremented to ${theNumber}`)
  }
  return <button onClick={makeTheNumberGoUp} >{theNumber}</button>
}
```

- To illustrate this, consider the following functional component:
- This component:
    - 🌟logs "*Render*" to the console when it's called, will tell us each time React calls the function
    - 🌟initializes a variable called `theNumber`,
    - 🌟defines a function which will increment theNumber and then output its value
    - 🌟and returns jsx for a button that calls our function and displays the value of theNumber

I'll give you a minute to look that over. And don't worry. It's on the next slide, just smaller. ⏳🌟



- What will happen when the button is clicked? …
- [read options]
- 🌟Here's the order of things just to remind you …
- I'll give you a minute ⏳🌟
- NOTHING! Here. I'll prove it.👨‍💻

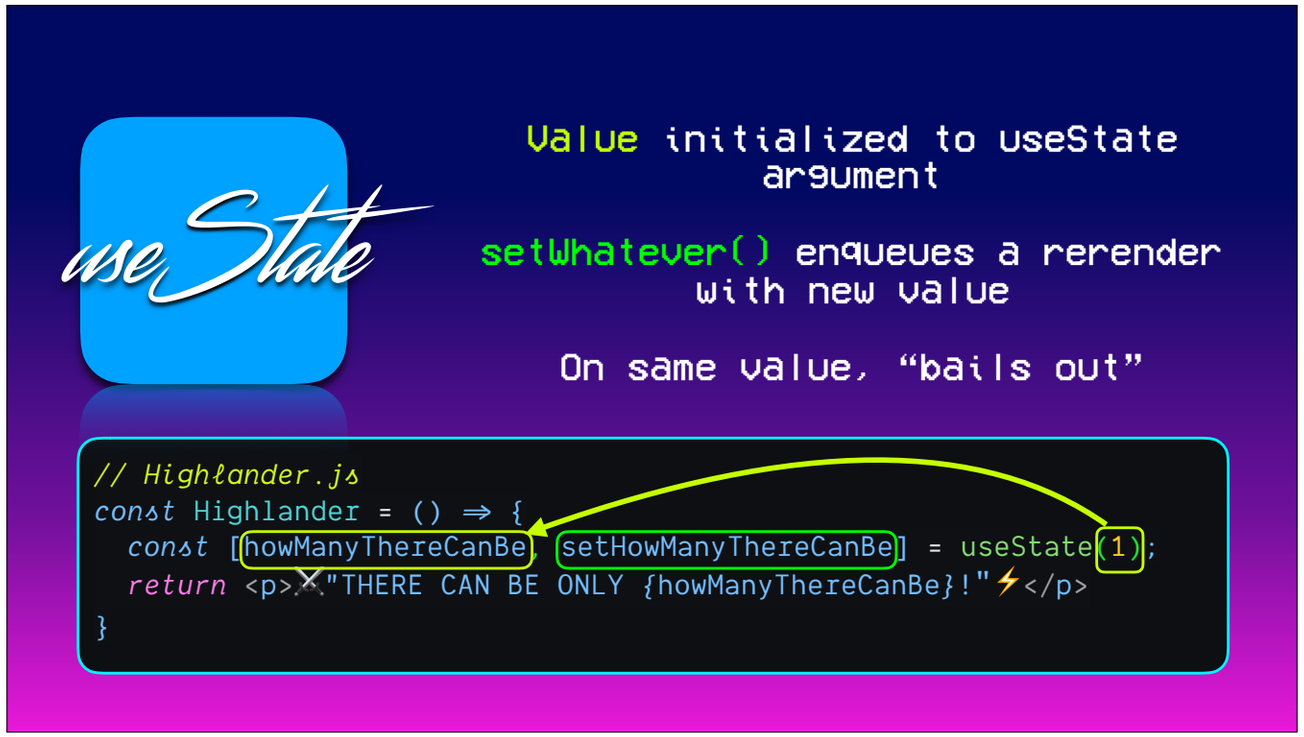
```
// LifeWithoutHooks.js
const LifeWithoutHooks = () => {
  console.log("Render")
  let theNumber = 0
  const makeTheNumberGoUp = () => {
    theNumber += 1
    console.log(`theNumber was incremented to ${theNumber}`)
  }
  return <button onClick={makeTheNumberGoUp} >{theNumber}</button>
}
```

- The `theNumber` value does indeed increment as can be seen in the browser console. However, since
- React calls the function once for JSX and ignores it until the parent says it needs to rerender, all of that incrementing is going on within the scope of the function call after React is already done with it.
- When the component does rerender, the new render's theNumber value is initialized to 0 is returned in the JSX and React never sees it increment.🌟

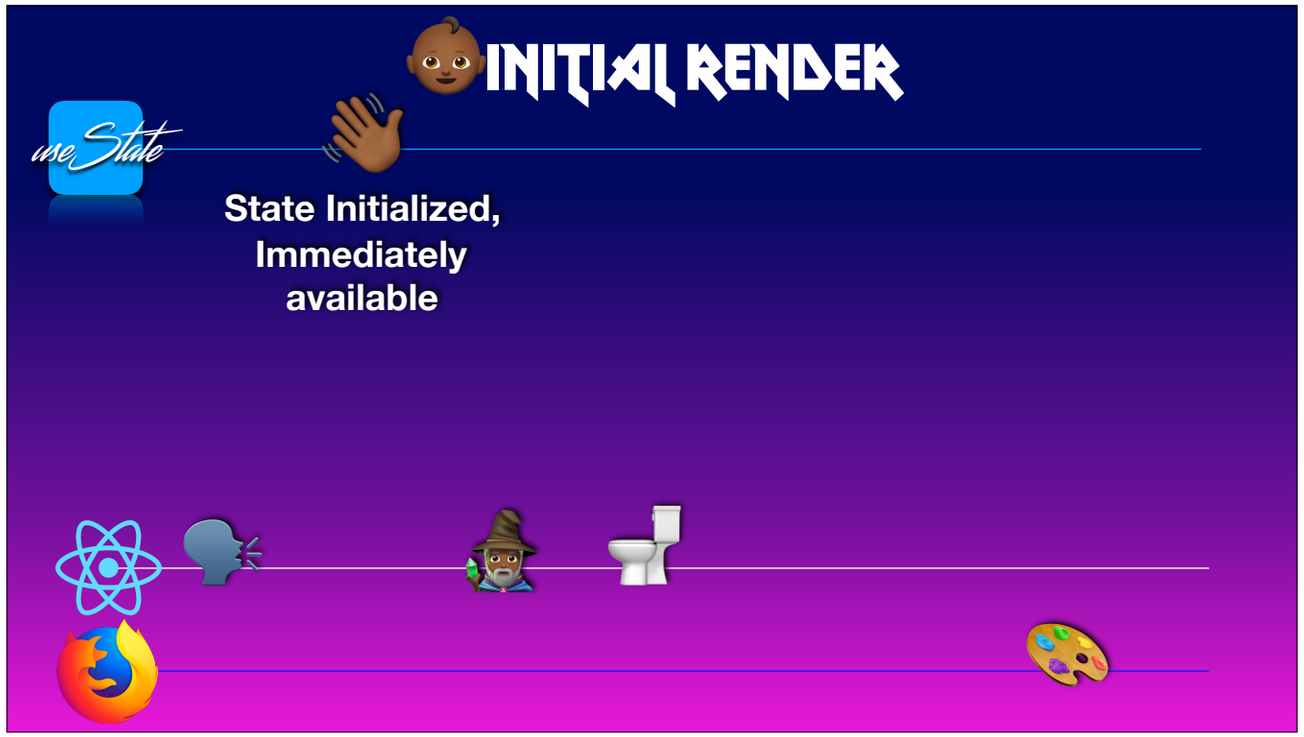
- Now, I'd like to introduce you to the hooks.
- I won't have time to cover all 10 distinct hooks that are currently part of React's API, but I'll try to cover the ones I feel are most valuable to get started with.
- For each one, I will:
    - explain its purpose, 🌟
    - how it works, 🌟
    - and show you some common use cases. 🌟

I've also found that, when working with hooks, it's incredibly valuable to know the order everything executes in relative to other hooks. Details of this are a little lacking in the standard documentation so I'll pay special attention to helping you leave with a conceptual understanding of how your hooks behave relative to the render cycle and to eachother. 🌟

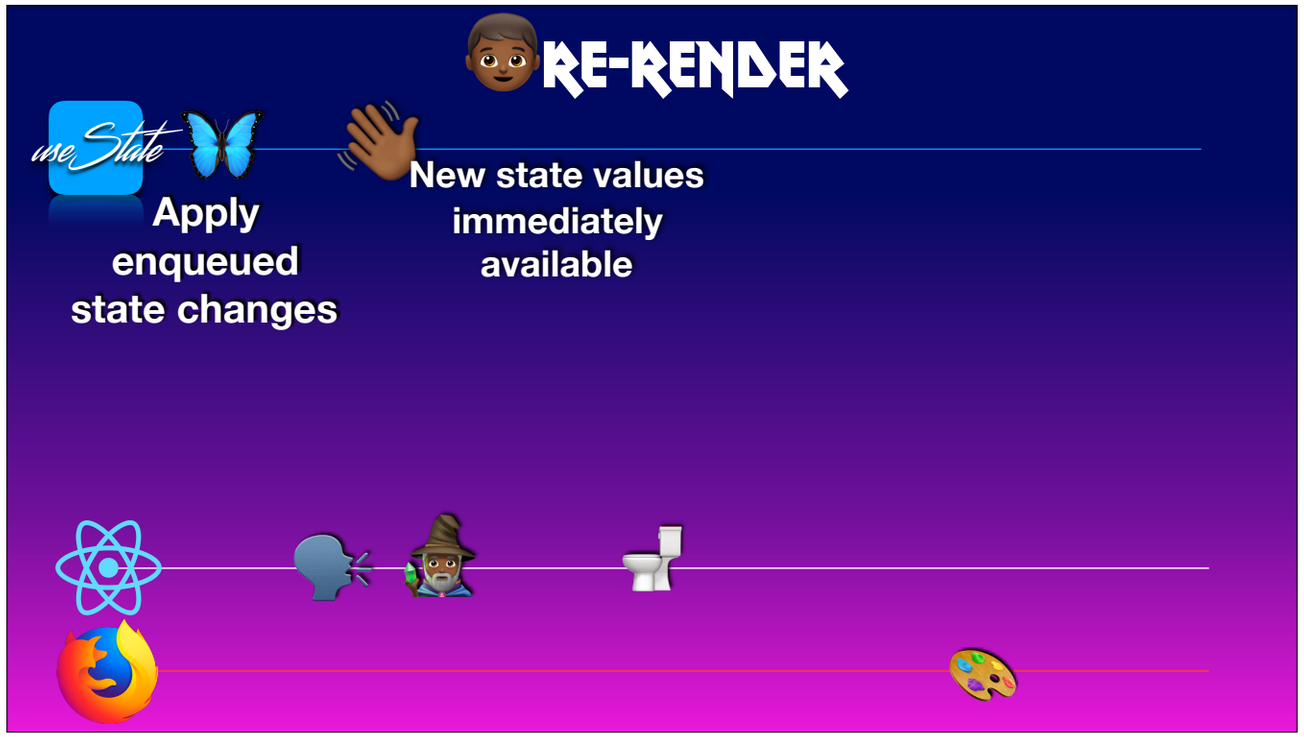
- In the previous example we saw exactly why functional components can't have stateful values of their own. Now we'll see exactly how they *actually can* have stateful values of their own!
- Easily the most important hook is the 🌟 `useState` hook... which looks like this: 🌟
- This hook defines a **stateful value**🌟 and a **function**🌟 for setting that value, returned in a pair.
- Convention is to assign these via array destructuturing,🌟 as seen, so that so that the `set` function is the name of the value🌟 prefixed with the word `set` in camelCase.



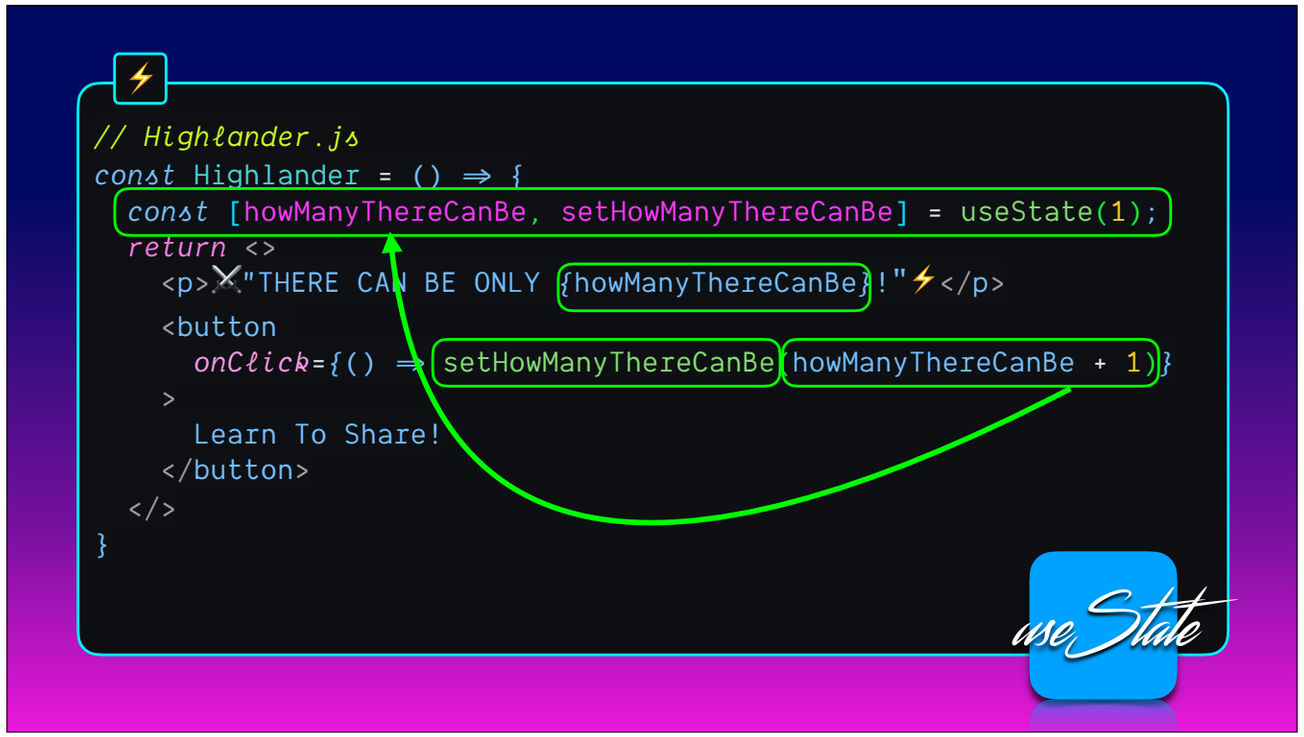
- On the initial render, 🌟the stateful value is whatever is passed as an argument to `useState`.
- Calling the `set` 🌟function enqueues a rerender of the component replacing the stateful value with the new value.
- If it's the same value, 🌟the render will "bail out," meaning that it may re-render itself, but won't rerender deeper into the tree.



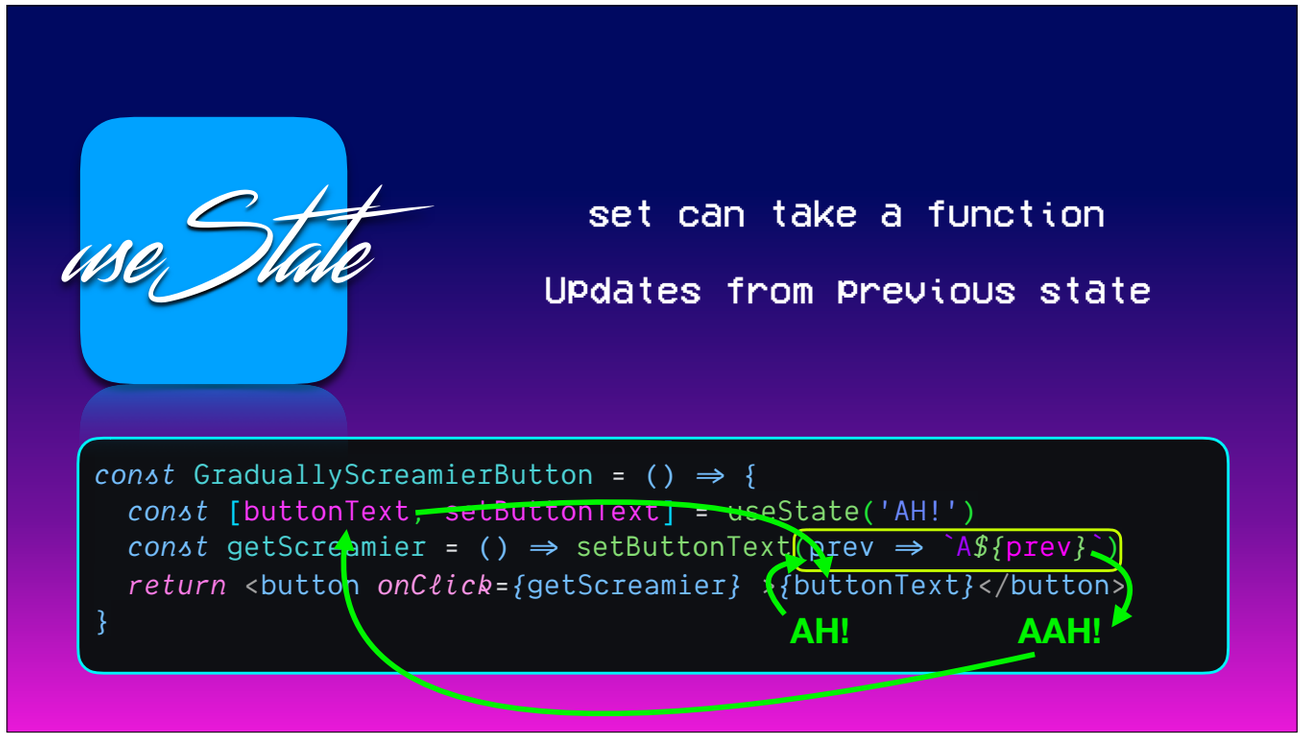
- 🌟On initial render, the stateful value returned by useState is available as its initial value as as soon as it's called.
- This means other code that comes after it in the function can derive stuff from its initial value.



- The stateful values for useState hooks are determined by calls to their setters before the function is called.
- React knows these values and tells each call to a hook to return its new value in the same way it returned its initial value.

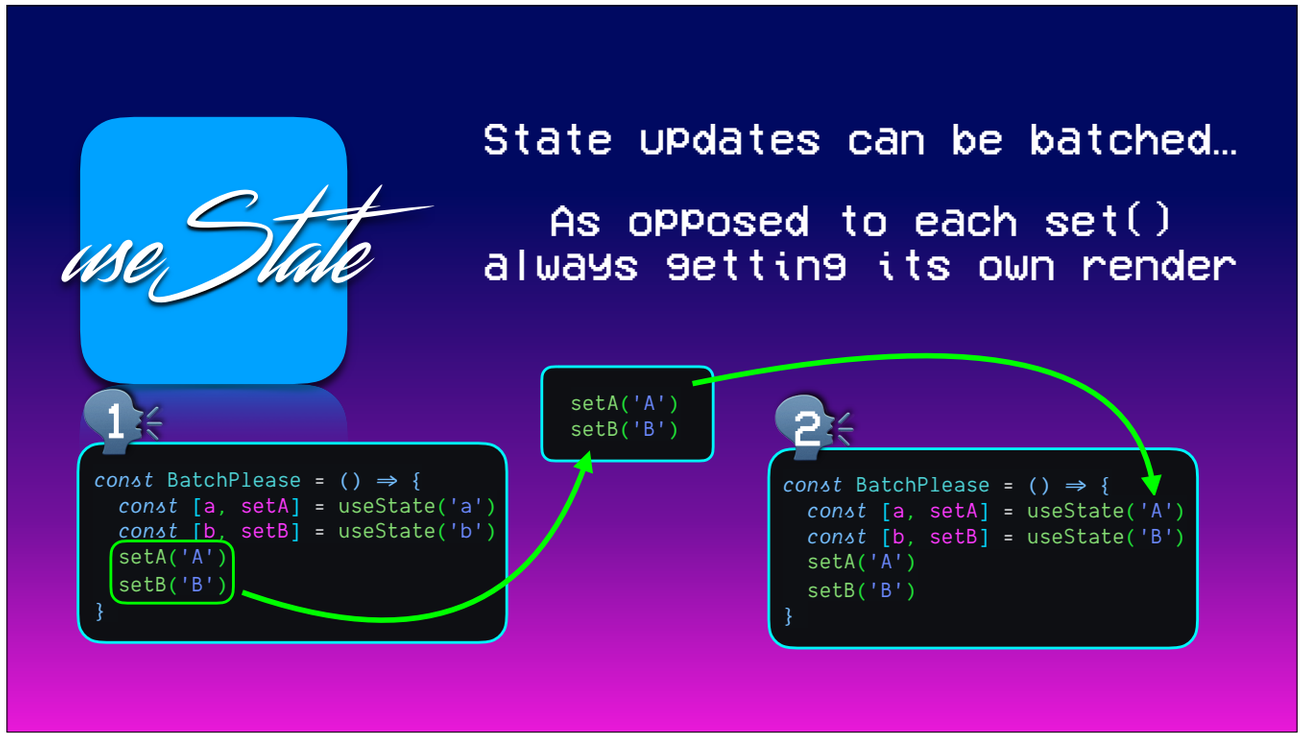


- Let's check out a beefed up version of our Highlander example:
    - We use useState 🌟 to define howManyThereCanBe and a function for setting it.
        - Initial value is, of course, 1
    - Using a template literal, we render "THERE CAN BE ONLY" and then our value 🌟
    - Then we render a button that calls our set function 🌟 with one more than the value. 🌟
        - Which enqueues a re-render with the new value
- Then our component re-renders with the new value! Let's check it out 👨‍💻

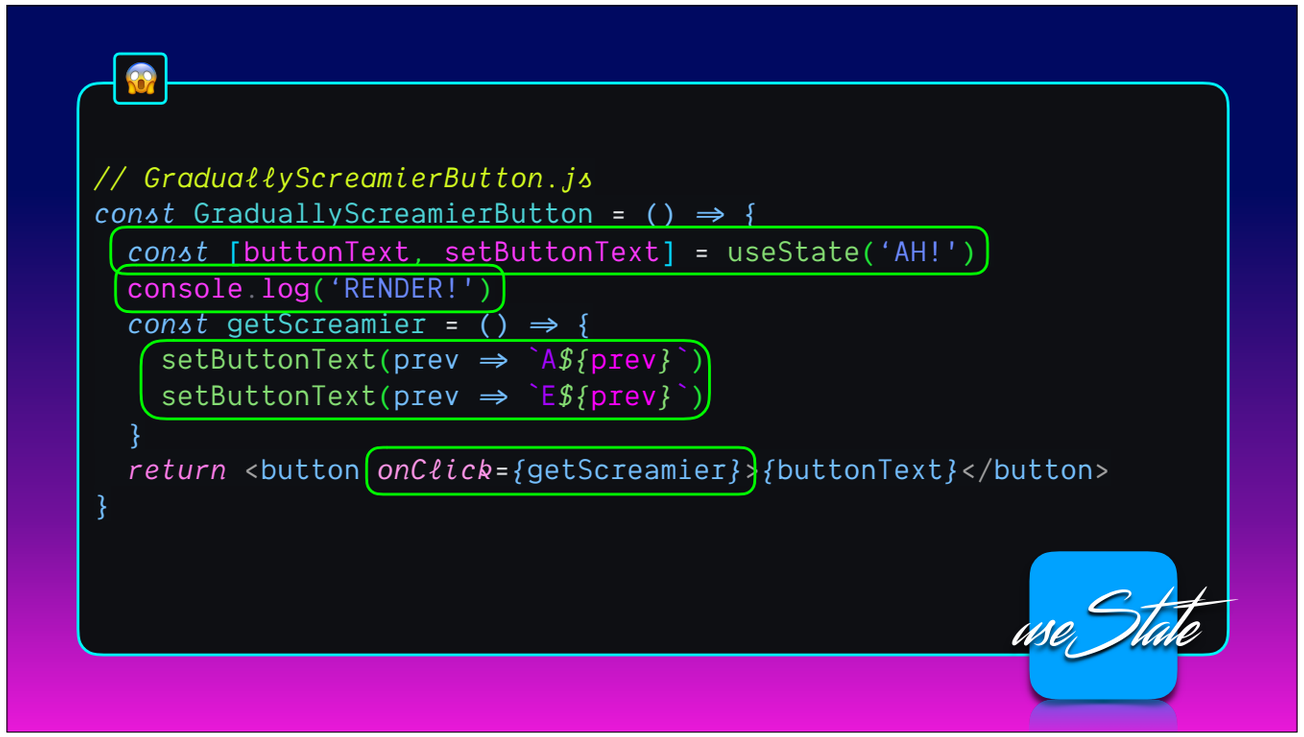


- A lot like `setState` in class components, the `set` function returned from `useState` can accept a function 🌟 to update state based on previous state 🌟:
- The first time the getScreamier function is called, the function will get the previous value 🌟 “AH!” And trigger a re-render with that same value with another A prepended 🌟
- When the component rerenders, ‘AAH!’ Will go into buttonText 🌟 which will then be returned in the JSX 🌟 as the text of the button

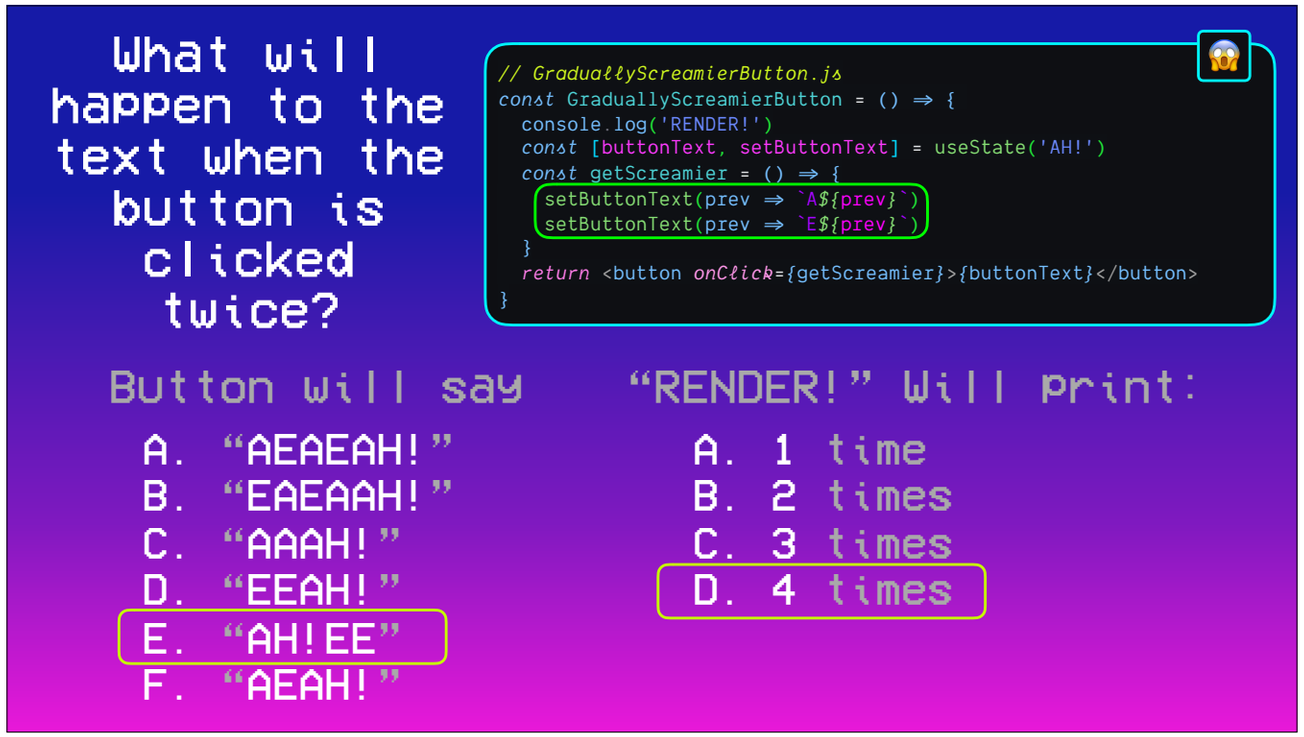
[❗ Maybe do two examples. One using the function updater and one not]



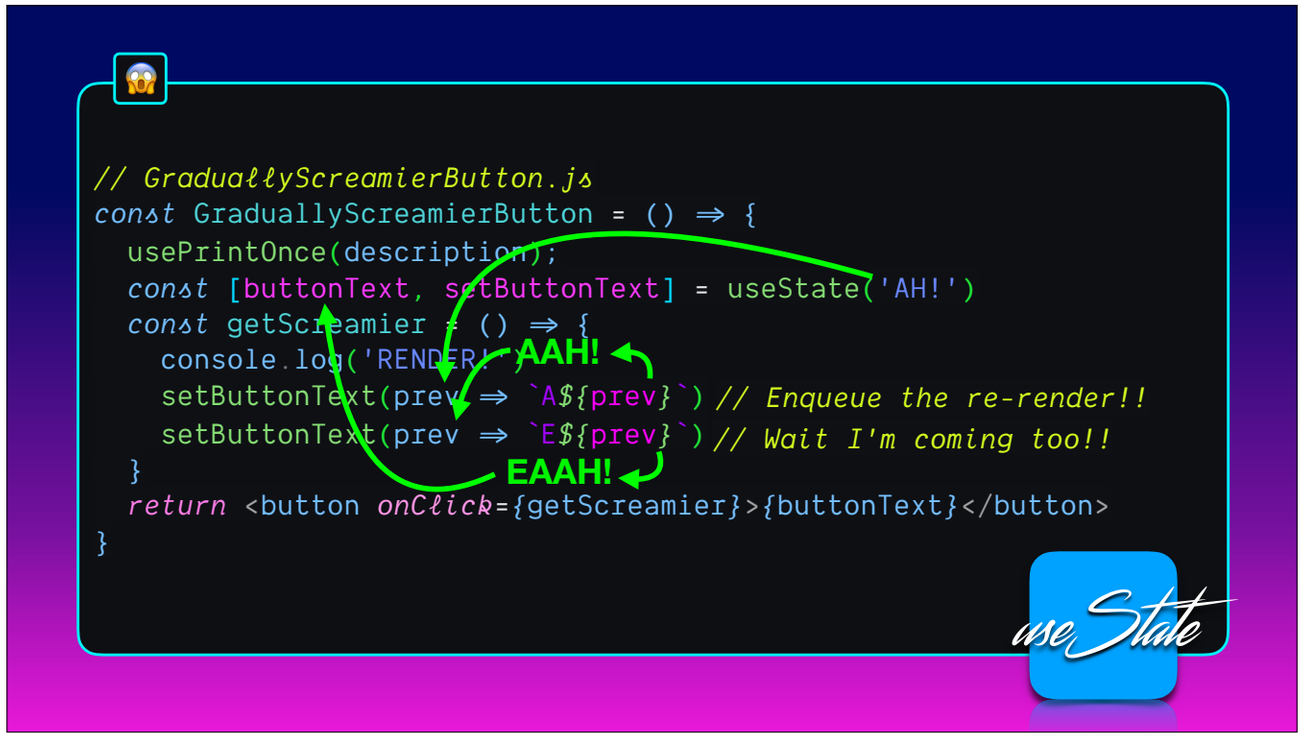
- Also a lot like `setState` in class components, updates to `useState` values at around the same time will be batched together 🌟 so that they're applied to the same rerender. 🌟
- This important optimization can immensely reduce the number of rerenders.



- As an example of how `useState`'s update batching works, consider the following bit of code:
- We define a stateful value, `buttonText` with `useState` to keep track of a text string. We define a function that logs "RENDER!" to the console and then calls `setButtonText` twice in a row: The first time prepending an 'A' to `buttonText`, the second time prepending an 'E' to `buttonText`. The component returns a button element that calls our function on each click.

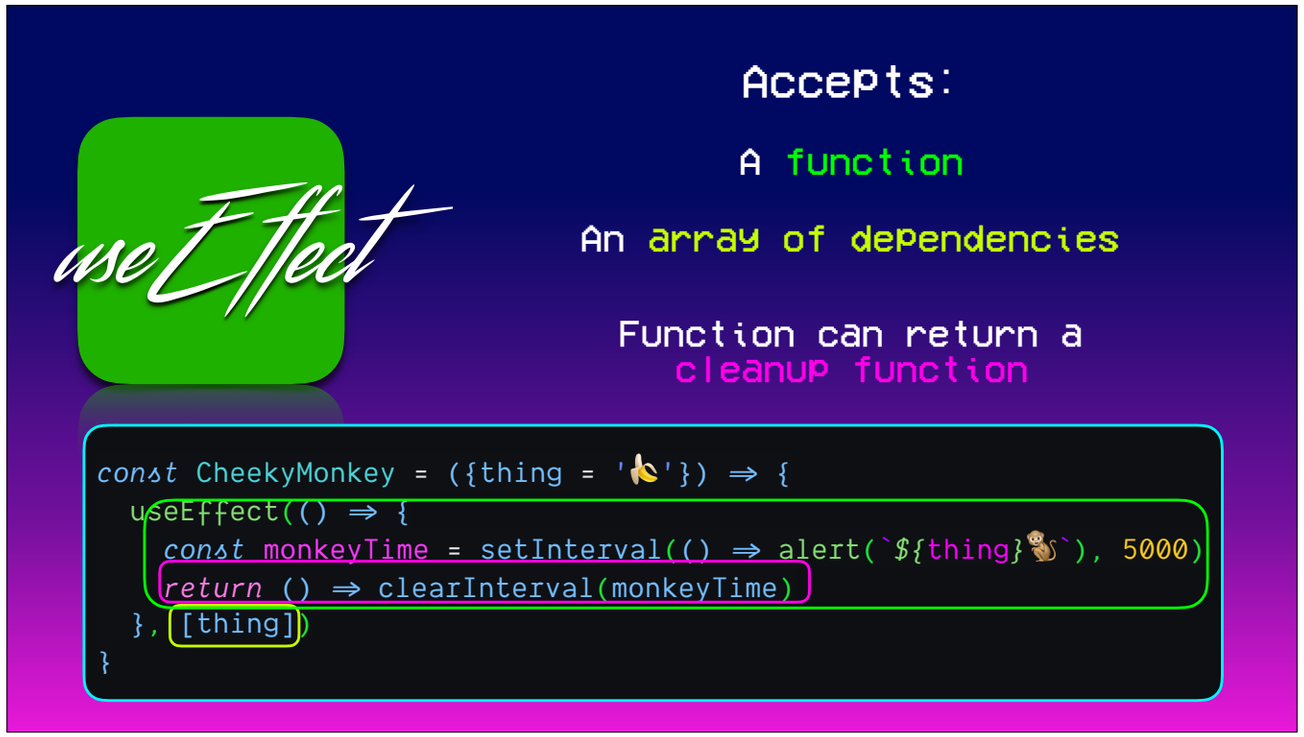


- Since setButtonText *enqueues* a rerender each time it's called, and it's called twice in a row appending a different value to the beginning of the previous value, which of the following behaviors would you expect? 🌟 🌟 🌟 🌟 🌟…
- 🌟 This is the important part.
- If you guessed **E** and **D**, you are the furthest from the right answers... 🌟 which are **B** and **B**. The resulting scream will start with **E** and render once per click.
- Here, I'll even prove it.👨‍💻😱

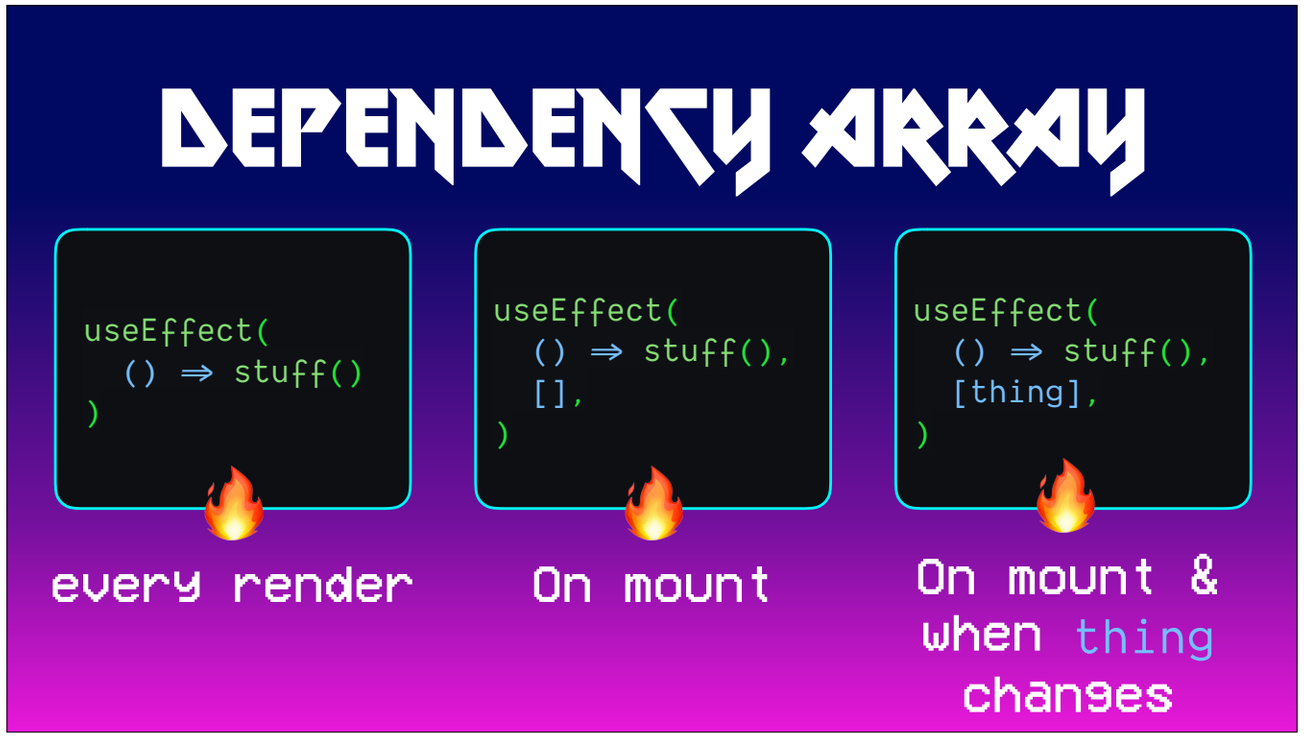


- This is because the first `setButtonText` enqueues the rerender, 🌟 but the rerender isn't instantaneous and JavaScript continues execution on this instance of the component. So when the second `setButtonText` enqueues a rerender to prepend an 'E' to our scream, 🌟 it does so after the 'A' is prepended but it still updates on the same render. 🌟

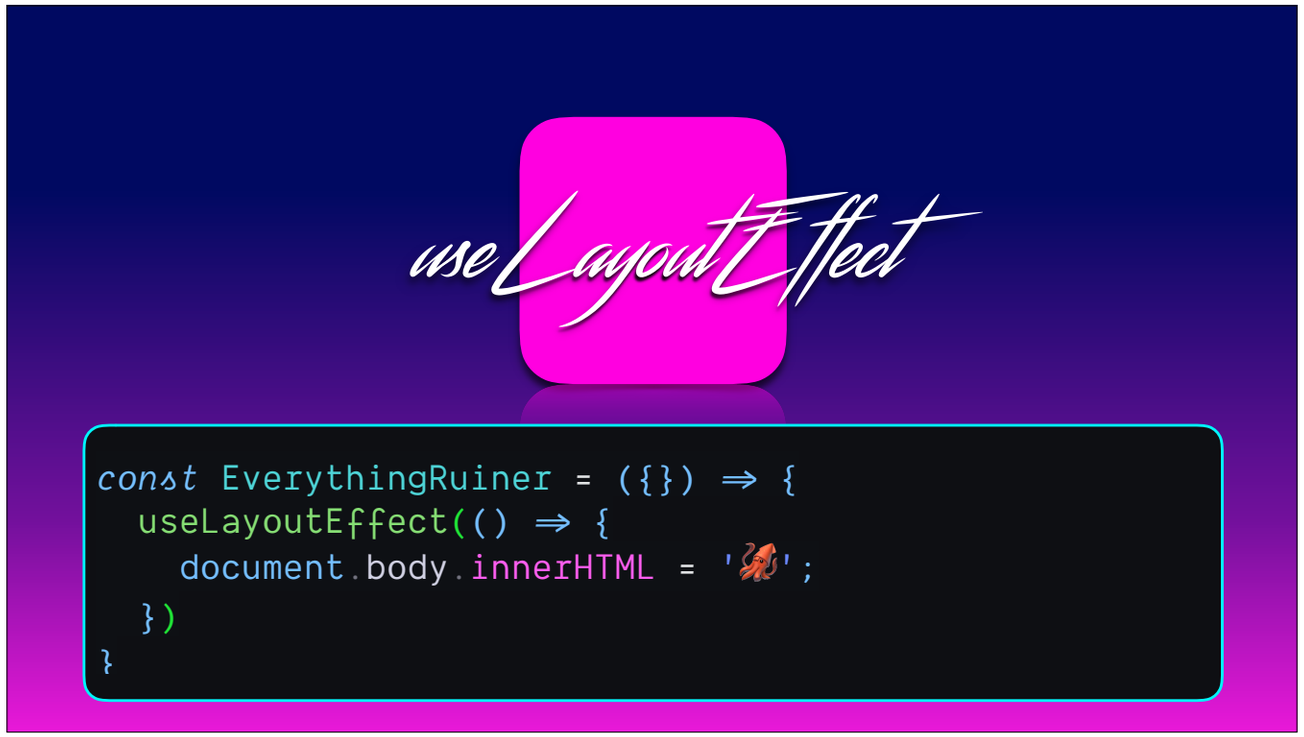
- Now that you're aquainted with `useState`, you just *have* to meet it's best friend, `useEffect`.
- `useEffect` doesn't return anything. Its purpose is to execute imperitive side-effects in functional components: stuff like data fetching, subscriptions, manual DOM mutation, logging, etc…



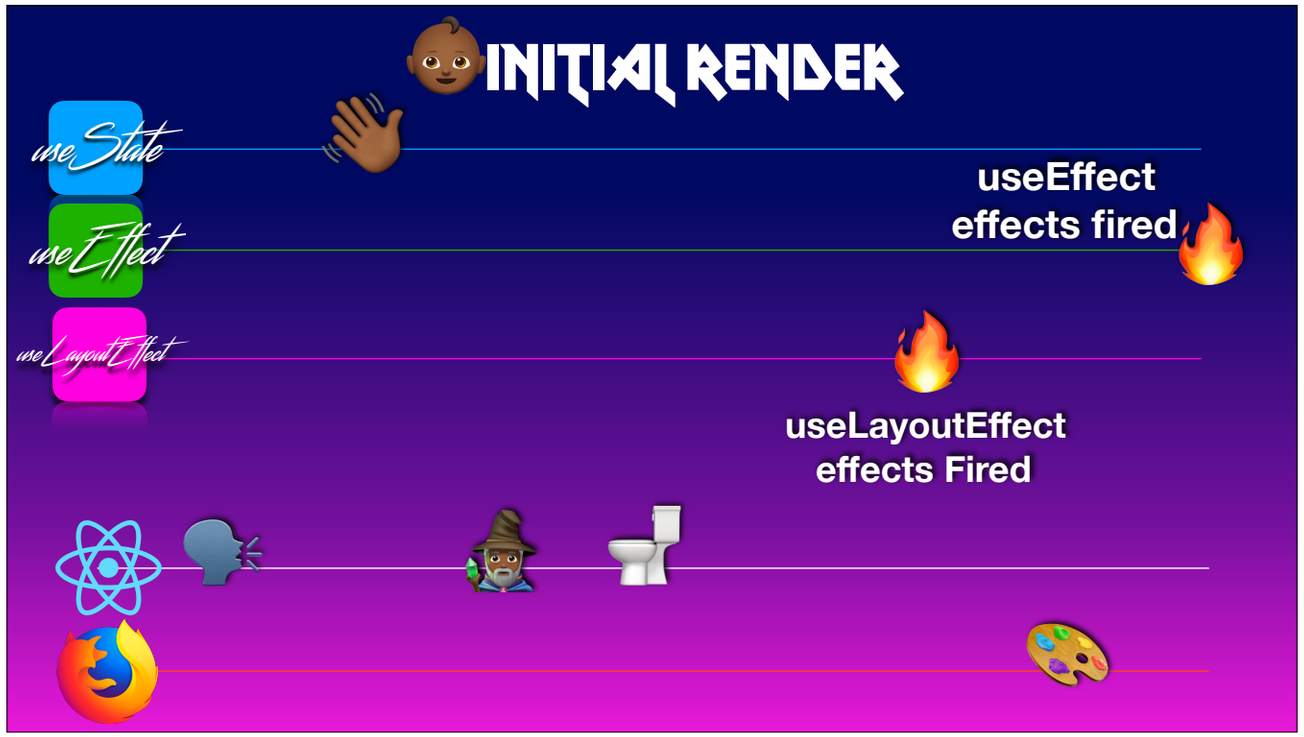
- `useEffect` accepts two arguments:
    - a function to be called
    - and an array of values that function depends on.
- The function can return a **cleanup function** that is used to undo any side-effects that the function may cause.



- The second argument tells react when it should re-run the effect function.
- There are three basic options when it comes to the dependency array:
    - Not passing one
    - Passing an empty array
    - Passing a non-empty array
- They have the following effects: …

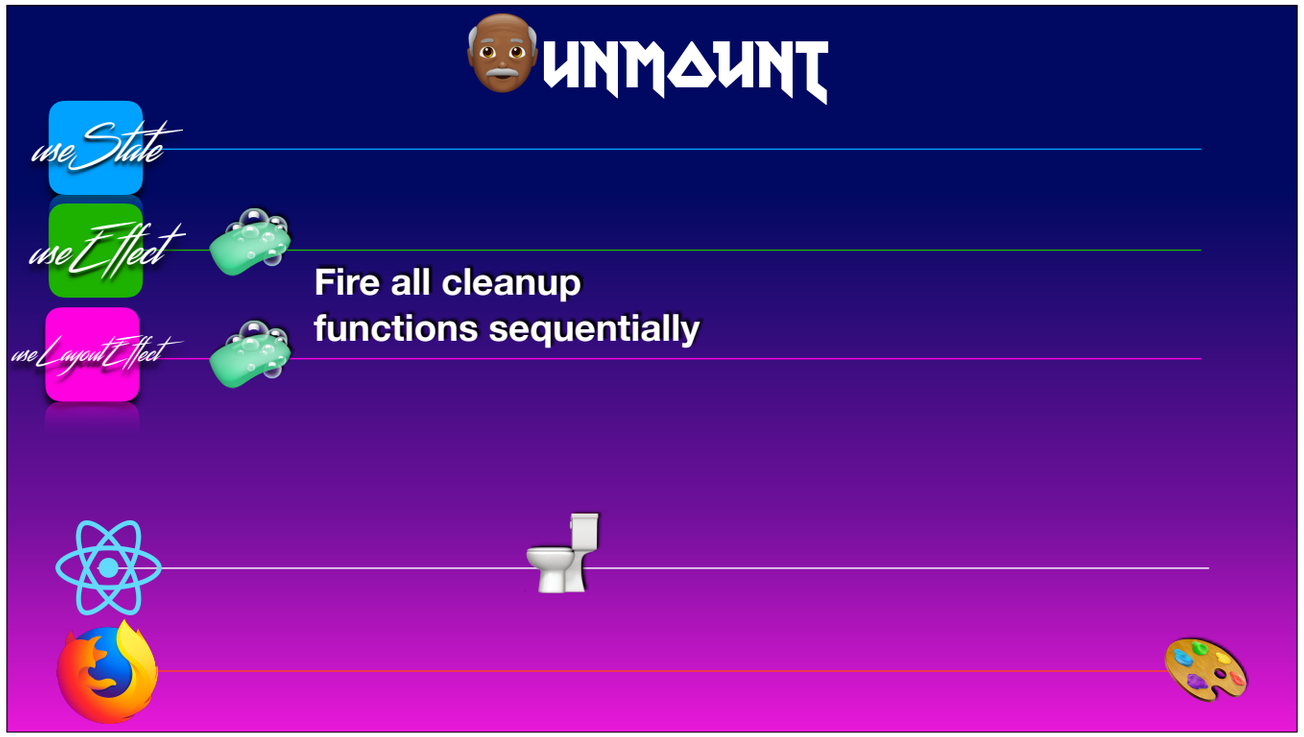


- There is another hook, `useLayoutEffect` that is identical to `useEffect` but fires synchronously after DOM changes and prevents the browser from painting until it's complete. This is useful for side-effects that affect the UI to avoid messy rerenders.
- `useEffect` is preferred where possible
    - so as not to block visual updates.



- Let’s check out the order these things happen in.
- As I mentioned, useLayoutEffect’s effect is fired synchronously before the browser paints. So it goes here. 🌟
- useEffect waits until all the rendering and painting is done and THEN 🌟 it fires its effects. 🌟

- For both useLayoutEffect and useEffect, it's guaranteed that their cleanup functions will fire before the effect fires again. 🌟
- For re-renders, the effects fire right after their cleanups. 🌟



- Interestingly, when a component unmounts, all of its cleanup functions for BOTH useLayoutEffect and useEffect are fired in the order they were defined and before even the changes are flushed to the DOM.

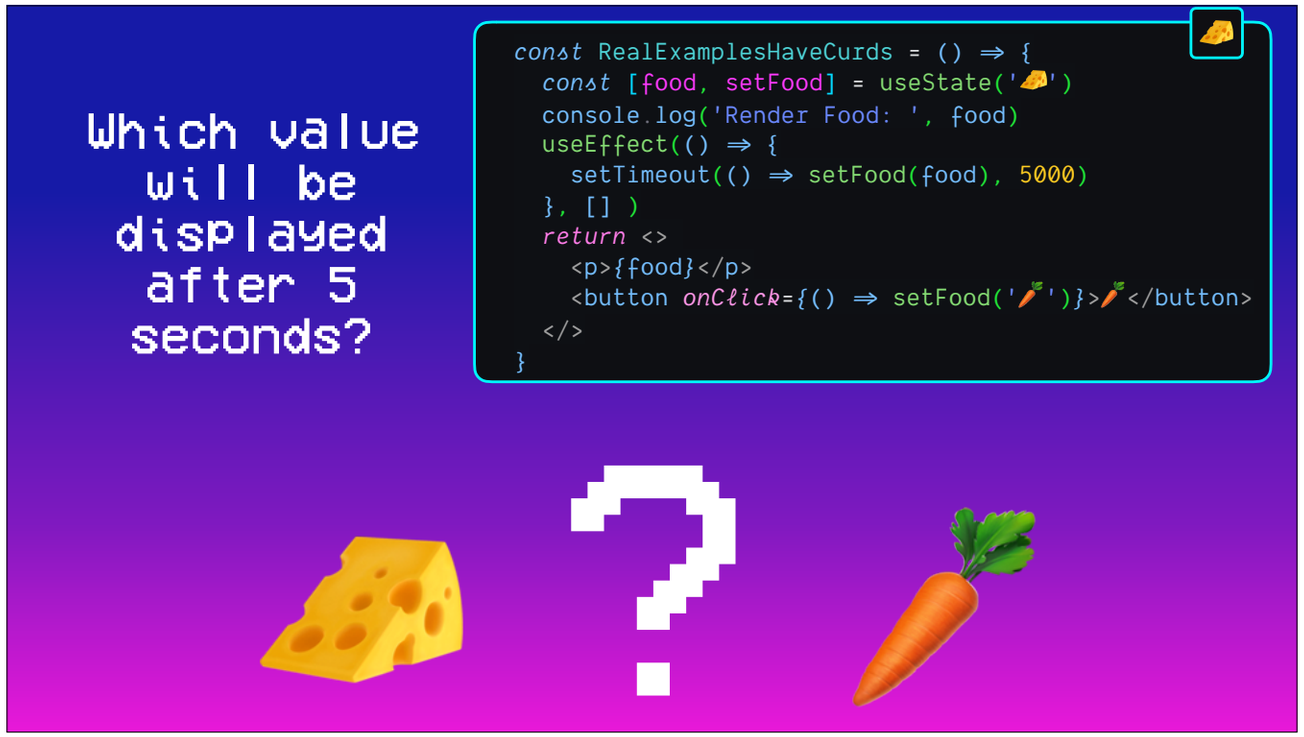
```
const EffectUser = () => {
  console.log('Render!')
  const [garbageValue, setGarbageValue] = useState('🛢️')
  const [importantValue, setImportantValue] = useState('🧐')
  useEffect(() => {
    console.log(`🚀Fire the side effect! ${importantValue}`)
    return () => console.log('🧹Clean up the side effect!')
  }, [importantValue])

  return <>
    <div>
      <p>{importantValue}</p>
      <button onClick={() => setImportantValue(prev => prev + '🧐')}>
        More important!
      </button>
    </div>
    <div>
      <p>{garbageValue}</p>
      <button onClick={() => setGarbageValue(prev => prev + '🛢️')}>
        More garbage!</button>
    </div>
  </>
}
```

- This example looks hefty but it's really pretty simple. It exists only to show how the dependency array works.
- We define 2 stateful values with useState: garbageValue🌟 and importantValue🌟.
- For each one, there's a paragraph element that shows the value🌟, and a button that adds an emoji to it🌟
- They'll both trigger a rerender. The only difference is, we have an effect that depends on importantValue.🌟 Let's check it out: 👨‍💻🛢️ You can see that the effect only fires when importantValue is updated.

```
const RealExamplesHaveCurds = () => {
  const [food, setFood] = useState('🧀')
  console.log('Render Food: ', food)
  useEffect(() => {
    setTimeout(() => setFood(food), 5000)
  }, [] )
  return <>
    <p>{food}</p>
    <button onClick={() => setFood('🥕')}>🥕</button>
  </>
}
```

useEffect

- Remember how all values defined in a functional component only belong to that render? useEffect gives us a tool to demonstrate that pretty clearly.
- In this example we define a stateful food value and initialize it to cheese.
- On each render, we log the food.
- Our effect (runs on mount because [ ]), setting a timeout to set our food state to itself after 5 seconds
- We render our food and a button that immediately setsFood to carrot.



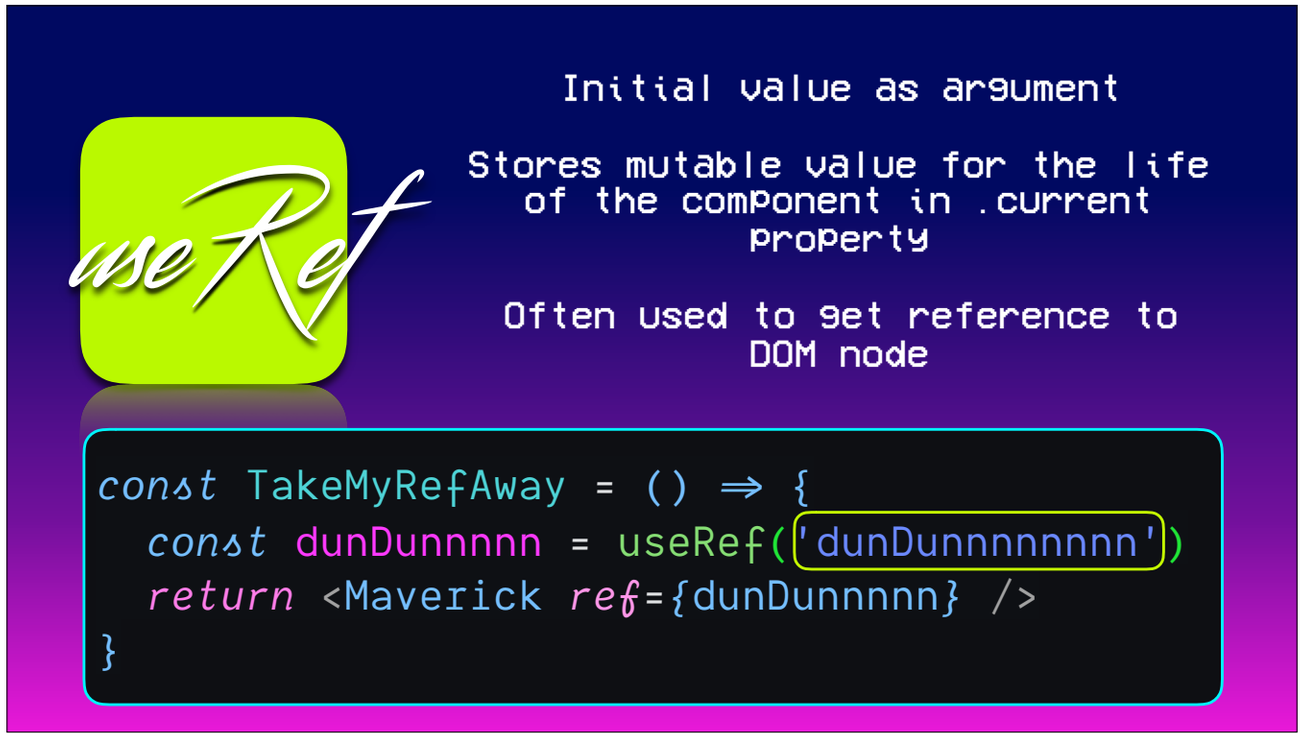
- Which value will be displayed after 5 seconds? I'll give you a second to ponder.
- Team Cheese?
- Team Carrot?
- Let's see, shall we?

```
const RealExamplesHaveCurds = () => {
  const [food, setFood] = useState('🧀')
  console.log('Render Food: ', food)
  useEffect(() => {
    setTimeout(() => setFood(food), 5000)
  }, [])
  return <>
    <p>{food}</p>
    <button onClick={() => setFood('🥕')}>🥕</button>
  </>
}
```

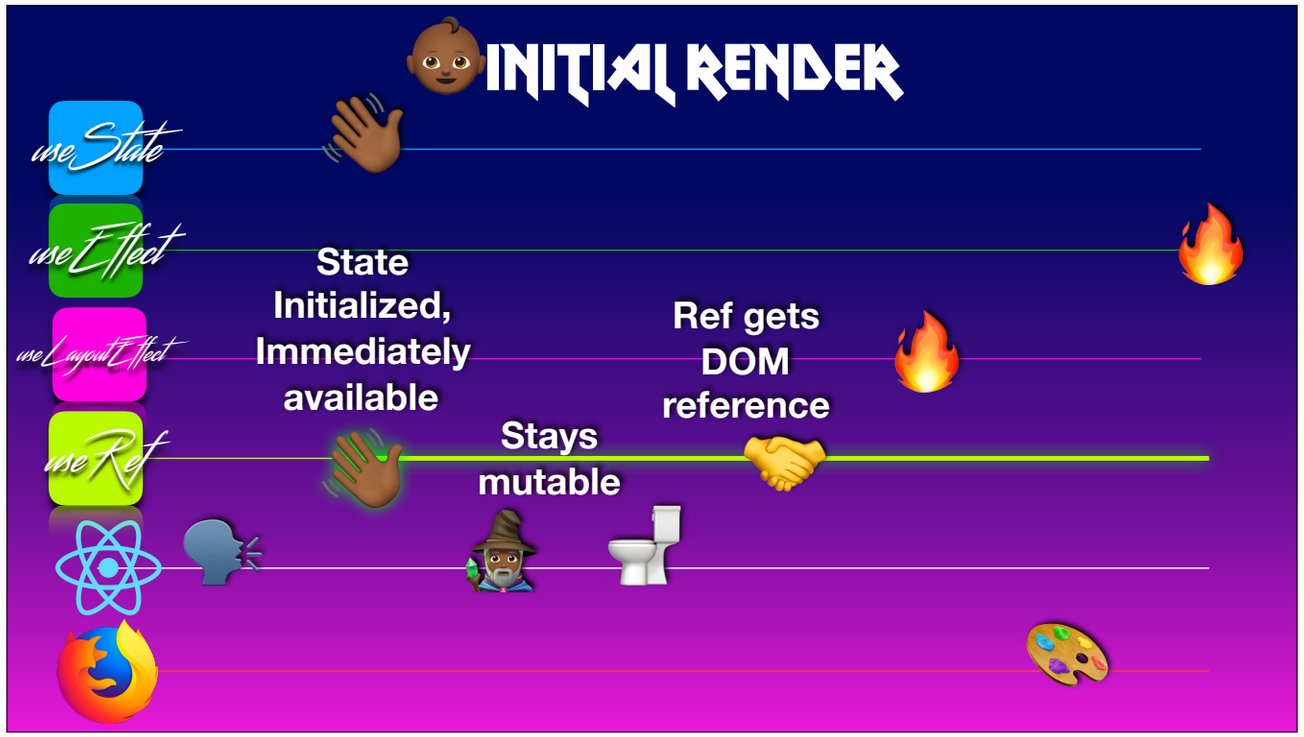
useEffect

- Since we passed an empty array to our useEffect
    - It only runs once, on the initial render.
- On the initial render, the value of **food is still cheese**
- The setTimeout and the call to setFood still belong to the first render.
- They've never even heard of carrot

- Now React is great with its functionalness and composability.
- But sometimes you just need a break from all that. 🌟
- useRef provides a way to maintain a mutable value that persists for the lifetime of the component independent of the render cycle.

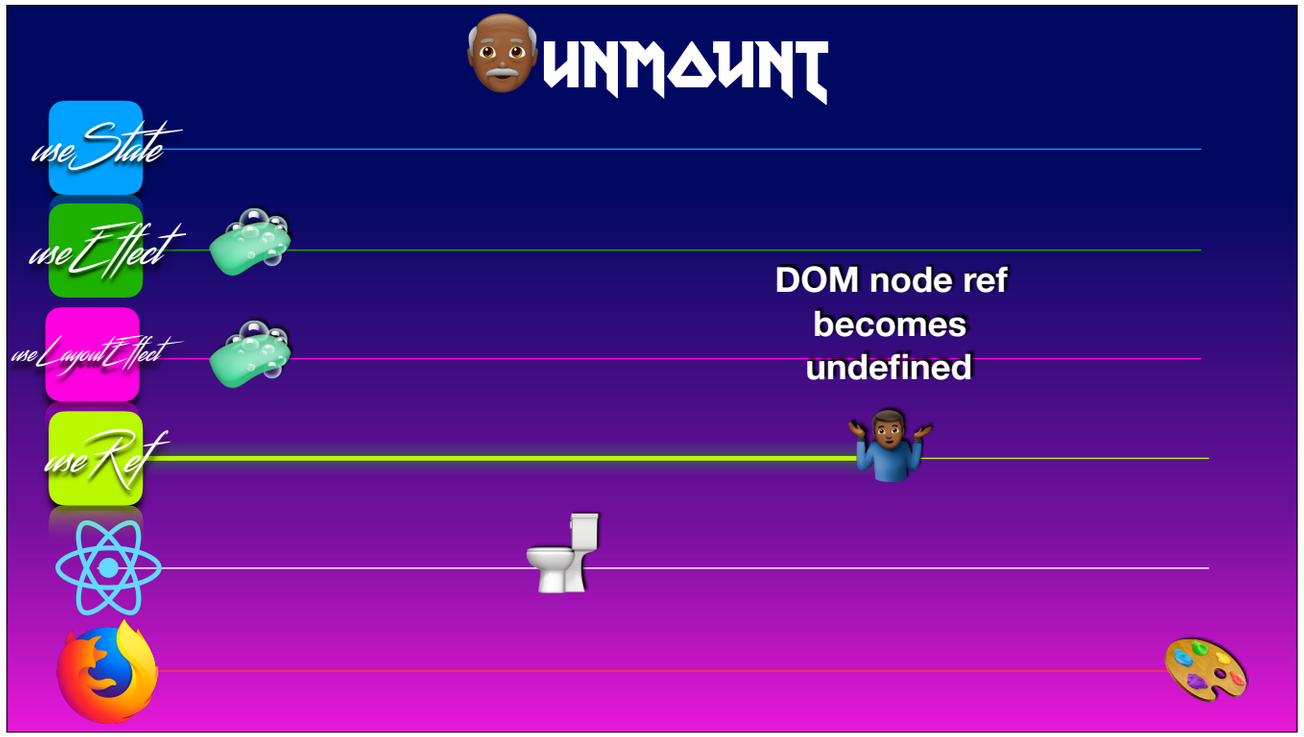


- Like `useState`, `useRef` accepts its initial value as an argument. 🌟
- The important difference is, after that moment, it lives independently of the render cycle and maintains a mutable value in its `.current` property for the entire life of the component.
- A very common use for `useRef` is to keep a reference to a DOM element around for imperative access. This is useful for, say, getting the value of an input field, but it's also a delightful escape hatch from React for when you want to make use of more imperative libraries like D3 or Three.js.



- Here's how it goes: Like useState, you'll find the value you pass as a prop available immediately🌟
    - The main difference it's in the ***.current*** property
- From then is ***mutable***
- If the ref is passed to a rendered element
    - It gets a reference to that element

- **Nothing really changes** for useRef on a re-render. It still holds whatever value it was last assigned and **remains mutable.**

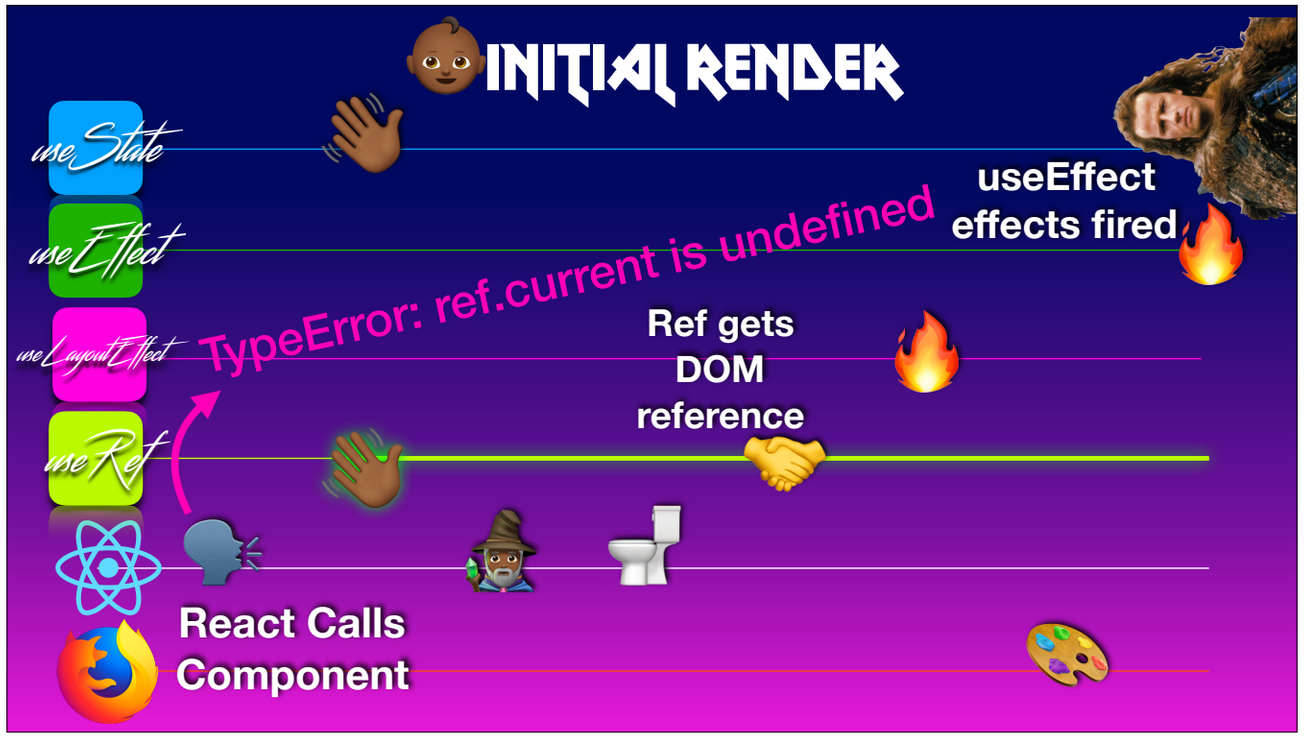


- When the component unmounts
- useRef's value keeps doing its thing until the changes are flushed to the DOM.
- When the DOM node it was holding doesn't exist anymore, naturally, the reference to it becomes **undefined.**

```
const MacLeod = ({qty}) => {
  const highlanderHolderRef = useRef()

  useEffect(() => {
    appendHighlander(highlanderHolderRef)
  })
  return <div
    id="highlanderHolder"
    ref={highlanderHolderRef}
  />
}
```

useRef

- As an example, let's take a look at the fun little bonus from our Highlander example. It works very simply. It defines a `ref`🌟, and gives it to a div🌟. Then uses `useEffect` to manually 🌟 append a highlander to our div via the `ref` on each render. Simple, right?
- But why do we have to append our highlander in a useEffect?



- These three events on our timeline are important to answer this question:
  - React calls the function and returns before the ref has a reference to its DOM element. So when the function is called, ref.current is undefined. 🌟
  - When we wrap it in a useEffect, 🌟 it waits until the very end before it acts on the DOM node.

```
// React v16.3.0 2018-03-29
<Context.Provider>
  ...
    ...
      ...
        ...
          ...
            <Context.Consumer>
              <Whatever />
            </Context.Consumer>
          ...
        ...
      ...
    ...
  ...
</Context.Provider>
```

- A little over a year ago (2018-03-29, v16.3.0), React released its officially supported Context API that allowed a sort of global state for all children of a component.
- This was done using Higher Order Components: Context.Provider around the parent, and Context.Consumer around whichever of its descendents needed to consume that context.

- Enter useContext
- Which looks like this

- useContext allows us to share state down the component tree without passing props through several layers of components that don't need them.(Known as prop drilling)
- It takes a React Context object as its argument.
    - For anyone **familiar** with the standard Context API, please note that this is the entire Context object, not Context.Consumer

```react
const WhenIGetThatFeeling = ({need}) => {
  return <div>{need}</div>
}

const WrapperPurgatory = ({value}) =>
  <div class="borderDiv">
    <WhenIGetThatFeeling need={value}/>
  </div>

const TopmostContext = () => {
  const [value, setValue] = useState('')
  return <>
    <input onChange={e => setValue(e.target.value)} value={value} />
    <WrapperPurgatory value={value} />
  </>
}
```

useContext

- To show you how it works, I've cooked up a little example.🌟 Let's start from the bottom:
- TopMostContext holds a state value,🌟 which is set by an input field 🌟and passed as a prop to WrapperPurgatory🌟
- WrapperPurgatory isn't as bad as wrapper hell but does enough to make us🌟 pass our value through a component that doesn't need it. This is to save space. Please imagine hundreds of layers of components.
- WhenIGetThatFeeling simply shows the value in a div.
- This works but it's inconvenient and it's a pattern that becomes burdensome in larger apps.

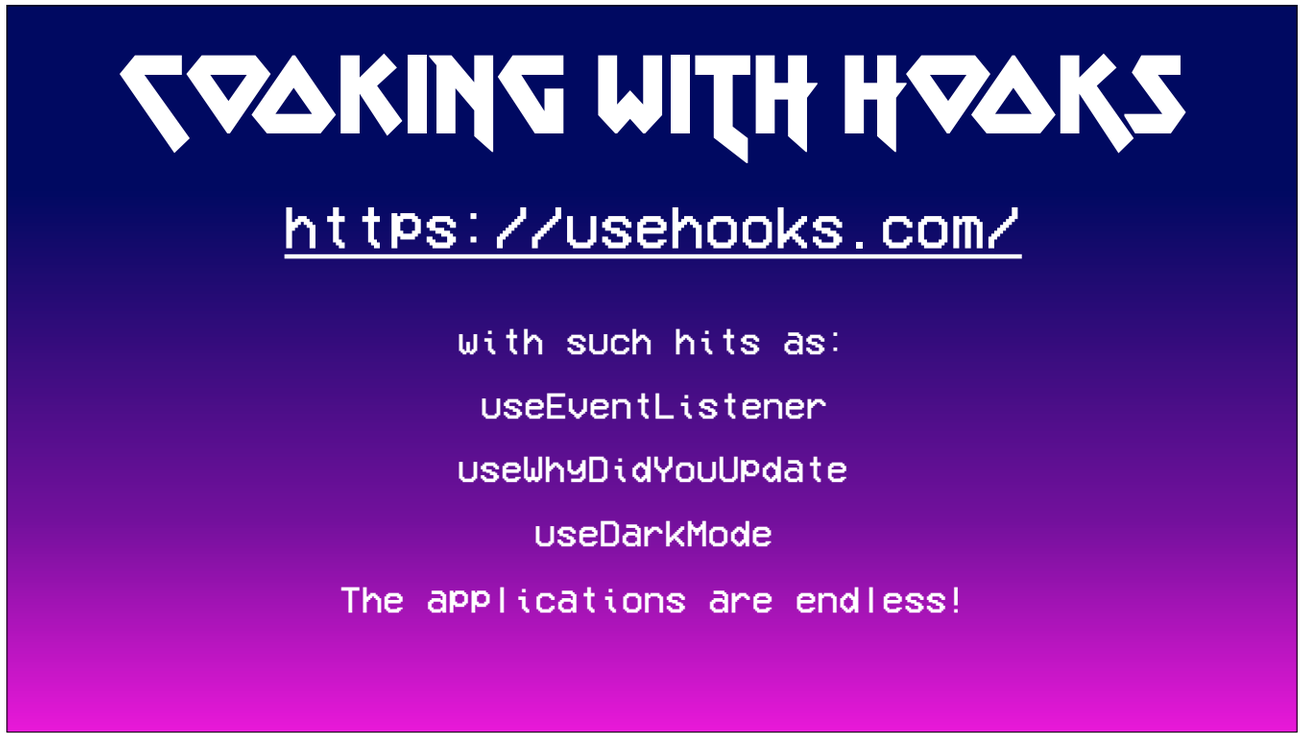
- If we use React's context API, we can skip over WrapperPurgatory entirely! 🌟
- I'm importing it just to show how it's shared between components in separate files, but MarvinContext.js only contains two lines 🌟 and we could have just as easily defined it here…
- We've made a few changes in this version. Obviously, we've eliminated prop drilling, but we've also wrapped our topmost component in the Provider for our context. 🌟 Which provides a value to the context 🌟
- In WhenIGetThatFeeling, we've defined our need value with the useContext hook. 🌟 Which gets the value from the context 🌟
- I've also added another div to WrapperPurgatory for dramatic effect. Let's check the demo just for fun DO ADD DEMO

COOKING WITH HOOKS

Hooks are composable! You can mix them and reuse them.

```
const useFetchData = (url, setData) => {
  useEffect(
    () => fetch(url).then(data => setData(data.json())), [url])
}

const Hookiee = () => {
  const [something, setSomething] = useState(null)
    useFetchData('http://somewhere.over/theRainbow', setSomething)
  return <p>{something}</p>
}
```

- One of the best things about hooks is their composability! They can be mixed together in other functions which can then be used as their own hooks and reused in multiple components!
- Here's a common example: Data fetching!
- This function takes a url string 🌟and a useState setter 🌟and combines them 🌟so the component only has to invoke one one hook instead of two!
- Neat!



- Naturally, there’s already some great resources for custom hooks:

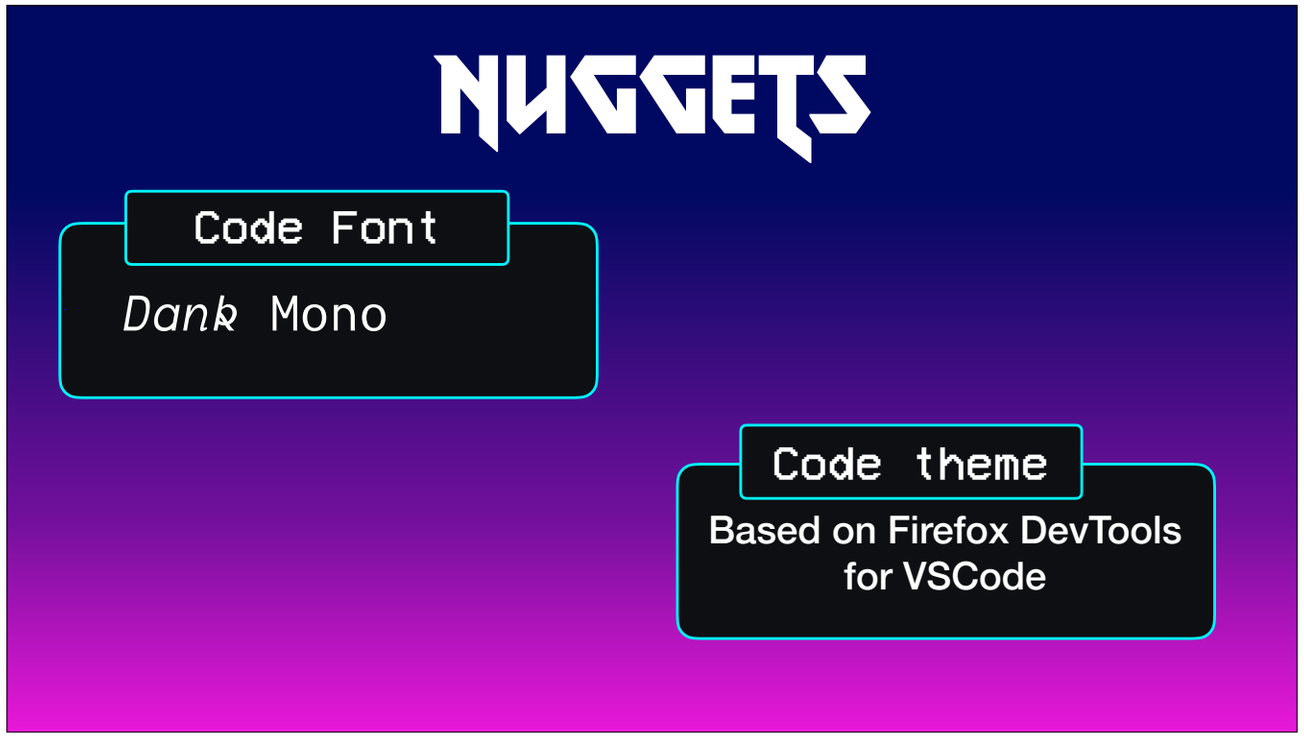
WHY HOOKS

Easy, scalable reuse of stateful logic between components

No more bulky, confusing classes

Full backward compatibility

- Only define hooks in the top level scope of your function
- NEVER IN A CONDITIONAL. The same hooks must be called every time react calls your function or it gets confused.
- There’s no urgent need to uplift old components that use classes. Classes will be supported for the foreseeable future
- useEffect dependencies should include all used values from component scope.
- Even if 2 distinct effects have the same dependencies, split them up! Each hook should have its own purpose.

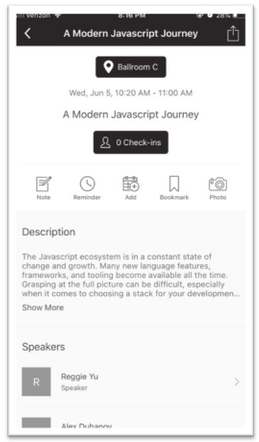


- [Font, image, and info sources]
- [pop culture references]
- [remind people to go to the GitHub repo]

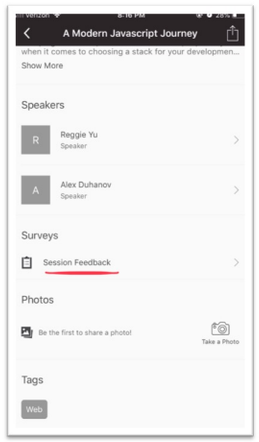
- Tech talks are a great way to spread great ideas and information, but they’re not without their problems:
    - Even the most well-researched talks are not immune to misinformation.
    - If a speaker is confident enough, people might not question anything they’ve just seen.
    - If you get lost or miss something, you might have to wait for the videos to be posted to go back and see what it was.
- For these reasons, I have made this entire talk completely open source. All slides, examples, and a version of this talk in MarkDown are available on GitHub.
- If you notice anything that doesn’t seem right during this talk, I ask that you please log an issue or open a PR on this repo. If you didn’t notice anything wrong, I encourage you to check it out in a few days to see if I’ve lied to you today.

PLEASE LEAVE FEEDBACK!

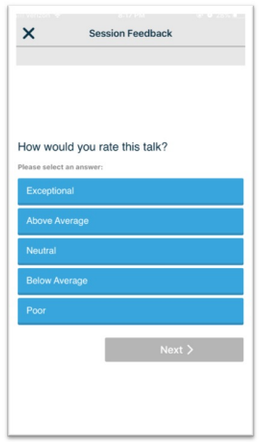
Search for this session.



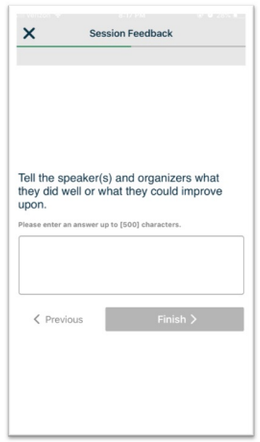
Select Session Feedback



Select a rating.



Leave a comment.



DON'T YOU

FORGET ABOUT ME

Don't, don't, don't, don't....