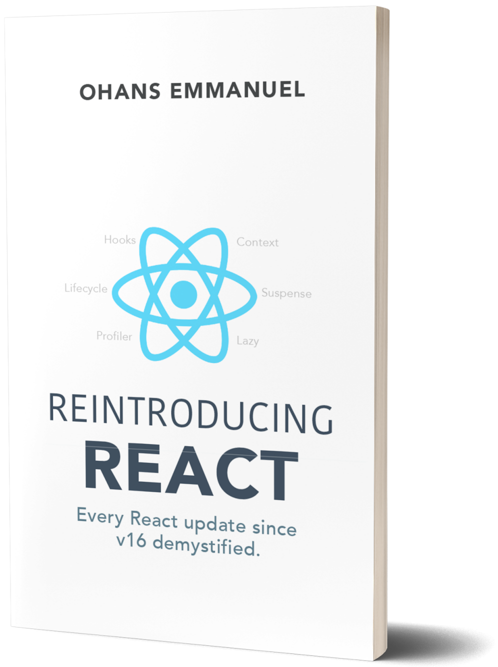


OHANS EMMANUEL

REINTRODUCING REACT

Every React update since v16 demystified.

Reintroducing React: Every React Update Since v16 Demystified.

In this book, unlike any you have come across before, I will deliver funny, unfeigned and dead serious comic strips about every React update since v16+. It'll be hilarious, either intentionally or unintentionally, easy on beginners as well as professionals, and will be very informative as a whole.

Why Comic Strips ?

I have been writing software for over 5 years. But I have done other things, too. I've been a graphics designer, a published author, teacher, and a long, long time ago, an amateur Illustrator.

I love the tech community, but sometimes as a group, we tend to be a little narrow-minded.

When people attempt to teach new technical concepts, they forget who they were before they became developers and just churn out a lot of technical jargon - like other developers they've seen.

When you get to know people, it turns out so many of us have different diverse backgrounds! "If you were a comedian, why not explain technical concepts with some comedy?

Wouldn't that would be so cool?

I want to show how we can become better as engineers, as teams, and as a community, by openly being our full, weird selves, and **teaching others with all**

that personality. But instead of just talking, I want to make it noteworthy and lead by example. So, you're welcome to my rendition of a comic strip inspired book about every React update since v16.

With recently released v16.8 features, there's going to be a lot of informative comic strips to be delivered!

Inspired by [Jani Eväkallio](#).

why do we need a reintroduction?
Reintroducing React:
A Comic Strip on Every React Update Since v16

Why Reintroduce React ?

I wrote my first React application 3 years ago. Between then and now, the fundamental principles of React have remained the same. React is just as declarative and component-based today as it was then.

That's great news, however, the way we write React applications today has changed!

There's been a lot of new additions (and well, removals).

If you learned React a while back it's not impossible that you haven't been up to date with every new feature/release. It's also possible to get lost on all the new features. Where exactly do you start? How important are they for your day to day use?

Even as an experienced engineer, I sometimes find unlearning old concepts and relearning new ones just as intimidating as learning a new concept from the scratch.

If that's the case with you, I hope I can provide the right help for you via this guide.

The same applies if you're just learning React.

There's a lot of state information out there. If you learn React with some old resource, yes, you'll learn the fundamentals of React, but modern React has new interesting features that are worth your attention. It's best to know those now, and not have to unlearn and relearn newer concepts.

Whether you've been writing React for a while, or new to developing React applications, I will be discussing **every** update to React since version 16.

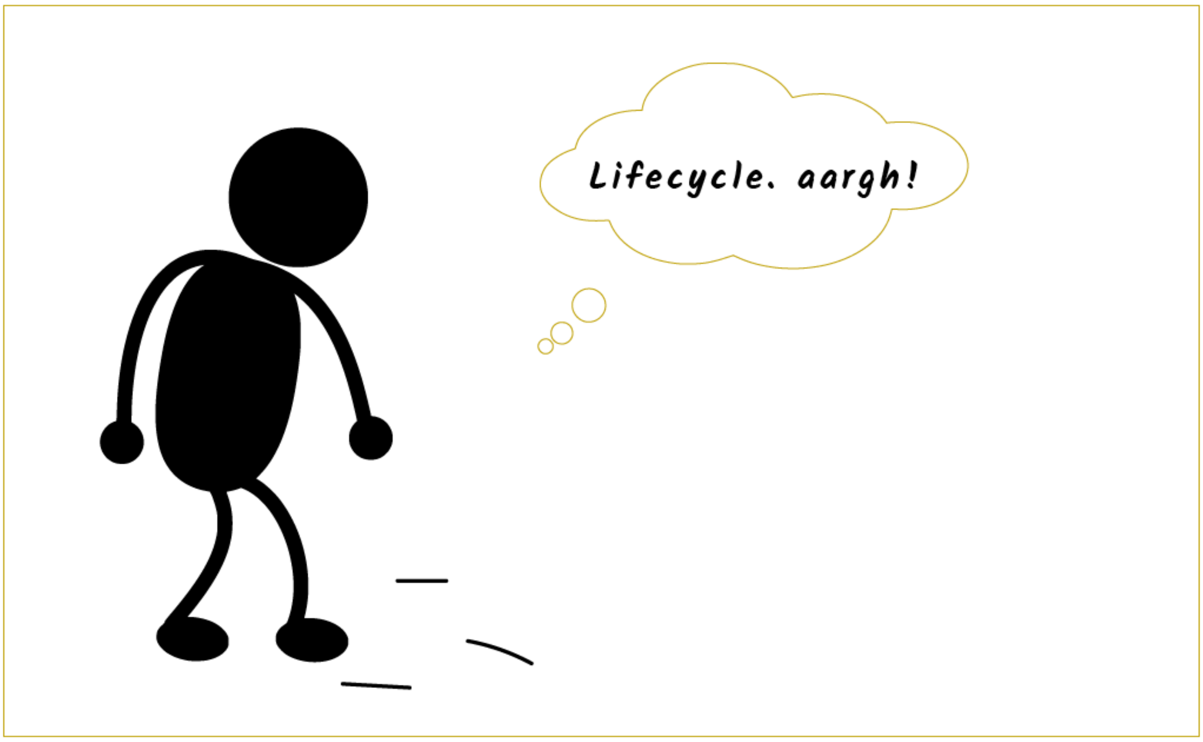
This will keep you in sync with the recent changes to React, and help you write better software.

Remember, a reintroduction to React is important for not just beginners, but professionals alike. It all depends on how well you've kept your ear to the ground, and really studied the many changes that have been released over the last 12 months.

Chapter 1: New Lifecycle Methods.

He's been writing software for a while, but new to the React ecosystem.

Meet John.

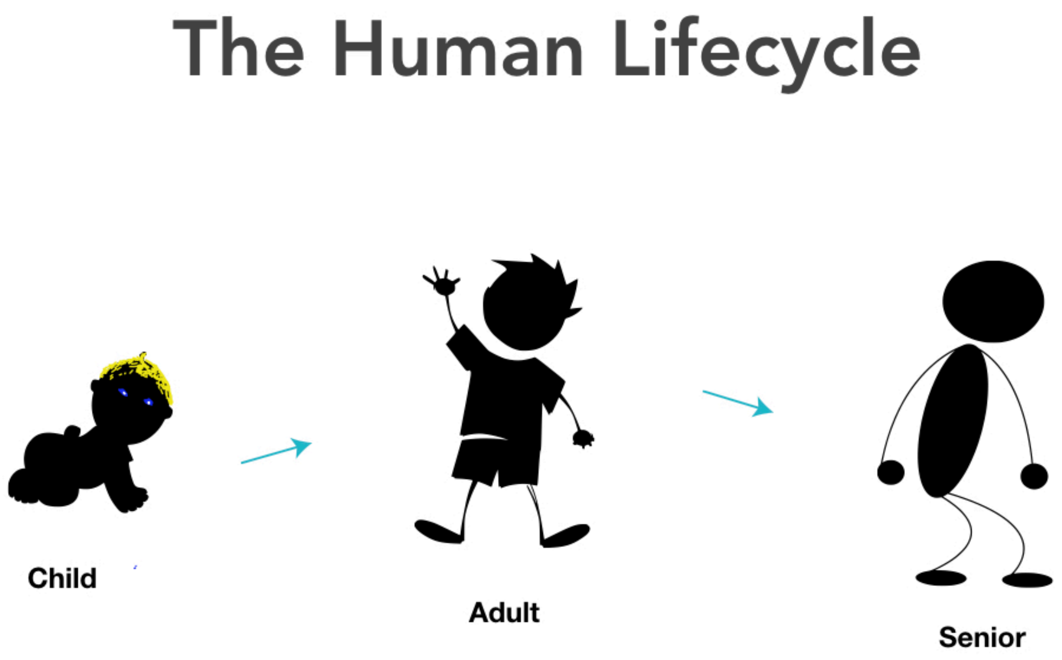


For a long time he didn't fully understand what lifecycle truly meant in the context of React apps.

When you think of lifecycle what comes to mind?

What's Lifecycle Anyway.

Well, consider humans.



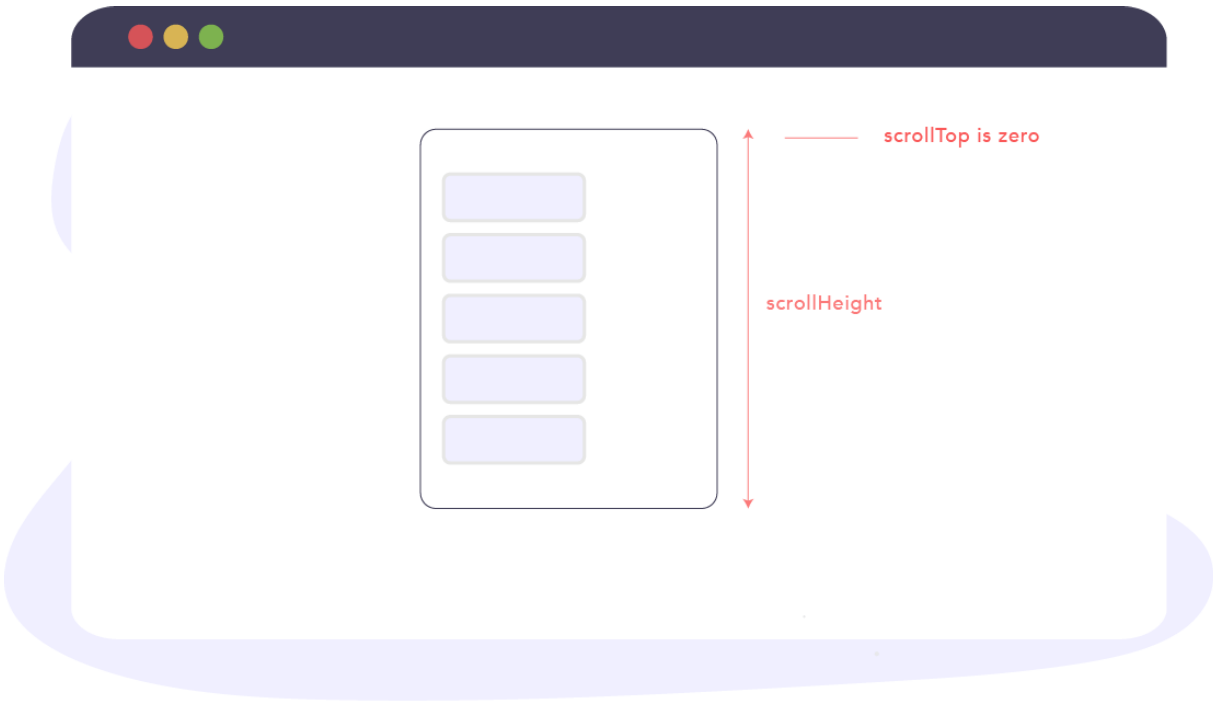
The typical lifecycle for a human is something like, "child" to "adult" to "elderly".

In the biological sense, lifecycle refers to the series of "changes in form" an organism undergoes.

The same applies to React components. They undergo a series of "changes in form".

Here's what a simple graphical representation for React components would be.

First, consider a situation where the entire height of all chat messages doesn't exceed the height of the chat pane.



Here, the expression `chatThreadRef.scrollHeight - chatThreadRef.scrollTop` will be equivalent to `chatThreadRef.scrollHeight - 0`.

When this is evaluated, it'll be equal to the `scrollHeight` of the chat pane – just before the new message is inserted to the DOM.

If you remember from the previous explanation, the value returned from the `getSnapshotBeforeUpdate` method is passed as the third argument to the `componentDidUpdate method`. We call this snapshot:

```
componentDidUpdate(prevProps, prevState, snapshot) {
}
```

The value passed in here – at this time, is the previous `scrollHeight` before the update to the DOM.

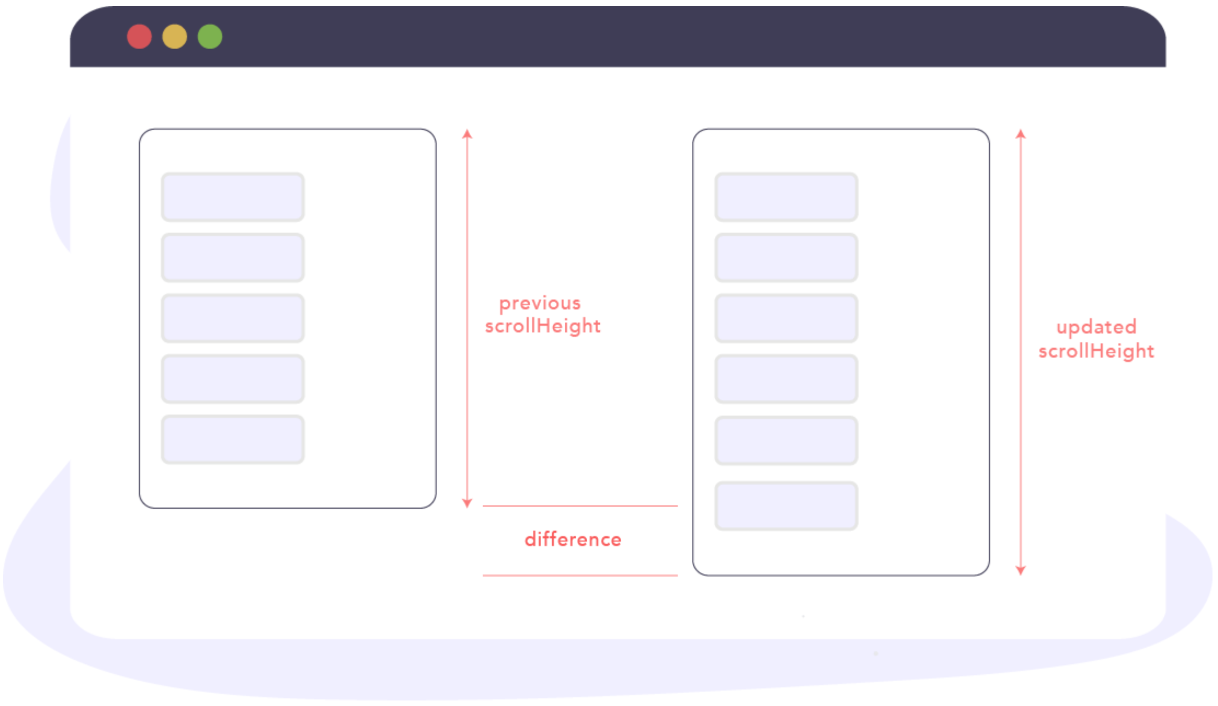
In the `componentDidUpdate` we have the following code, but what does it do?

```
componentDidUpdate(prevProps, prevState, snapshot) {
    if (snapshot !== null) {
      const chatThreadRef = this.chatThreadRef.current;
      chatThreadRef.scrollTop = chatThreadRef.scrollHeight - snapshot;
    }
  }
```

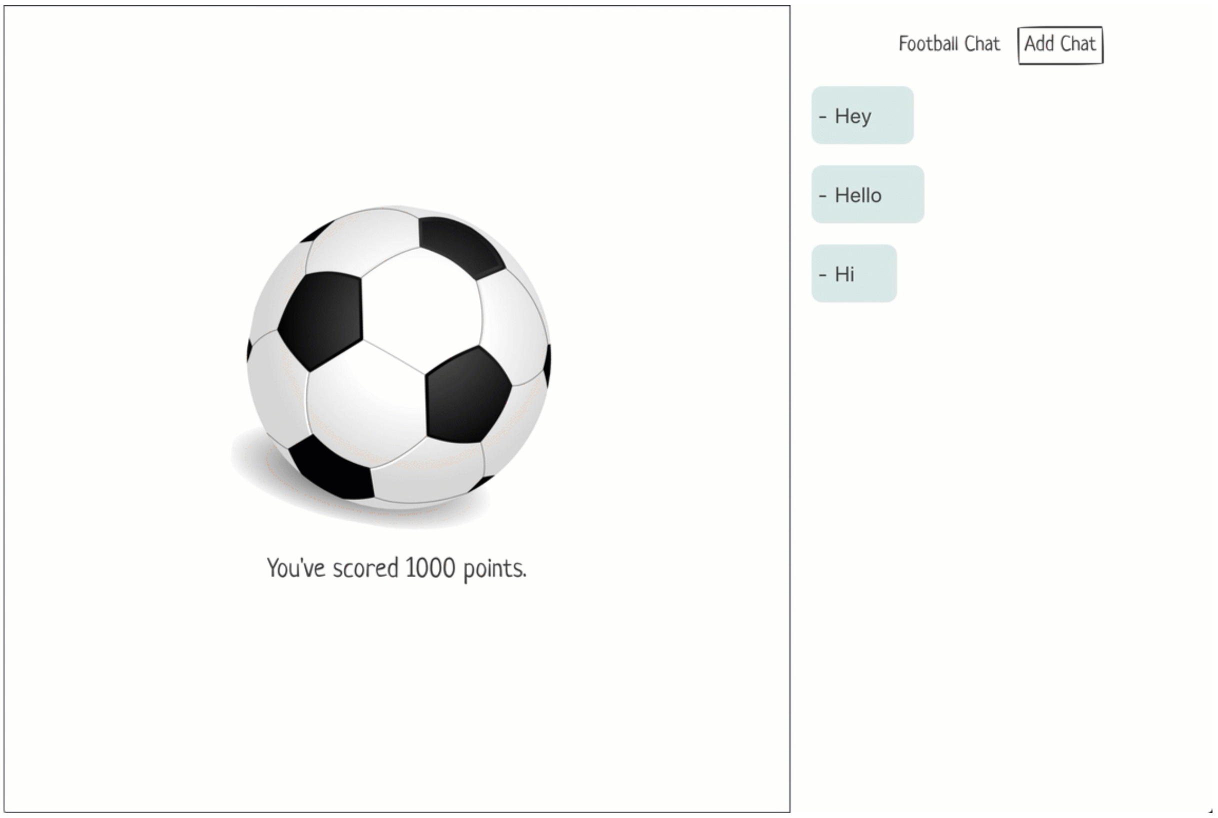
In actuality, we are programmatically scrolling the pane vertically from the top down, by a distance equal to `chatThreadRef.scrollHeight - snapshot;`

Since `snapshot` refers to the `scrollHeight` before the update, the above expression returns the `height` of the new chat message plus any other related height owing to the update.

Please see the graphic below:



When the entire chat pane height is occupied with messages (and already scrolled up a bit), the `snapshot` value returned by the `getSnapshotBeforeUpdate` method will be equal to the actual height of the chat pane.



Conclusion.

It is worth mentioning that while new additions were made to the component lifecycle methods, some other methods such as `componentWillMount` , `componentWillUpdate`, `componentWillReceiveProps` were deprecated.

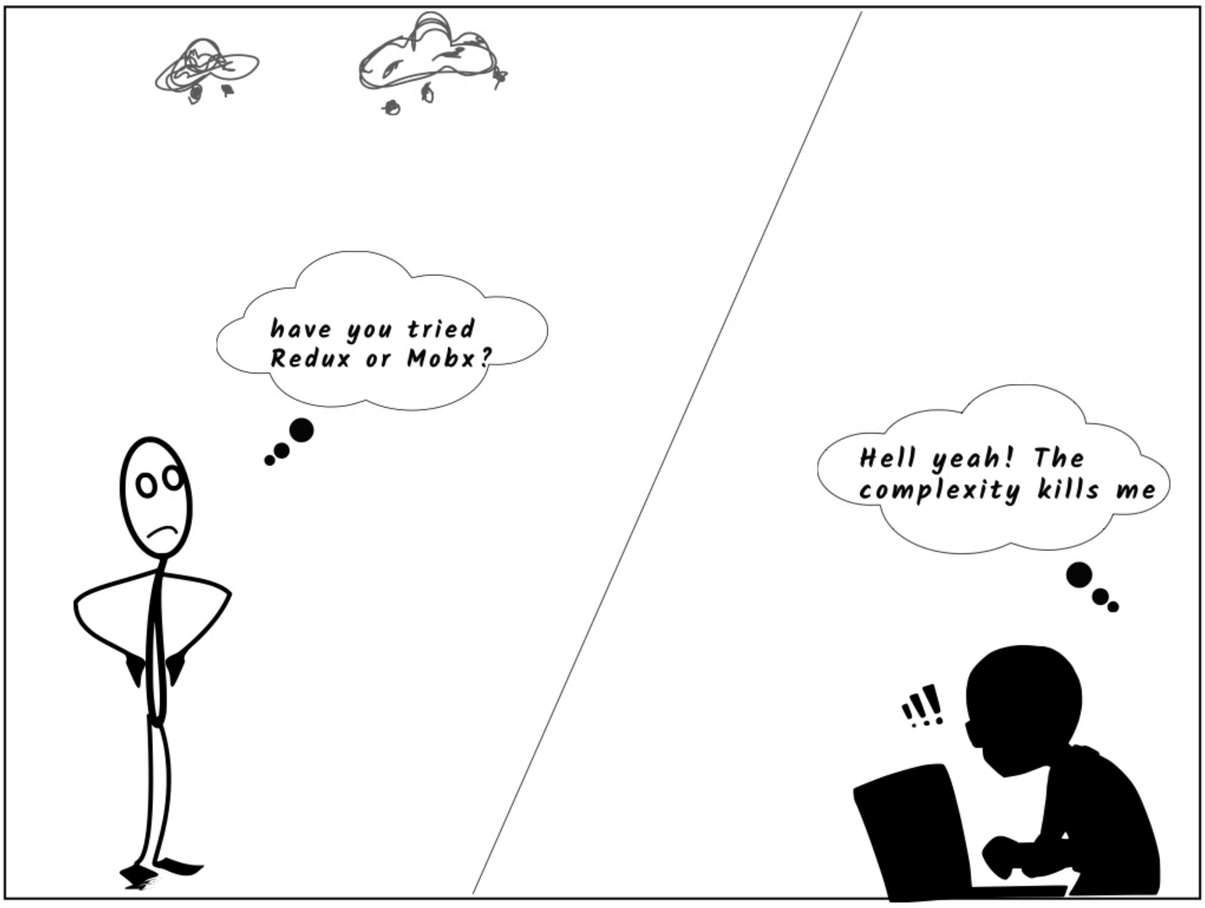
Chapter 2: Simpler State Management with the Context API.

John's an amazing developer, and he loves what he does. However, he's frequently been facing the same problem when writing React applications.

Props drilling, the term used to describe passing down props needlessly through a deeply nested component tree, has plagued John for a while now!

Luckily, John has a friend who always has all the answers. Hopefully, she can suggest a way out.

John reaches out to Mia, and she steps in to offer some advice.


have you tried Redux or Mobx?
Hell yeah! The complexity kills me
!!!

Mia is a fabulous engineer as well, and she suggests using some state management library such as `Redux` or `MobX`.

These are great choices, however, for most of John's use cases, he finds them a little too bloated for his need.

"*Can't I have something simpler and native to React itself*", says John.

Conclusion

This illustrates that you can pass not only state values, but also their corresponding updater functions in a context Provider. These will be available to be consumed anywhere in your component tree.

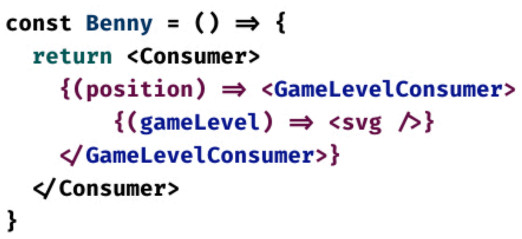
Having made the bank app work well with the Context API, I'm pretty sure John will be proud of the progress we've made!

Chapter 3: ContextType – Using Context without a Consumer.

So far, John has had a great experience with the Context. Thanks for Mia who recommended such great tool.

However, there's a little problem.

As John uses the context API more often, he begins to realise a problem.



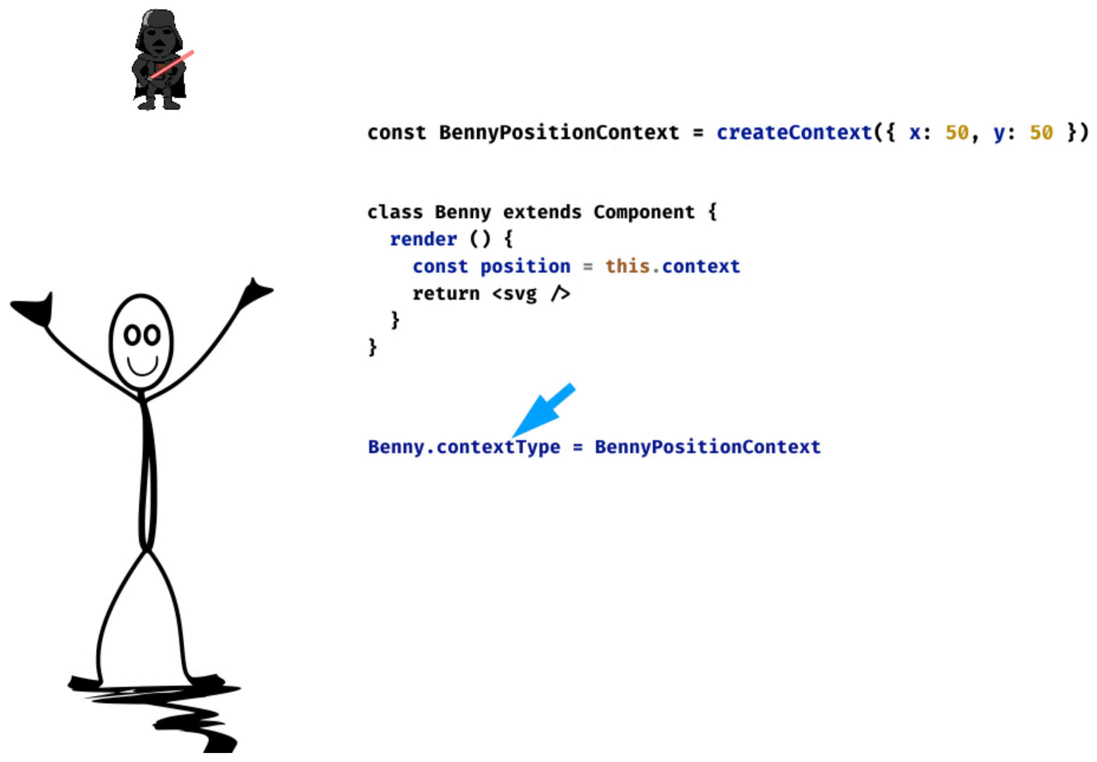
```
const Benny = () => {
  return <Consumer>
    {(position) => <GameLevelConsumer>
        {(gameLevel) => <svg />}
    </GameLevelConsumer>}
  </Consumer>
}
```

```
  }
}
```

In this example, `Benny` consumes the initial context values `{ x: 50, y: 50 }` from the context object.

However, using a `Consumer` forces you to use a render prop API that may lead to nested code.

Let's get rid of the `Consumer` component by using the `contextType` class property.



```
const BennyPositionContext = createContext({ x: 50, y: 50 })

class Benny extends Component {
  render () {
    const position = this.context
    return <svg />
  }
}

Benny.contextType = BennyPositionContext
```

Getting this to work is fairly easy.

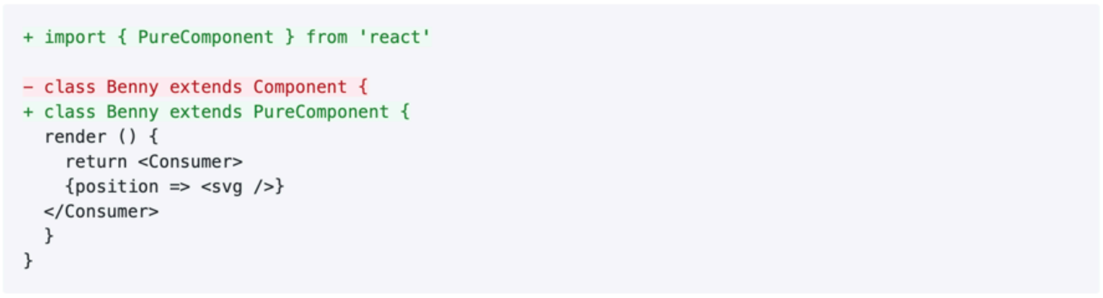
First, you set the `contextType` property on the class component to a context object.

```
const BennyPositionContext = createContext({ x: 50, y: 50 })
// Class Benny extends Component ...
```

Chapter 4: React.memo === Functional PureComponent.

A few weeks ago John refactored the `Benny` component to a `PureComponent`.

Here's what his change looked like:

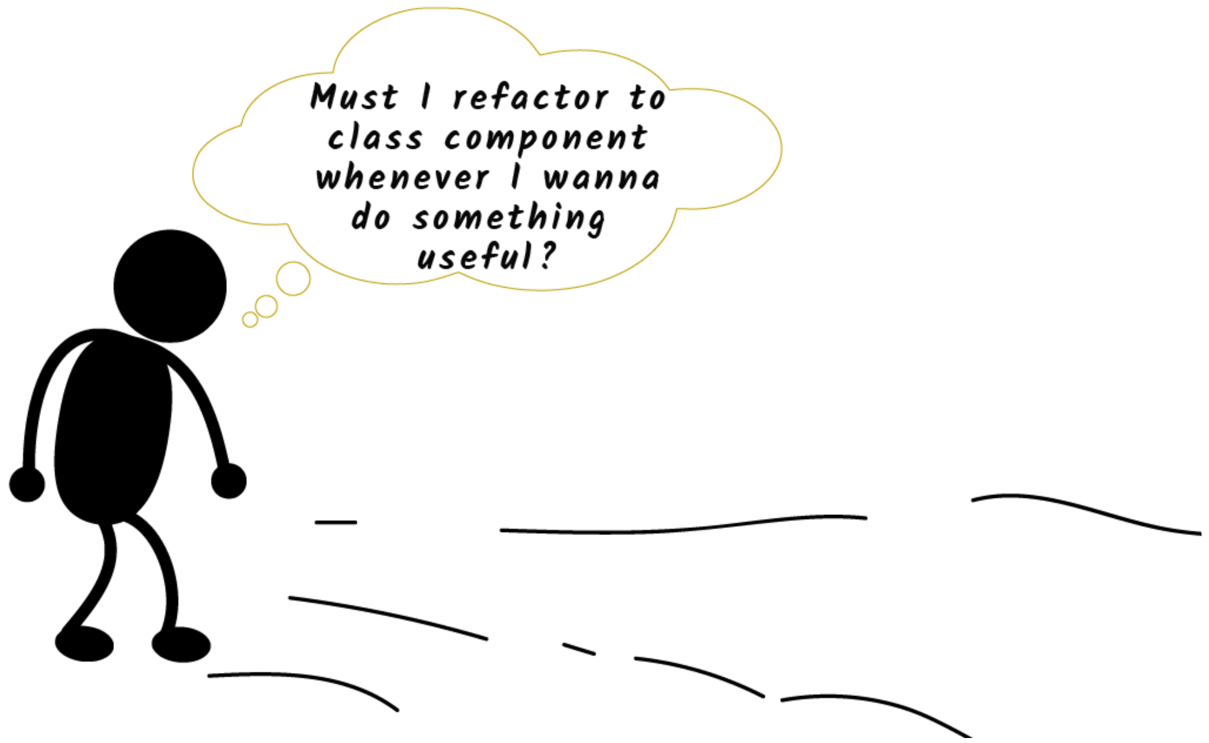


```
+ import { PureComponent } from 'react'

- class Benny extends Component {
+ class Benny extends PureComponent {
  render () {
    return <Consumer>
    {position => <svg />}
  </Consumer>
  }
}
```

Well, that looks good.

However, the only reason he refactored the `Benny` component to a class component was because be needed to extend `React.PureComponent`.


Must I refactor to class component whenever I wanna do something useful?

The solution works, but what if we could achieve the same effect without having to refactor from functional to class components?

Refactoring large components just because you need a specific React feature isn't the most pleasant of experiences.

How React.memo works.

`React.memo` is the perfect replacement for the class' `PureComponent`. All you do is wrap your functional component in the `React.memo` function call and you get the exact behaviour `PureComponent` gives.

Here's a quick example:

```
// before

import React from 'react'

function MyComponent ({name}) {

    return ( <div>
```

Chapter 5: The Profiler – Identifying Performance Bottlenecks.

It's Friday and Mia is headed back home. On her way home she can't help but think to herself.

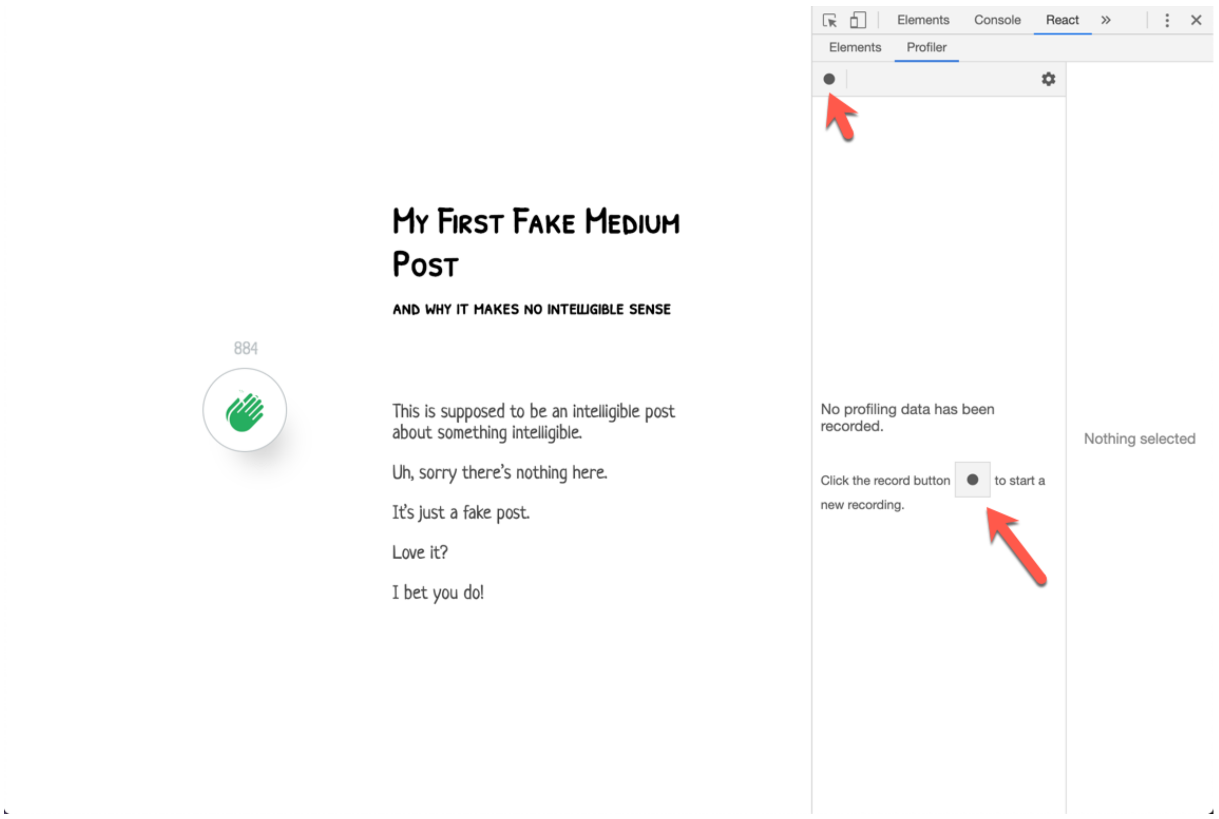
"What have I achieved today?" Mia says her to herself.

After a long careful thought, she comes to the conclusion that she accomplished just one thing the entire day.

How does the Profiler Work?

The Profiler works by recording a session of actual usage of your application. In this recording session it gathers information about the components in your application and displays some interesting information you can exploit to find performance bottlenecks.

To get started, click the record button.



My First Fake Medium Post

AND WHY IT MAKES NO INTELLIGIBLE SENSE

This is supposed to be an intelligible post about something intelligible.

Uh, sorry there's nothing here.

It's just a fake post.

Love it?

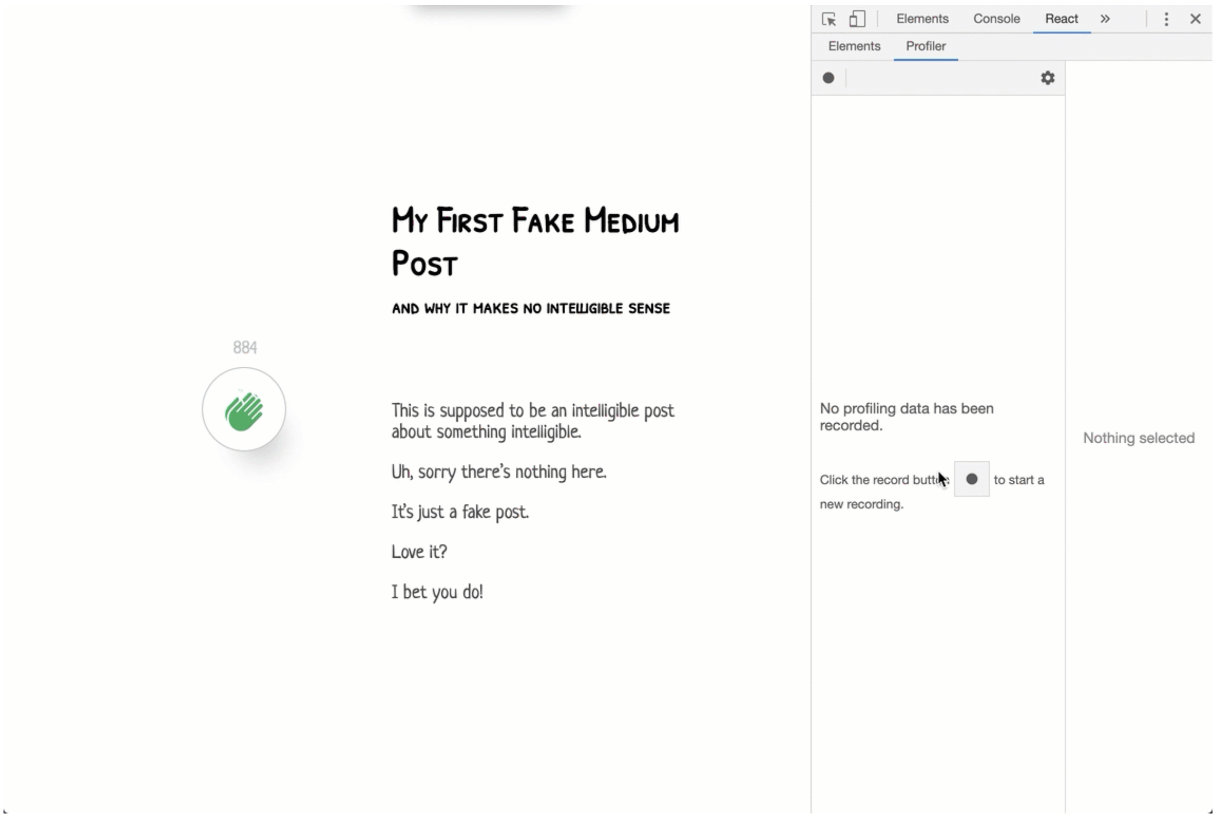
I bet you do!

No profiling data has been recorded.

Click the record button ● to start a new recording.

After clicking 'record', you then go ahead to use your application as you'd expect a user to.

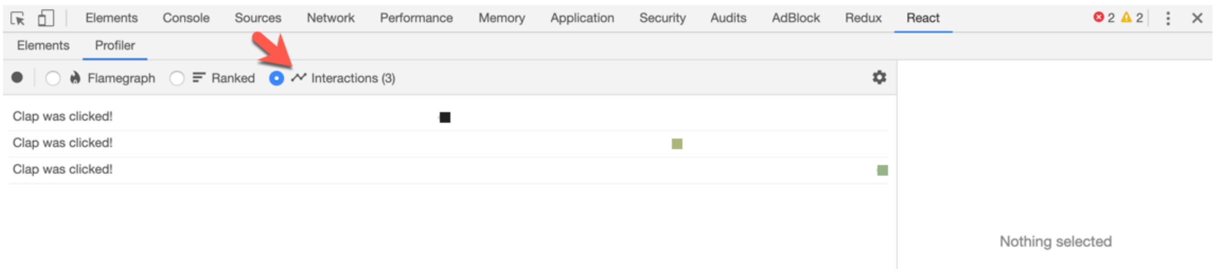
In this case, I've gone ahead to click the medium clap button 3 times!


884
My First Fake Medium Post
AND WHY IT MAKES NO INTELLIGIBLE SENSE
This is supposed to be an intelligible post about something intelligible.
Uh, sorry there's nothing here.
It's just a fake post.
Love it?
I bet you do!
Elements
Console
React
Elements
Profiler
No profiling data has been recorded.
Click the record button ● to start a new recording.
Nothing selected

Once you're done interacting with your application, hit stop to view the information the Profiler provides.

Making Sense of the Profiler Results.

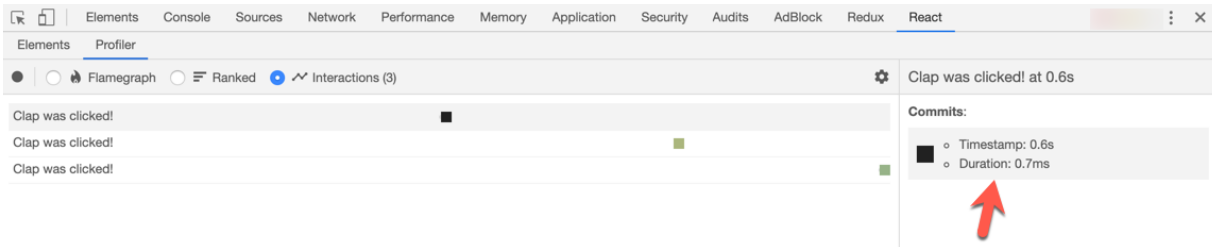
On the far right of the profiler screen, you'll find a visual representation of the number of commits made during your interaction with your application.



Clap was clicked!
Clap was clicked!
Clap was clicked!

Nothing selected

Clicking three times now record 3 interactions and you can click on any of the interactions to view more details about them.



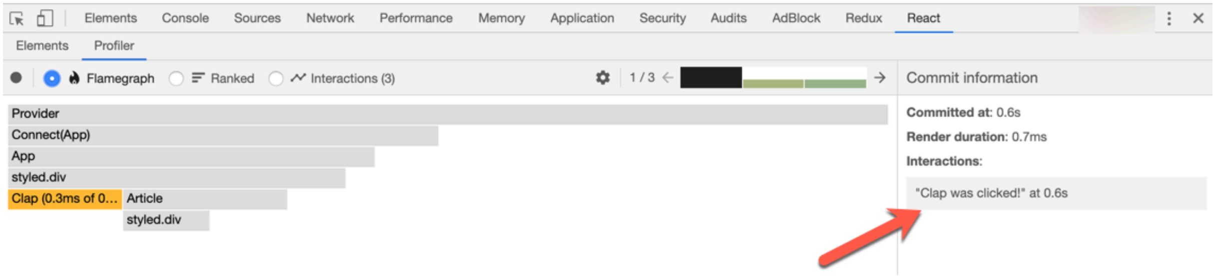
Clap was clicked!
Clap was clicked!
Clap was clicked!

Clap was clicked! at 0.6s

Commits:

- Timestamp: 0.6s
- Duration: 0.7ms

The interactions will also show up in the flame and ranked charts.



Commit information

Committed at: 0.6s

Render duration: 0.7ms

Interactions:

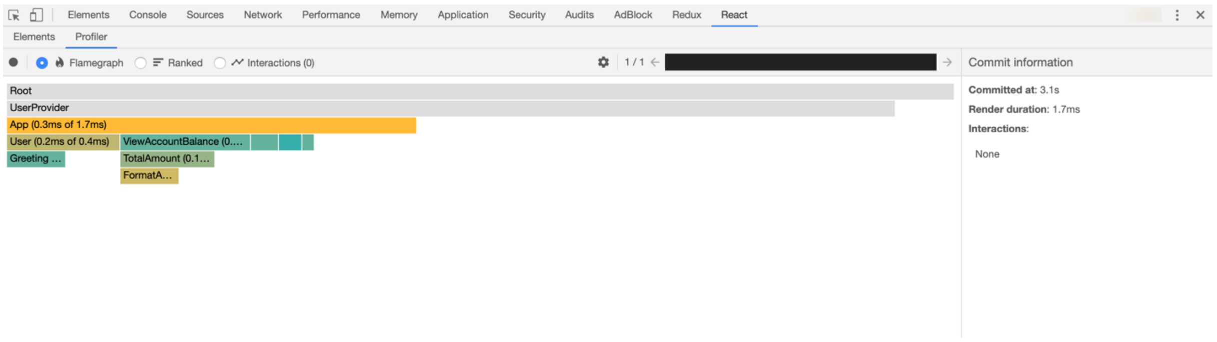
"Clap was clicked!" at 0.6s

Example: Identifying Performance BottleNecks in the Bank Application.

"Hey John, what have you done?!!!", said Mia as she stumped into John's office.

"I was just profiling the bank application, and it's so not performant", she added.

John wasn't surprised. He had not spent a lot of time thinking about performance, but with Mia up in his face, he began to have a rethink.



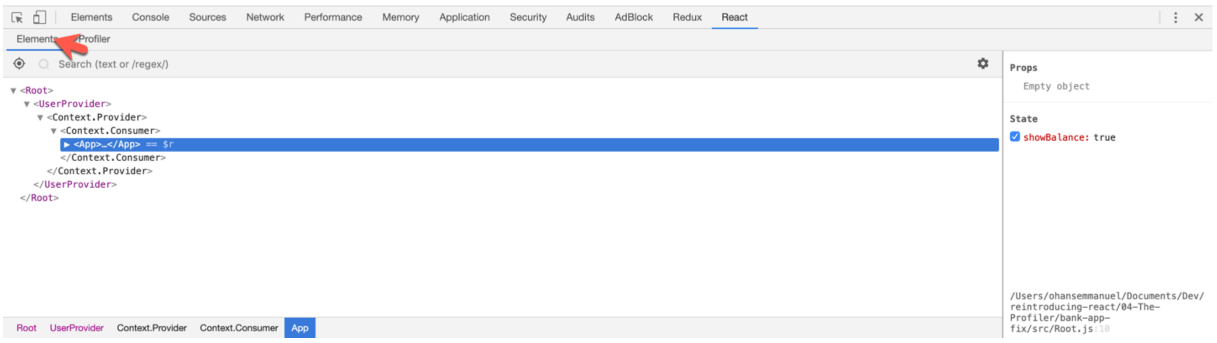
```
Root
UserProvider
App (0.3ms of 1.7ms)
User (0.2ms of 0.4ms)    ViewAccountBalance (0....
Greeting ...             TotalAmount (0.1...
                         FormatA...
```

Commit information

Committed at: 3.1s
Render duration: 1.7ms
Interactions:
None

From the flame graph above, every child component of `App` as been re-rendered. They all had nothing to do with this visual update, so those are wasted rendered.

NB: If you need to check the hierarchy of components more clearly, remember you can always click the elements tab:



```
▼ <Root>
  ▼ <UserProvider>
    ▼ <Context.Provider>
      ▼ <Context.Consumer>
        ▶ <App>…</App> == $r
        </Context.Consumer>
      </Context.Provider>
    </UserProvider>
  </Root>
```

Props
Empty object

State
showBalance: true

/Users/ohansemmanuel/Documents/Dev/reintroducing-react/04-The-Profiler/bank-app-fix/src/Root.js:10

Well, since these child components are functional components, let's use `React.memo` to memoize the render results so they don't change except there's a change in props.

```
// User.js
import { memo } from 'react'
const User = memo(({ profilePic }) => {
  ...
})
// ViewAccountBalance.js
import { memo } from 'react'
const ViewAccountBalance = memo(({ showBalance, displayBalance }) => {
      ...
})
```

Conclusion.

Profiling applications and identifying performance leaks is fun and rewarding. I hope you've gained relevant knowledge in this section.

Chapter 6: Lazy Loading with React.Lazy and Suspense.

“Hey John, we need to look into lazy loading some modules in the Benny application”, says Tunde, John’s manager.

John’s had great feedback from his manager for the past few months. Every now and then Tunde storms into the office with a new project idea. Today, it’s lazy loading with `React.Lazy` and `Suspense`.

John’s never lazy loaded a module with React.Lazy and Suspense before now. This is all new to him, so he ventures into a quick study to deliver on what his manager has requested.

What is Lazy Loading?

When you bundle your application, you likely have the entire application bundled in one large chunk.

As your app grows, so does the bundle.

```
const AwesomeA = React.lazy(() => import('AwesomeA'))
```

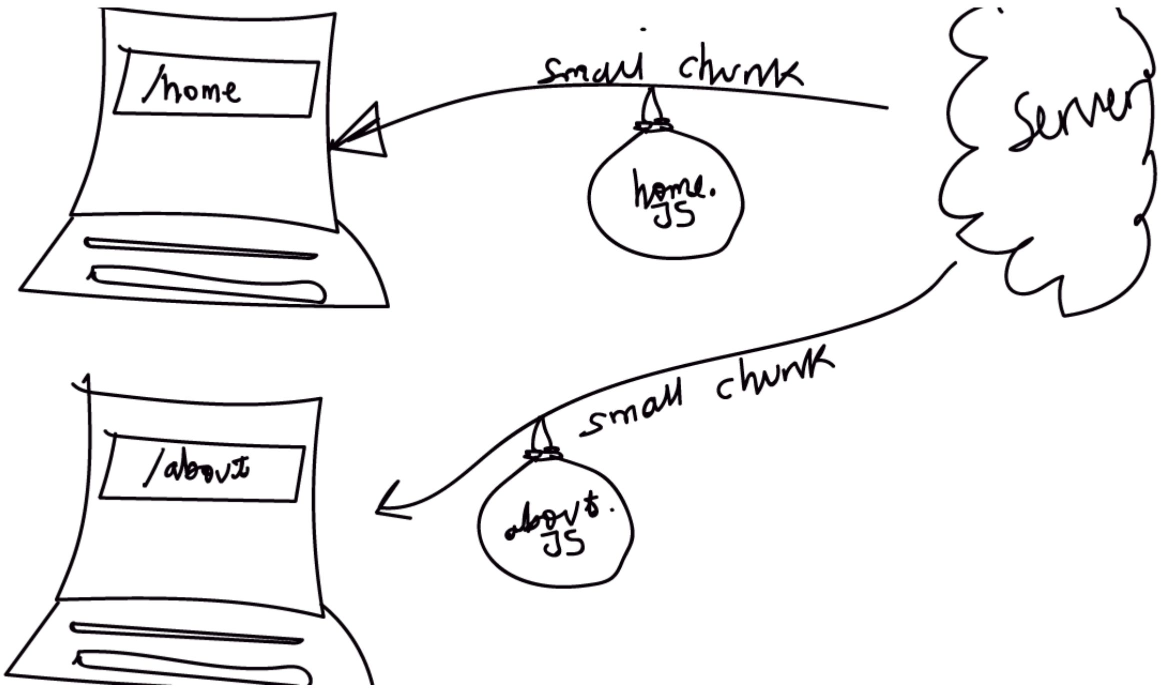
Problem solved!

Code splitting routes.

Code splitting advocates that instead of sending this large chunk of code to the user at once, you may dynamically send chunks to the user when they need it.

We had looked at component based code splitting in the earlier examples, but another common approach is with route based code splitting.

In this method, the code is split into chunks based on the routes in the application.



We could also take our knowledge of lazy loading one step further by code splitting routes.

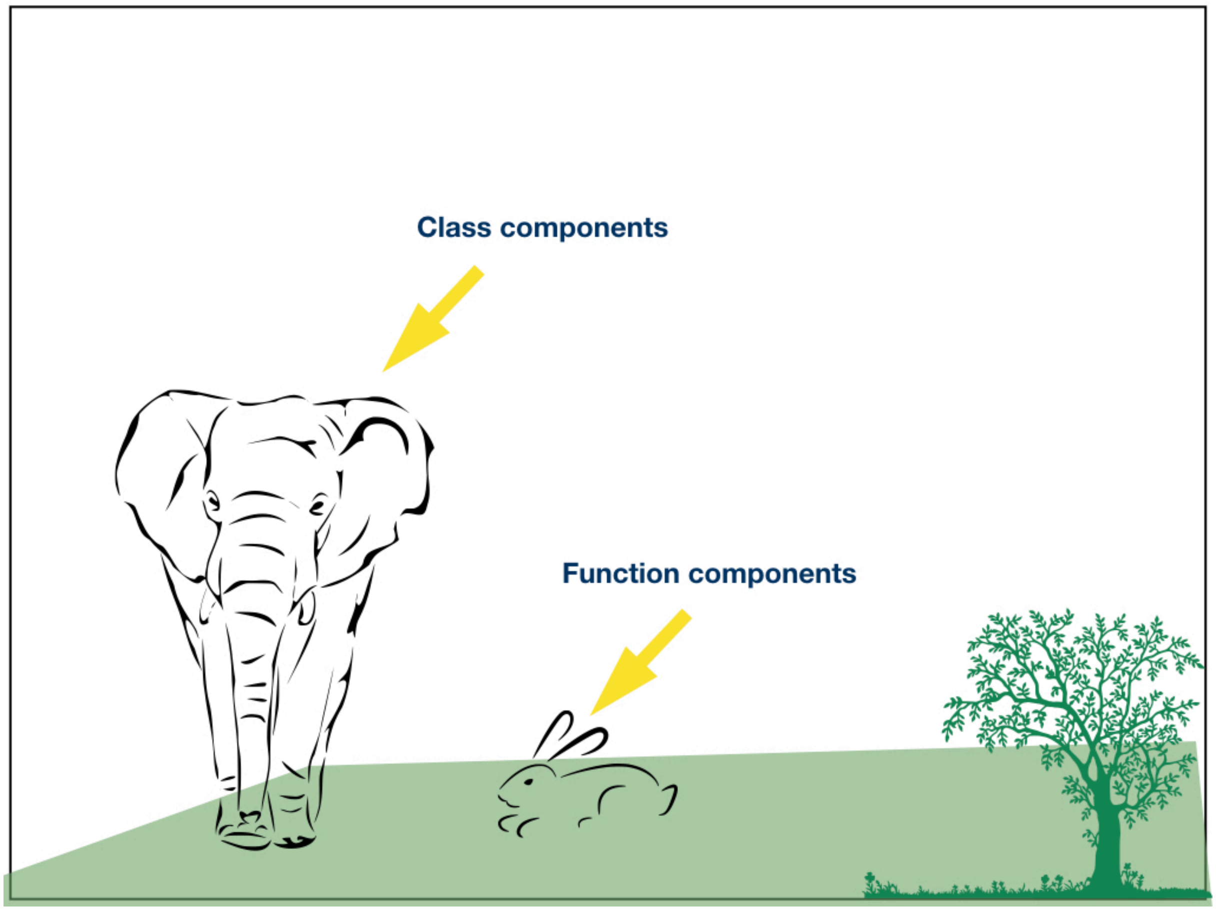
Consider a typical React app that uses `react-router` for route matching.

```
const App = () => (
  <Router>
```

It’s likely this will change in the future, but in the mean time, if you care about SSR, using react-loadable is your best bet for lazy loading `React` components.

Chapter 7: Hooks — Building Simpler React Apps.

For the past 3 years John's been writing React apps, functional components have mostly been dumb.



If you wanted local state or some other complex side effects, you had to reach out to class component. You either painfully refactor your functional components to class components or nothing else.

It's a bright Thursday afternoon, and while having lunch, Mia introduces John to Hooks.

Hey! You must have heard about hooks? You don't live under a rock, huh?

She speaks so enthusiastically, it piques John's interest.

Of all the things Mia said, one thing struck John. "*With hooks, functional components become just as powerful (if not more powerful) than your typical class components*", said Mia.

Chapter 8: Advanced React Patterns with Hooks

With the release of hooks, certain React patterns have gone out of favour. They can still used, but for most use cases you're likely better off using hooks. For example, choose hooks over render props or higher order components.

There may be specific use cases where these could still be used, but most of the times, choose hooks.

That being said, we will now consider some more advanced React patterns implemented with hooks.

Ready?

Introduction

This chapter may be the longest in the book, and for good reason. Hooks are likely the way we'll be writing React components in the next couple of years, and so they are quite important.

In this chapter, we'll consider the following advanced React patterns:

- Compound Components
- Props Collection
- Prop Getters
- State Initializers
- State Reducer

- Control Props

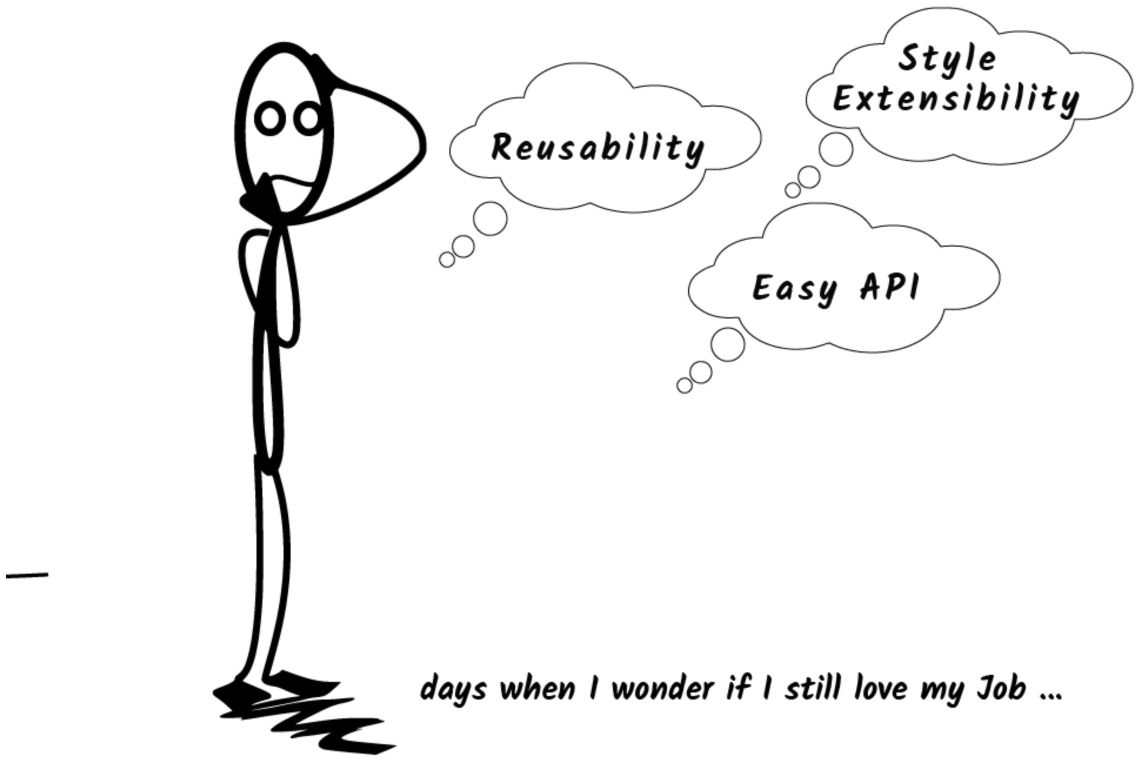
If you're completely new to these advanced patterns, don't worry, I'll explain them in detail. If you're familiar with how these patterns work from previous experiences with class components, I'll show you how to use these patterns with hooks.

Now, let's get started.

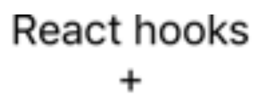
Why Advanced Patterns?

John's had a fairly good career. Today he's a senior frontend engineer at *ReactCorp*. A great startup changing the world for good.

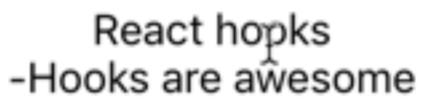
ReactCorp is beginning to scale their workforce. A lot of engineers are being hired and John's beginning to work on building reusable components for the entire team of engineers.



Here's what's rendered when not expanded:


React hooks
+

And when expanded:


React hooks
-Hooks are awesome

This works but it has to be the ugliest component I've ever seen. We can do better.

Manageable Styling for Reusable Components

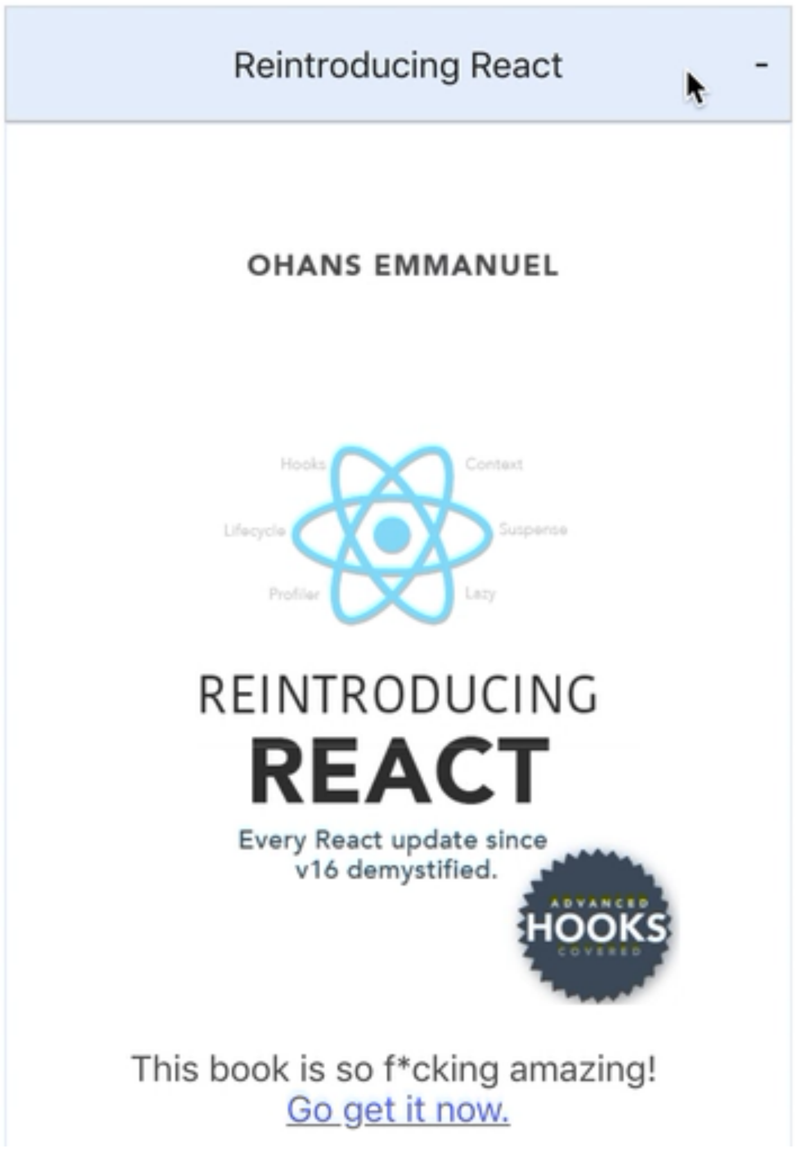
Hate it or not, styling (or CSS) is integral to how the web works.

While there's a number of ways to style a `React` component, and I'm sure you have a favourite, when you build reusable components it's always a good idea to expose a frictionless API for overriding default styles.

Usually, I recommend letting it possible to have your components stylable via both `style` and `className` props.

For example:

```
// this should work.
<MyComponent style={{name: "value"}} />
// and this.
<MyComponent className="my-class-name-with-dope-styles" />
```


Reintroducing React -

OHANS EMMANUEL

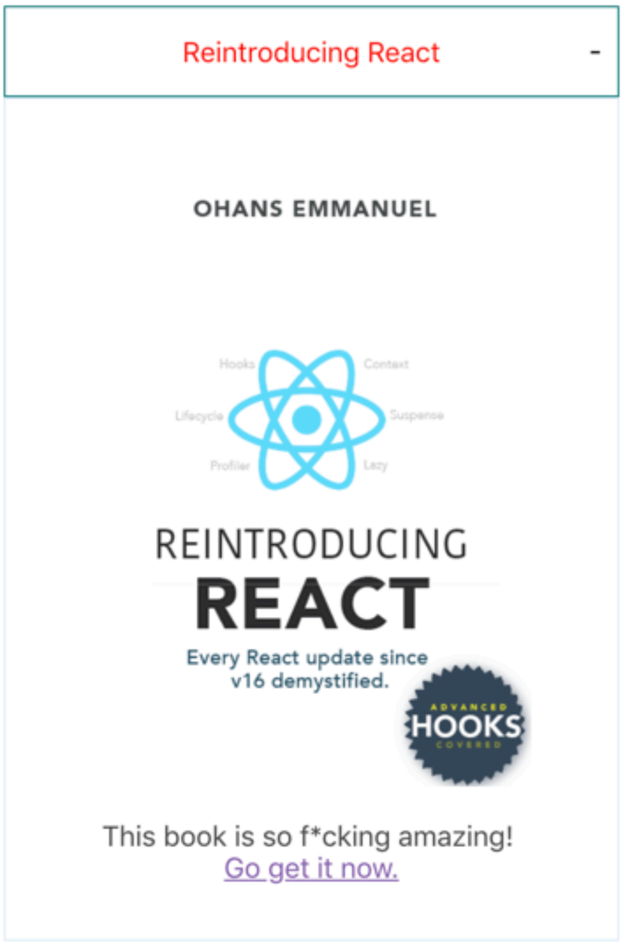
REINTRODUCING REACT

Every React update since v16 demystified.

This book is so f*cking amazing!
Go get it now.

And this didn't take a lot of code.

```
<Expandable>
    <Expandable.Header>Reintroducing React</Expandable.Header>
    <Expandable.Icon />
    <Expandable.Body>
        <img
            src='https://i.imgur.com/qpj4Y7N.png'
            style={{ width: '250px' }}
            alt='reintroducing react book cover'
```


Reintroducing React -

This book is so f*cking amazing!
Go get it now.

Yay! it works as expected.

Conclusion

This has been a lengthy discourse on Advanced React Patterns with Hooks. If you don't get all of it yet, spend a little more time practising these patterns in your day to day work, I'm pretty sure you'll get a hang of it real quick.

When you do, go be the React engineer building highly reusable components with advanced hook patterns.