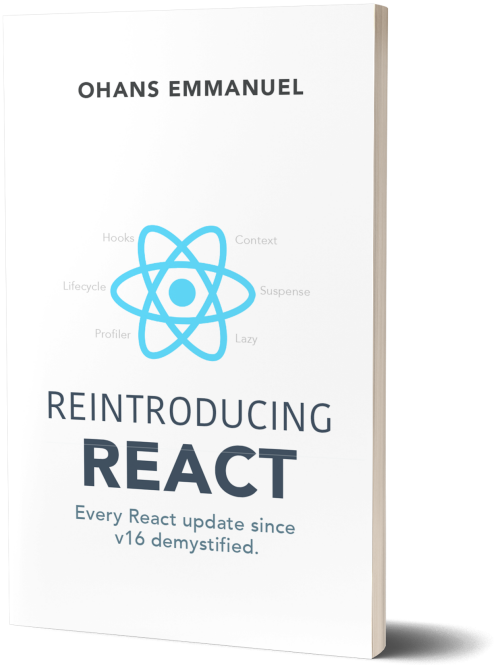

OHANS EMMANUEL
Hooks
Context
Lifecycle
Suspense
Profiler
Lazy
REINTRODUCING
REACT
Every React update since
v16 demystified.

Table Of Contents

Reintroducing React: Every React Update Since v16 Demystified.

In this book, unlike any you may 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.

Code Repository

Please find the book's associated code repository on Github. Useful for working alongside the examples discussed

Why Reintroduce React ?

I wrote my first React application 3 to 4 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.

On the bright side, I'm bringing you a one-stop reference to all the changes.

In this book, I'll take you on a journey – alongside some humour and unique content to follow.

Ready?

What's Changed since version 16?

If you think not much has changed, think again.

Here's a list of the relevant changes we'll be discussing in this guide:

- New Lifecycle Methods.
- Simpler State Management with the Context API.
- ContextType - Using Context without a Consumer.
- The Profiler: Using Charts and Interactions.
- Lazy Loading with React.Lazy and Suspense.
- Functional PureComponent with React.memo
- Simplifying React apps with Hooks!
- Advanced React Component Patterns with Hooks.

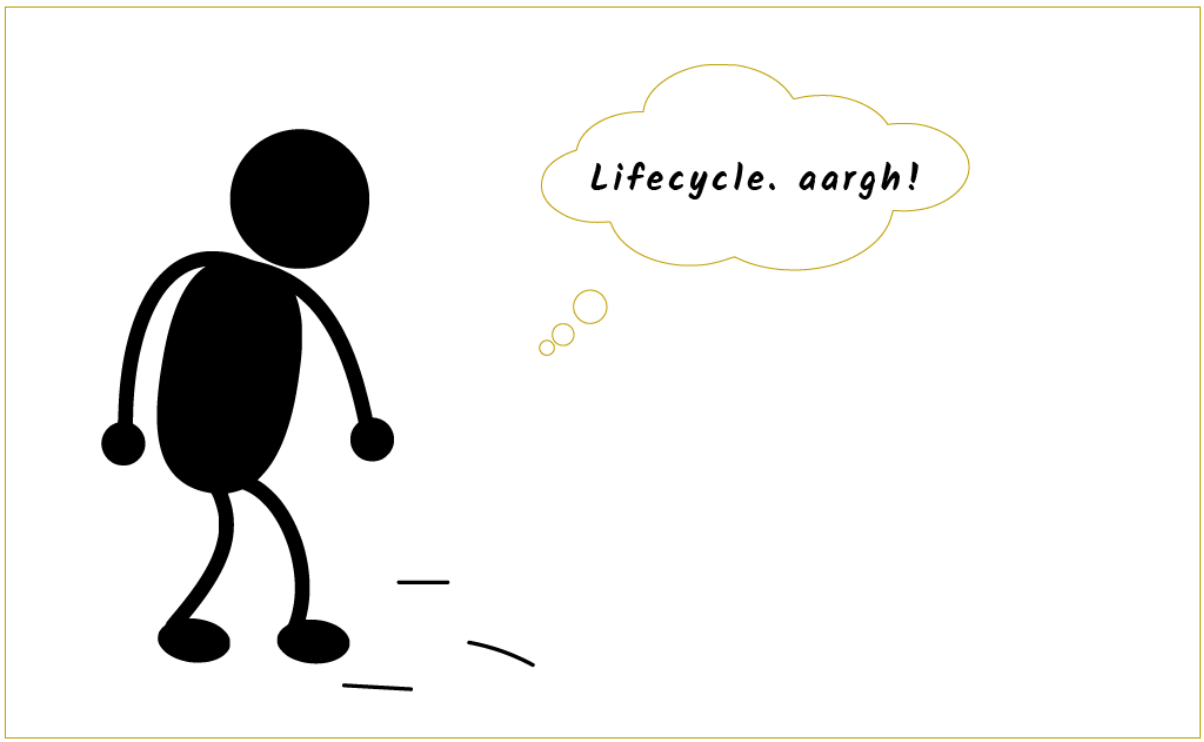
It goes without saying that a lot has been introduced since version 16. For ease, each of these topics have been broken down into separate sections.

In the next section I'll begin to discuss these changes by having a look at the new lifecycle methods available from version 16.

Chapter 1: New Lifecycle Methods.

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

Meet John.



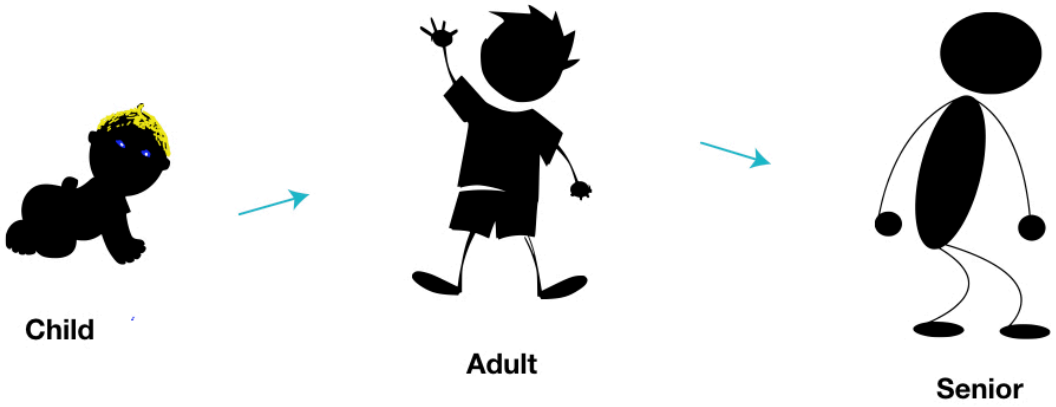
For a long time he didn't fully understand what lifecycle truly meant in the context of React apps.

When you think of lifecycle what comes to mind?

What's Lifecycle Anyway.

Well, consider humans.

The Human Lifecycle



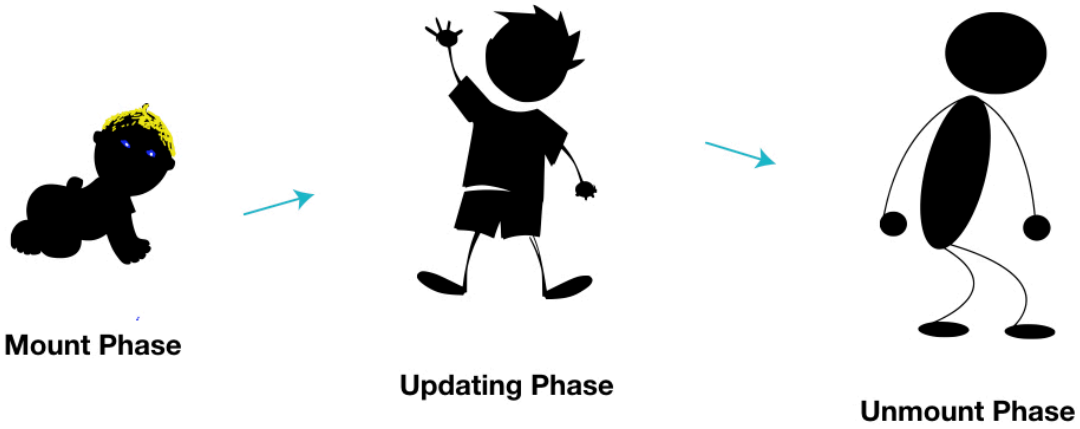
The typical lifecycle for a human is something like, “child” to “adult” to “elderly”.

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.

React Components?



The four essential phases or lifecycle attributed to a React component include:

- **Mounting** – like the birth of a child, at this phase the component is created (your code, and react's internals) then inserted into the DOM
- **Updating** – like humans "grow", in this phase a React component undergoes growth by being updated via changes in props or state.
- **Unmounting** – Like the death of a human, this is the phase the component is removed from the DOM.
- **Error Handling** – Think of this as being comparable to when humans fall sick and visit the doctor. Sometimes your code doesn't run or there's a bug somewhere. When this happens, the component is in the error handling phase. I intentionally skipped this phase in the illustration earlier.

Lifecycle Methods.

Now that you understand what lifecycle means. What are “lifecycle methods”?

Knowing the phases /lifecycle a React component goes through is one part of the equation. The other part is understanding the methods React makes available (or invokes) at each phase.

The methods invoked during different phase/lifecycle of a component is what’s popularly known as the component lifecycle methods e.g. In the mounting and updating phases, the `render` lifecycle method is always invoked.

There are lifecycle methods available on all 4 phases of a component — mounting, updating, unmounting and error handling.

Knowing when a lifecycle method is invoked (i.e the associated lifecycle/phase) means you can go ahead to write related logic within the method and know it’ll be invoked at the right time.

With the basics out of the way, let’s have a look at the actual new lifecycle methods available from version 16.

static getDerivedStateFromProps.

Before explaining how this lifecycle method works, let me show you how the method is used.

The basic structure looks like this:

```
const MyComponent extends React.Component {
  ...

  static getDerivedStateFromProps() {
     //do stuff here
  }
```

```
}
```

The method takes in `props` and `state`:

```
...

  static getDerivedStateFromProps(props, state) {
    //do stuff here
  }

...
```

And you can either return an object to update the state of the component:

```
...

  static getDerivedStateFromProps(props, state) {
     return {
        points: 200 // update state with this
     }
  }

  ...
```

Or return null to make no updates:

```
...

  static getDerivedStateFromProps(props, state) {
    return null
  }

...
```

I know what you're thinking. Why exactly is this lifecycle method important?

Well, it is one of the rarely used lifecycle methods, but it comes in handy in certain scenarios.

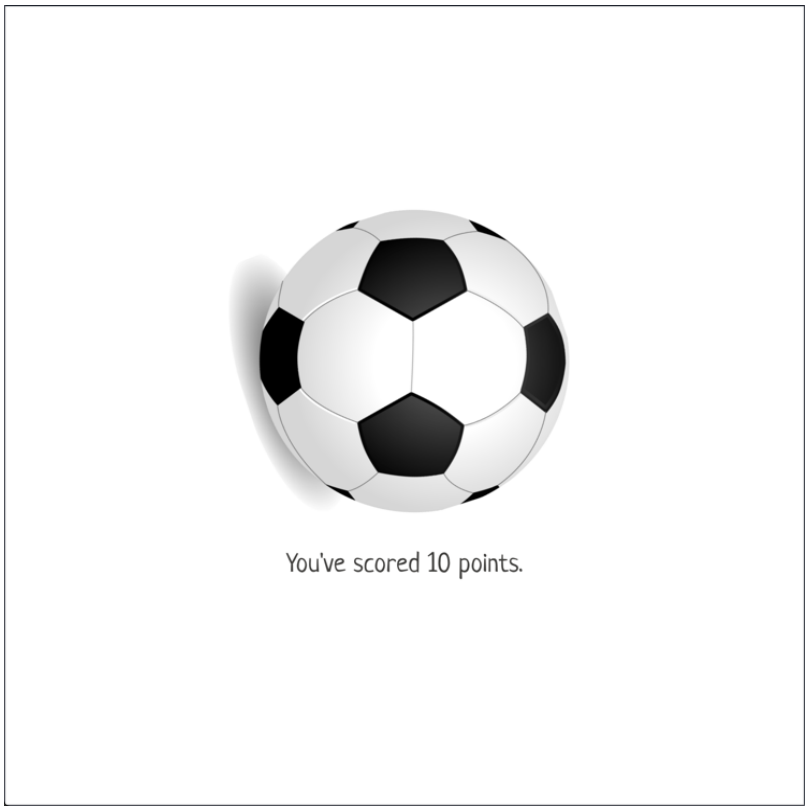
Firstly, this method is called (or invoked) **before** the component is rendered to the DOM on initial mount.

Below's a quick example.

Consider a simple component that renders the number of points scored by a football team.

As you may have expected, the number of points is stored in the component state object:

```
class App extends Component {
  state = {
    points: 10
  }
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            You've scored {this.state.points} points.
          </p>
        </header>
      </div>
    );
  }
}
```



Note that the text reads, *you have scored 10 points* – where 10 is the number of points in the state object.

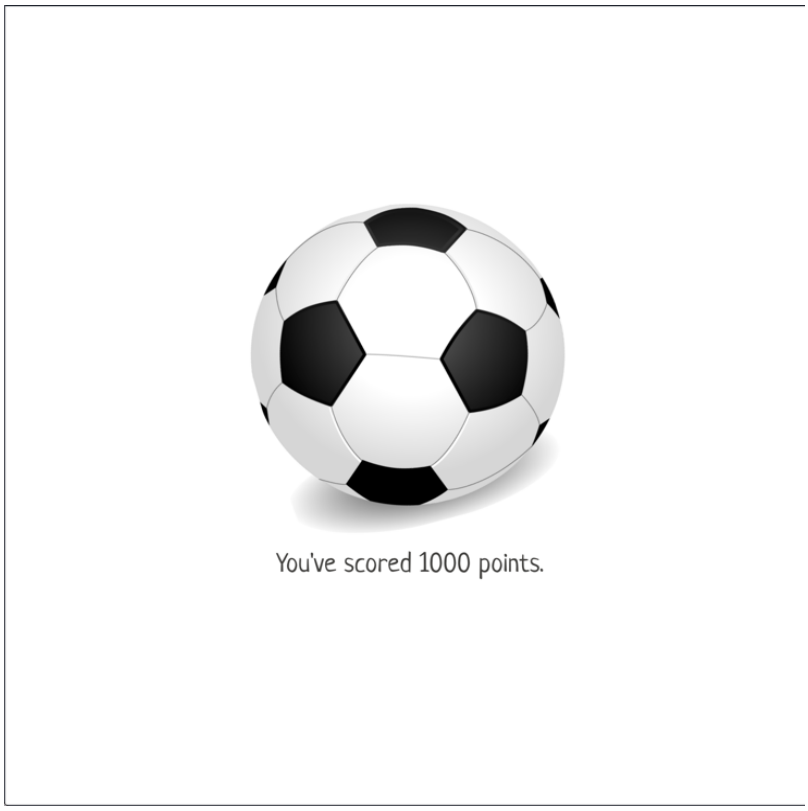
Just an as an example, if you put in the static getDerivedStateFromProps method as shown below, what number of points will be rendered?

```
class App extends Component {
  state = {
    points: 10
  }
  // *******
  //  NB: Not the recommended way to use this method. Just an example.
Unconditionally overriding state here is generally considered a bad idea
  // ********
  static getDerivedStateFromProps(props, state) {
    return {
      points: 1000
    }
```

```
  }
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            You've scored {this.state.points} points.
          </p>
        </header>
      </div>
    );
  }
}
```

Right now, we have the static getDerivedStateFromProps component lifecycle method in there. If you remember from the previous explanation, this method is called before the component is mounted to the DOM. By returning an object, we update the state of the component before it is even rendered.

And here's what we get:



With the 1000 coming from updating state within the `static getDerivedStateFromProps` method.

Well, this example is contrived, and not really the way you'd use the `static getDerivedStateFromProps` method. I just wanted to make sure you understood the basics first.

With this lifecycle method, just because you can update state doesn't mean you should go ahead and do this. There are specific use cases for the `static getDerivedStateFromProps` method, or you'll be solving a problem with the wrong tool.

So when should you use the `static getDerivedStateFromProps` lifecycle method?

The method name `getDerivedStateFromProps` comprises five different words, "Get Derived State From Props".

`getDerivedStateFromProps`

Also, component state in this manner is referred to as Derived State.

As a rule of thumb, derived state should be used sparingly as you can introduce subtle bugs into your application if you aren't sure of what you're doing.

getSnapshotBeforeUpdate.

In the updating component phase, right after the render method is called, the `getSnapshotBeforeUpdate` lifecycle method is called next.

This one is a little tricky, but I'll take my time to explain how it works.

Chances are you may not always reach out for this lifecycle method, but it may come in handy in certain special cases. Specifically when you need to grab some information from the DOM (and potentially change it) just after an update is made.

Here's the important thing. The value queried from the DOM in `getSnapshotBeforeUpdate` will refer to the value just before the DOM is updated. Even though the render method was previously called.

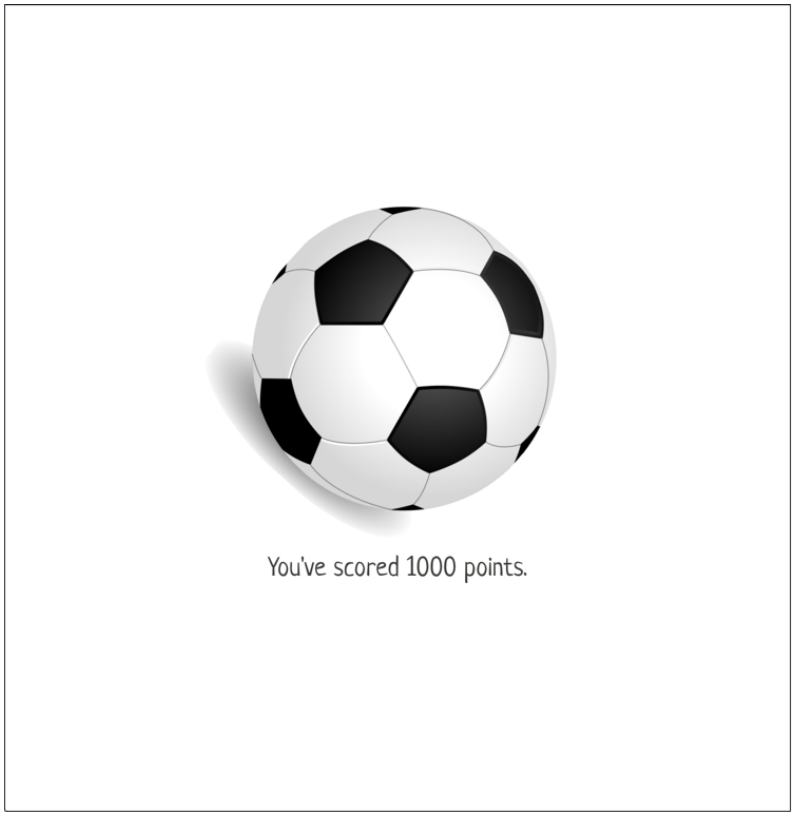
An analogy that may help has to do with how you use version control systems such as git.

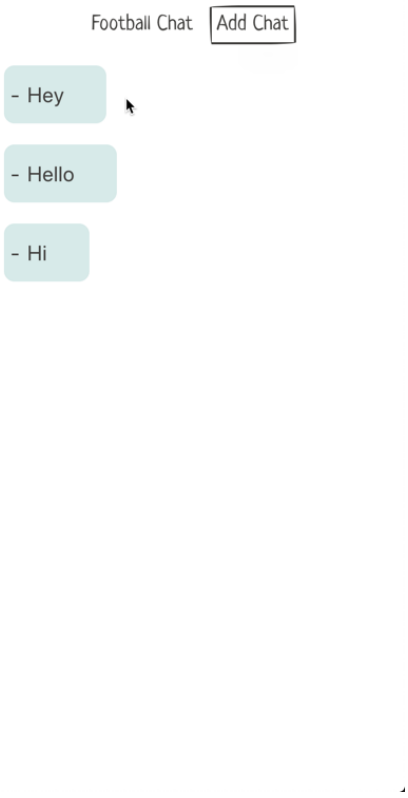
A basic example is that you write code, and stage your changes before pushing to the repo.

In this case, assume the render function was called to stage your changes before actually pushing to the DOM. So, before the actual DOM update, information retrieved from getSnapshotBeforeUpdate refers to those before the actual visual DOM update.

Actual updates to the DOM may be asynchronous, but the `getSnapshotBeforeUpdate` lifecycle method will always be called immediately before the DOM is updated.

Don't worry if you don't get it yet. I have an example for you.





The implementation of the chat pane is as simple as you may have imagined. Within the `App` component is an unordered list with a `Chats` component:

```
<ul className="chat-thread">
    <Chats chatList={this.state.chatList} />
 </ul>
```

The `Chats` component renders the list of chats, and for this, it needs a chatList prop. This is basically an Array. In this case, an array of 3 string values, "Hey", "Hello", "Hi".

The `Chats` component has a simple implementation as follows:

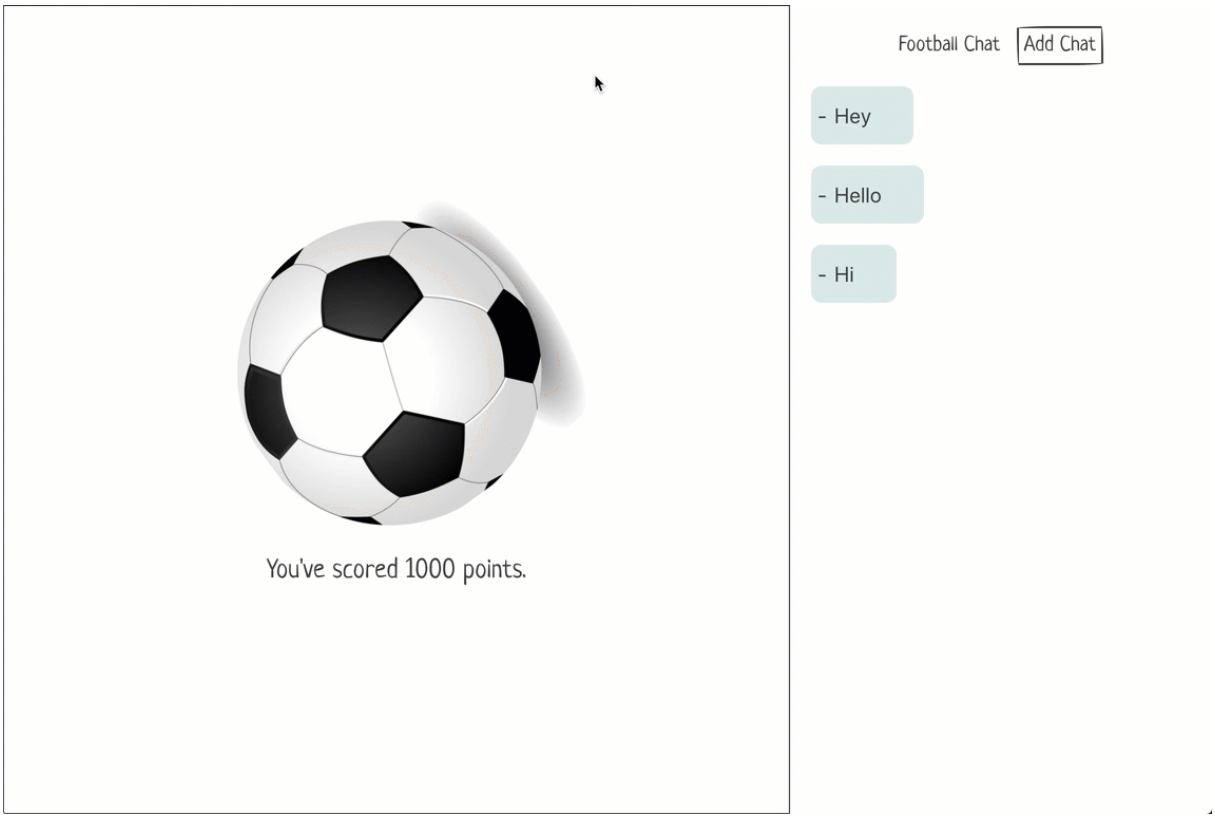
```
class Chats extends Component {
  render() {
    return (
      <React.Fragment>
        {this.props.chatList.map((chat, i) => (
          <li key={i} className="chat-bubble">
            {chat}
          </li>
        ))}
      </React.Fragment>
    );
  }
}
```

It just maps through the `chatList` prop and renders a list item which is in turn styled to look like a chat bubble :).

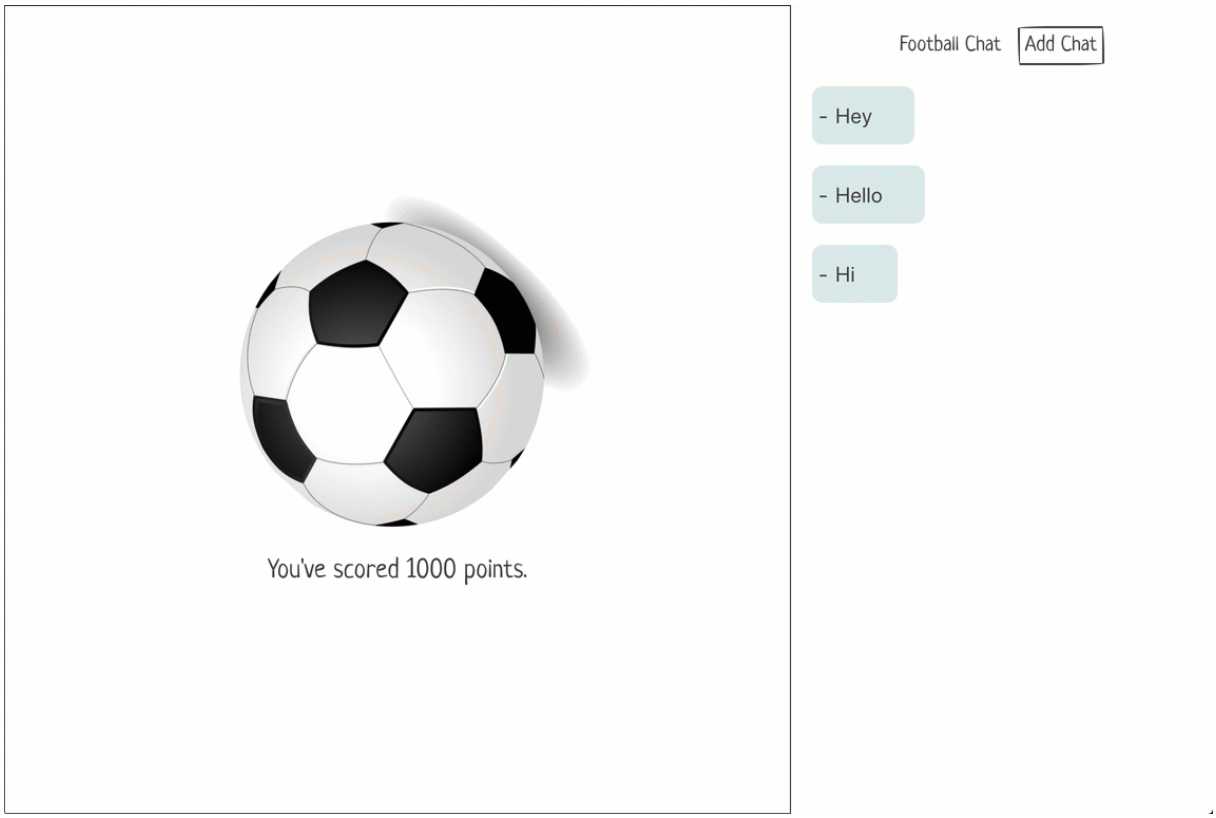
There's one more thing though. Within the chat pane header is an "Add Chat" button.

Clicking this button will add a new chat text, "Hello", to the list of rendered messages.

Here's that in action:



The problem here, as with most chat applications is that whenever the number of chat messages exceeds the available height of the chat window, the expected behaviour is to auto scroll down the chat pane so that the latest chat message is visible. That's not the case now.



Let's see how we may solve this using the getSnapshotBeforeUpdate lifecycle method.

The way the `getSnapshotBeforeUpdate` lifecycle method works is that when it is invoked, it gets passed the previous props and state as arguments.

So we can use the `prevProps` and `prevState` parameters as shown below:

```
getSnapshotBeforeUpdate(prevProps, prevState) {


}
```

Within this method, you're expected to either return a value or null:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
   return value || null // where 'value' is a  valid JavaScript value
}
```

Whatever value is returned here is then passed on to another lifecycle method. You'll get to see what I mean soon.

The `getSnapshotBeforeUpdate` lifecycle method doesn't work on its own. It is meant to be used in conjunction with the `componentDidUpdate` lifecycle method.

Whatever value is returned from the `getSnapshotBeforeUpdate` lifecycle method is passed as the third argument to the `componentDidUpdate` method.

Let's call the returned value from `getSnapshotBeforeUpdate`, *snapshot*, and here's what we get thereafter:

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

The `componentDidUpdate` lifecycle method is invoked after the `getSnapshotBeforeUpdate` is invoked. As with the getSnapshotBeforeUpdate method it receives the previous props and state as arguments. It also receives the returned value from `getSnapshotBeforeUpdate` as final argument.

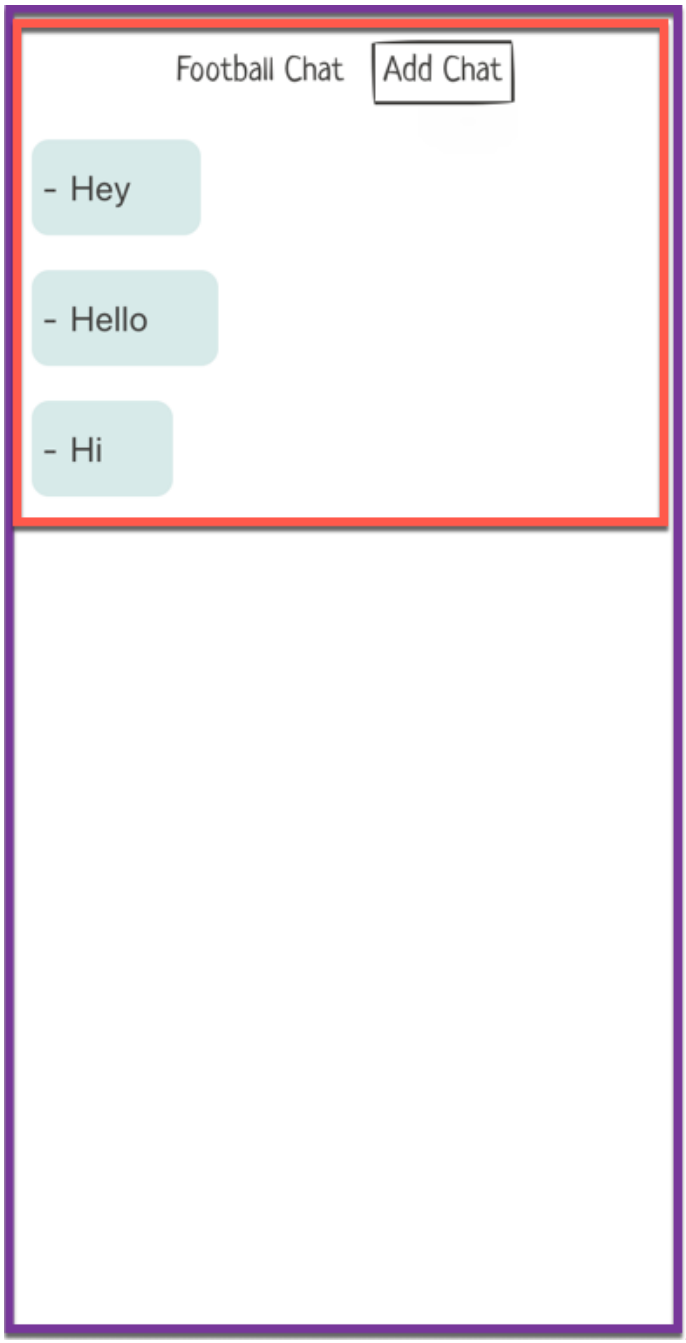
Here's all the code required to maintain the scroll position within the chat pane:

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

Let me explain what's going on there.

Below's the chat window:

However, the graphic below highlights the actual region that holds the chat messages (the unordered list, `ul` which houses the messages).



It is this `ul` we hold a reference to using a React Ref:

```
<ul className="chat-thread" ref={this.chatThreadRef}>
   ...
</ul>
```

First off, because `getSnapshotBeforeUpdate` may be triggered for updates via any number of props or even a state update, we wrap to code in a conditional that checks if there's indeed a new chat message:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    if (this.state.chatList > prevState.chatList) {
      // write logic here
    }
}
```

The `getSnapshotBeforeUpdate` method above has to return a value yet. If no chat message was added, we will just return `null`:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    if (this.state.chatList > prevState.chatList) {
      // write logic here
    }
    return null
}
```

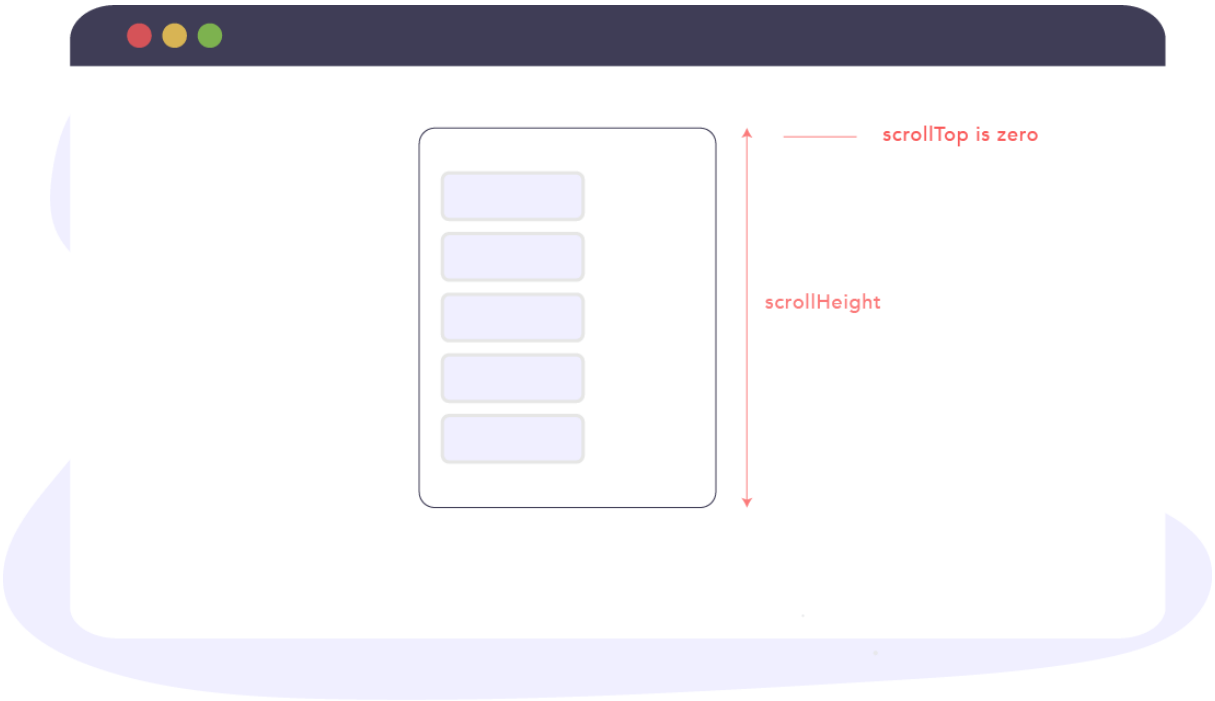
Now consider the full code for the `getSnapshotBeforeUpdate` method:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    if (this.state.chatList > prevState.chatList) {
      const chatThreadRef = this.chatThreadRef.current;
      return chatThreadRef.scrollHeight - chatThreadRef.scrollTop;
    }
    return null;
  }
```

Does it make sense to you?

Not yet, I suppose.

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.

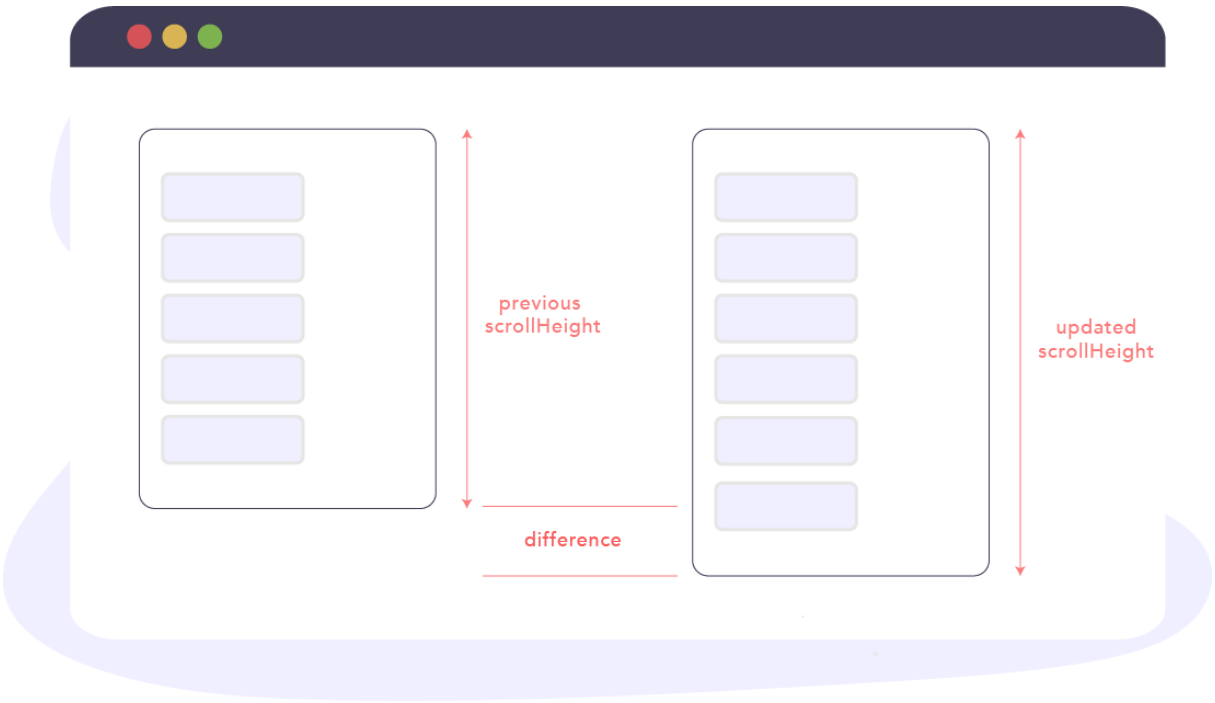
In the `componentDidUpdate` we have the following code, but what does it do?

```
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`;

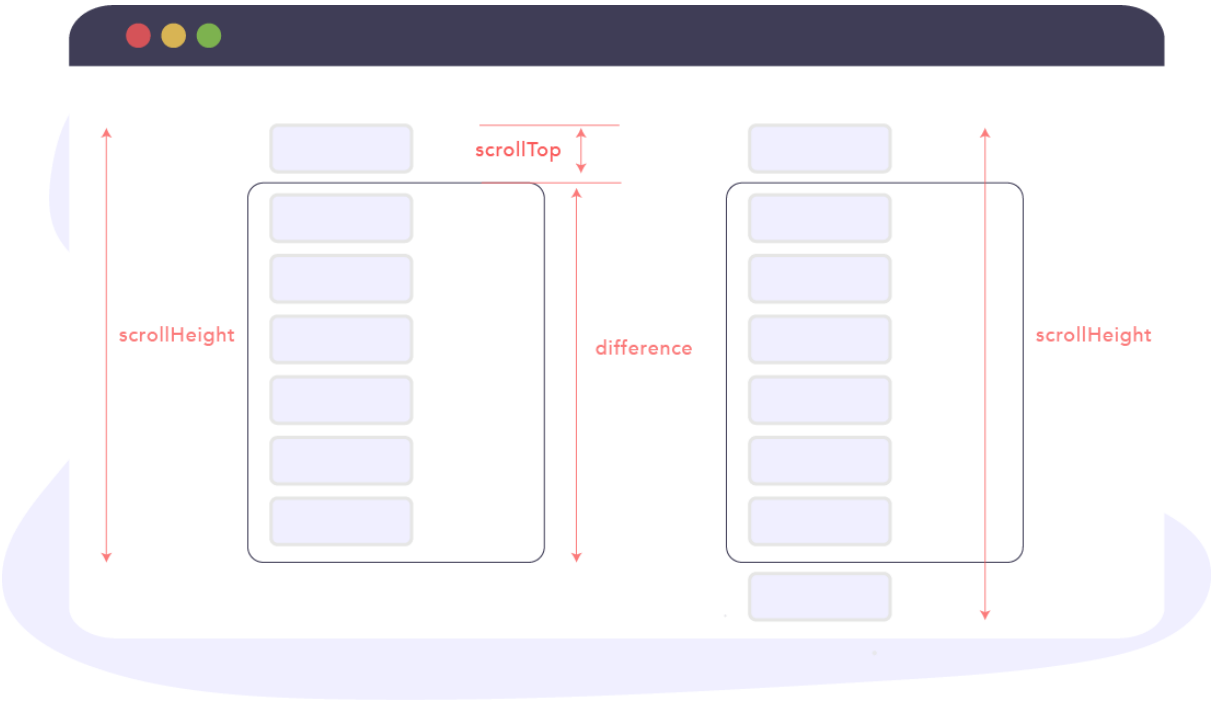
Since `snapshot` refers to the `scrollHeight` before the update, the above expression returns the `height` of the new chat message plus any other related height owing to the update.

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.

The computation from `componentDidUpdate` will set to `scrollTop` value to the sum of the heights of extra messages - exactly what we want.



Yeah, that's it.

If you got stuck, I'm sure going through the explanation (one more time) or checking the source code will help clarify your questions.

The Error Handling Lifecycle Methods.

Sometimes things go bad, errors are thrown. The error lifecycle methods are invoked when an error is thrown by a descendant component i.e a component below them.

Let's implement a simple component to catch errors in the demo app. For this, we'll create a new component called `ErrorBoundary`.

Here's the most basic implementation:

```
import React, { Component } from 'react';
class ErrorBoundary extends Component {
  state = {};
  render() {
    return null;
  }
}
export default ErrorBoundary;
```

Now, let's incorporate the error lifecycle methods.

static getDerivedStateFromError.

Whenever an error is thrown in a descendant component, this method is called first, and the error thrown passed as an argument.

Whatever value is returned from this method is used to update the state of the component.

Let's update the `ErrorBoundary` component to use this lifecycle method.

```
import React, { Component } from "react";
class ErrorBoundary extends Component {
  state = {};

  static getDerivedStateFromError(error) {
    console.log(`Error log from getDerivedStateFromError: ${error}`);
    return { hasError: true };
  }

  render() {
    return null;
```

```
  }
}

export default ErrorBoundary;
```

Right now, whenever an error is thrown in a descendant component, the error will be logged to the console, `console.error(error)`, and an object is returned from the `getDerivedStateFromError` method. This will be used to update the state of the ErrorBoundary component i.e with `hasError: true`.

componentDidCatch.

The `componentDidCatch` method is also called after an error in a descendant component is thrown. Apart from the error thrown, it is passed one more argument which represents more information about the error:

```
componentDidCatch(error, info) {
}
```

In this method, you can send the `error` or `info` received to an external logging service. Unlike `getDerivedStateFromError`, the `componentDidCatch` allows for side-effects:

```
componentDidCatch(error, info) {
    logToExternalService(error, info) // this is allowed.
        //Where logToExternalService may make an API call.
}
```

Let's update the `ErrorBoundary` component to use this lifecycle method:

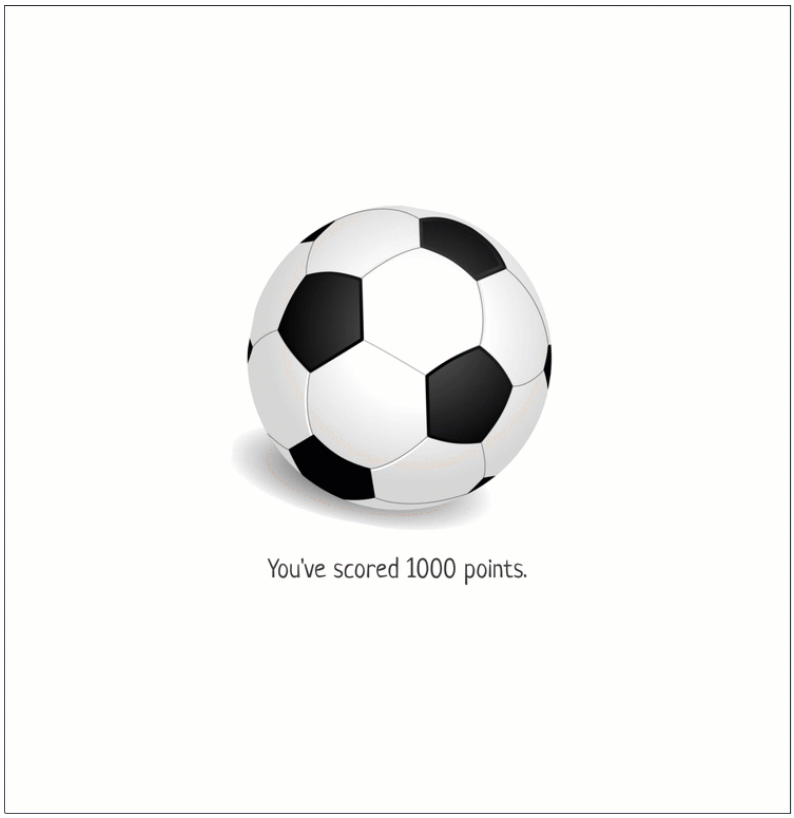
```
import React, { Component } from "react";
class ErrorBoundary extends Component {
  state = { hasError: false };
  static getDerivedStateFromError(error) {
```

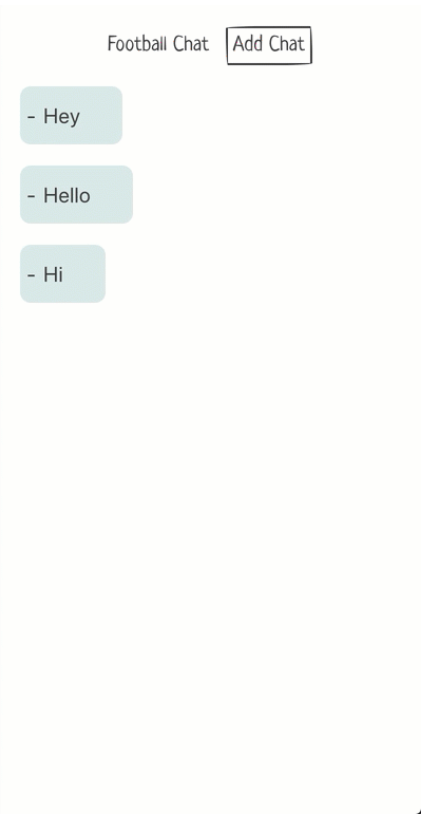
```
    console.log(`Error log from getDerivedStateFromError: ${error}`);
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    console.log(`Error log from componentDidCatch: ${error}`);
    console.log(info);
  }
  render() {
    return null
  }
}
export default ErrorBoundary;
```

Also, since the `ErrorBoundary` can only catch errors from descendant components, we'll have the component render whatever is passed as `Children` or render a default error UI if something went wrong:

```
...
render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
 }
```

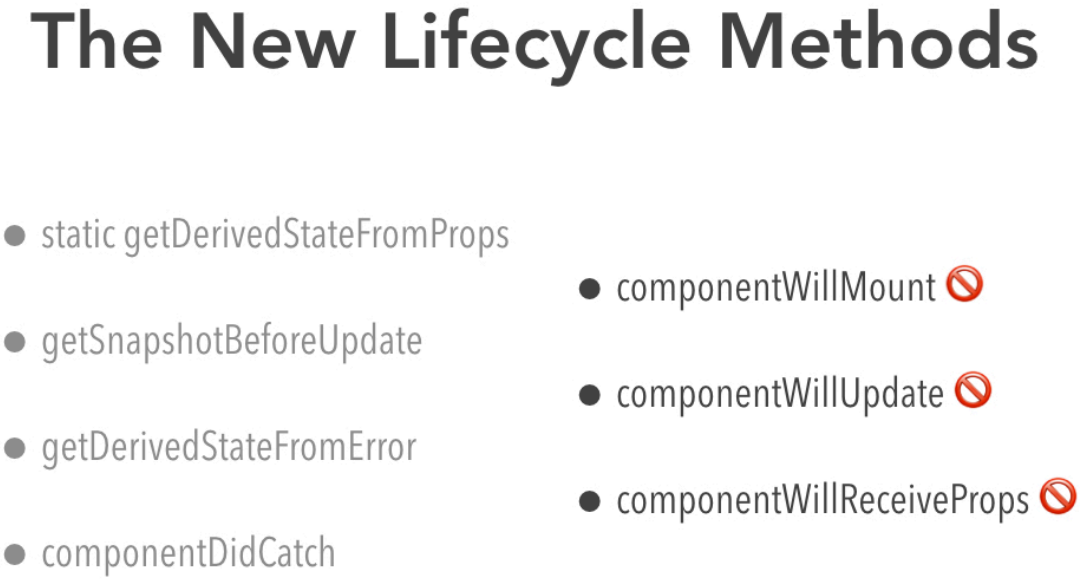
I have simulated a javascript error whenever you add a 5th chat message. Have a look at the error boundary at work:





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.



Now you're up to date on the changes made to the component lifecycle methods since React version 16!

Click to Tweet.

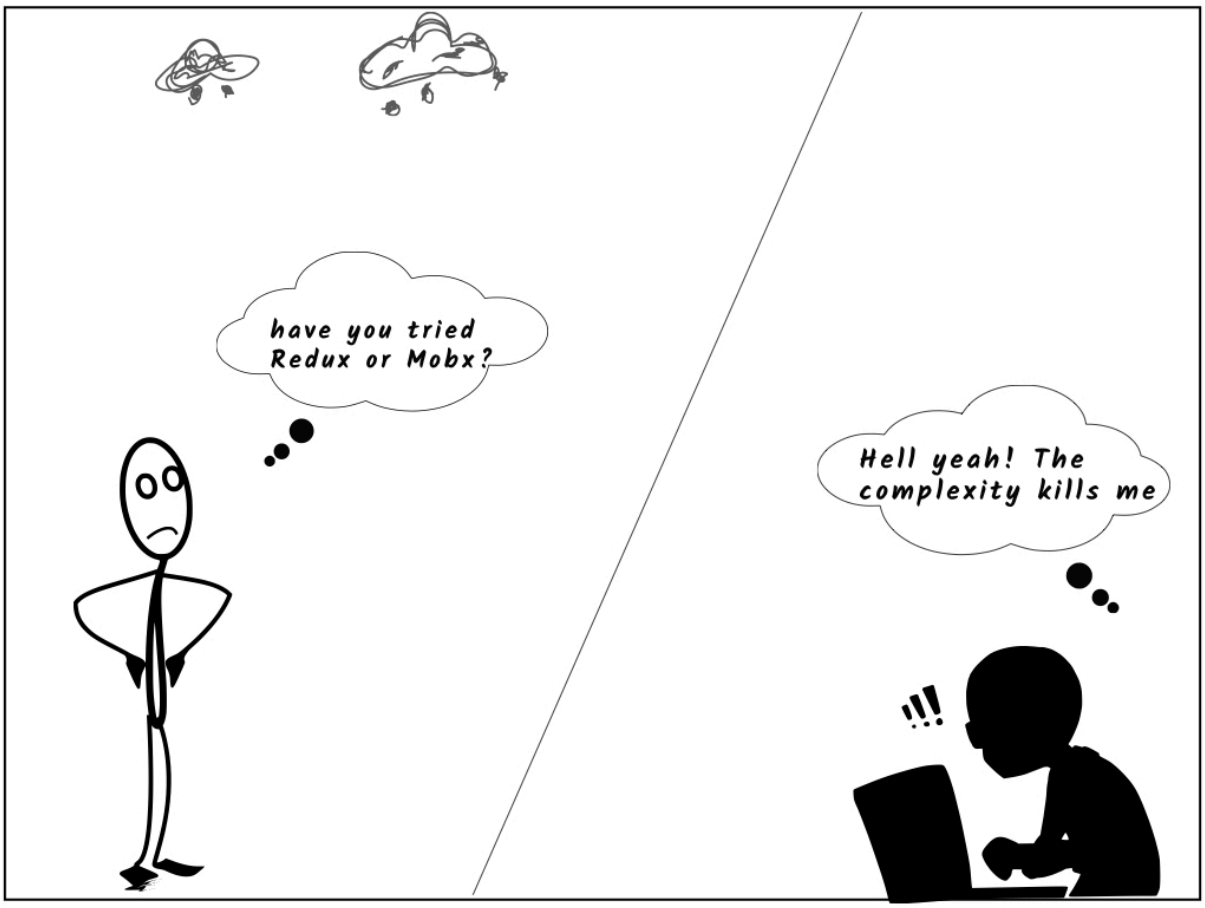
Chapter 2: Simpler State Management with the Context API.

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.



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.

Mia goes on a desperate search to help a friend in need, and she finds the Context API.

Mia recommends using React's Context API to solve the problem. John is now happy, excited to see what the Context API could offer, and he goes about his work productively.

This marks the beginning of John's experience with the Context API.

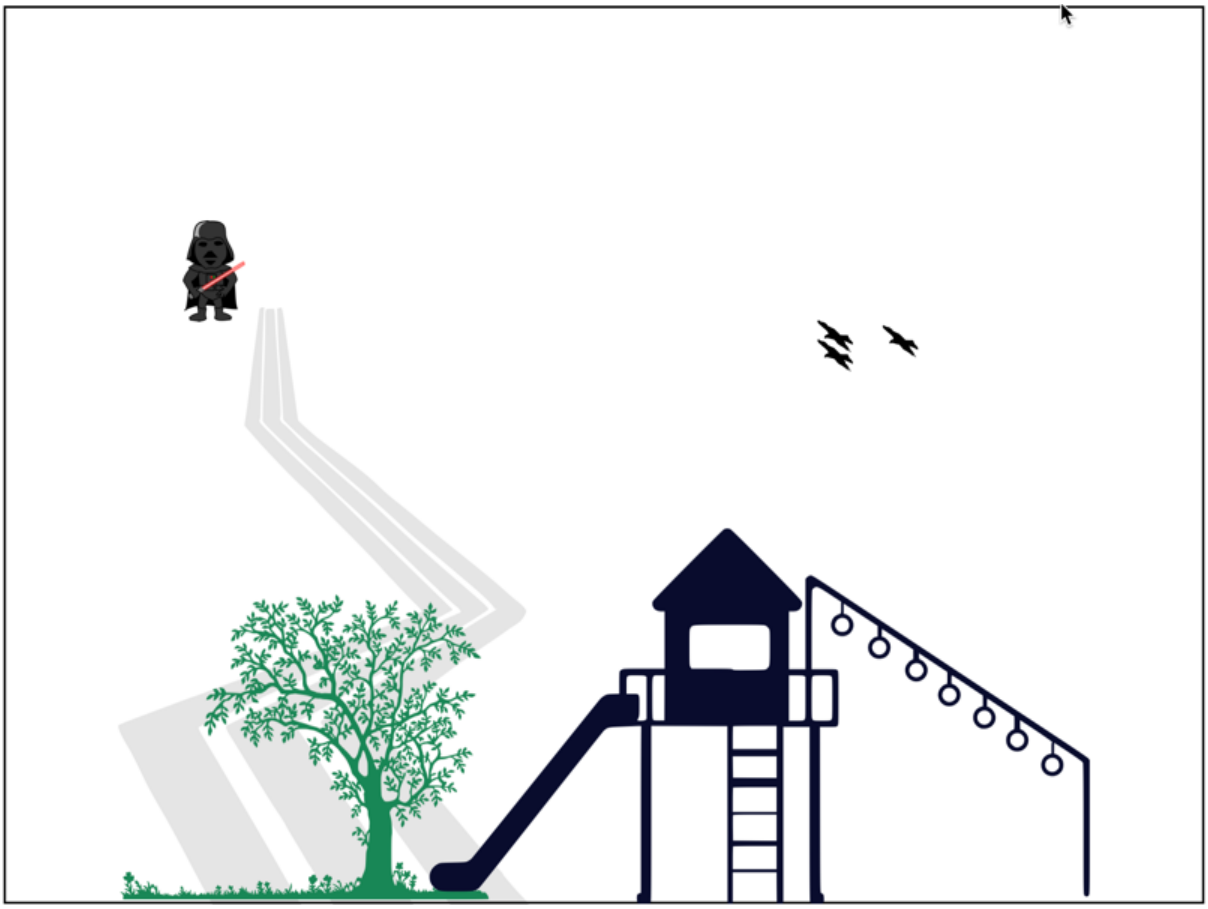
Introduction to Context.

The Context API exists to make it easy to share data considered "global" within a component tree.

Let's have a look at an illustrated example before we delve into writing any code.

Well, John has began working with the Context API and has mostly been impressed with it so far. Today he has a new project to work on, and he intends to use the context API.

Let's see what this new project is about.

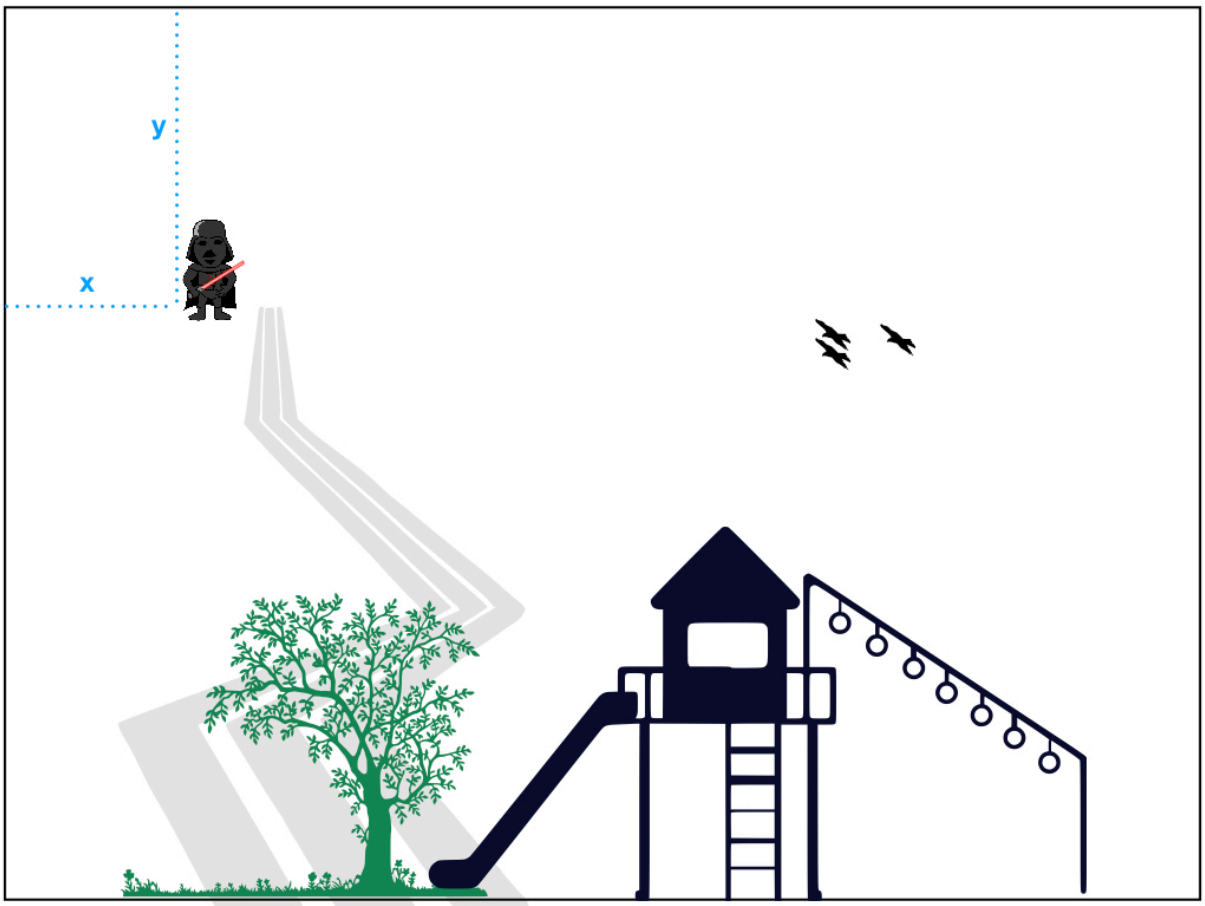


John is expected to build a game for a new client of his. This game is called **Benny Home Run**, and it seems like a great place to use the Context API.

The sole aim of the game is to move Benny from his start position to his new home.

To build this game, John must keep track of the position of Benny.

Since Benny's position is such an integral part of the entire application, it may as well be tracked via some global application state.



Did I just say "*global application state*"?

Yeah!

That sounds like a good fit for the Context API.

So, how may John build this?

```
import {createContext} from 'react';
const BennyPositionContext = createContext({ x: 50, y: 50 })
const { Provider, Consumer } = BennyPositionContext
```

First, there's the need to import the `createContext` method from `React`

```
import {createContext} from 'react';
```

The `createContext` method allows you to create what's referred to as a context object. Consider this to be the data structure that powers retrieving and saving state values.

To create a context object, you invoke the `createContext` method with (or without) an initial state value to be saved in the context object.

```
createContext(initialStateValue)
```

Here's what that looks like in the Benny app:

```
const BennyPositionContext = createContext({
```

```
    x: 50,
    y: 50
})
```

The `createContext` method is invoked with an initial state value corresponding to the initial position values (x and y) for Benny.

Looks good!

But, after creating a context object, how exactly do you gain access to the state values within your application?

Well, every context object comes with a `Provider` and `Consumer` component.

The `Provider` component "provides" the value saved in the context object to its children, while the `Consumer` component "consumes" the values from within any child component.

I know that was a mouth full, so let's break it apart slowly.

In the Benny example, we can go ahead and destructure the `BennyPositionContext` to retrieve the Provider and Consumer components.

```
const BennyPositionContext = createContext({
    x: 50,
    y: 50
})
// get provider and consumer
const { Provider, Consumer } = BennyPositionContext
```

Since `Provider` provides value saved in the context object to its **children**, we could wrap a tree of components with the `Provider` component as shown below:

```
<Provider>
   <Root /> // the root component for the Benny app.
</Provider>
```

Now, any child component within the `Root` component will have access to the default values stored in the context object.

Consider the following tree of components for the Benny app.

```
<Provider>
   <Scene>
     <Benny />
   </Scene>
</Provider>
```

Where `Scene` and `Benny` are children of the `Root` component and represent the game scene and benny character respectively.

In this example, the `Scene` or the even more nested `Benny` component will have access to the value provided by the `Provider` component.

It is worth mentioning that a `Provider` also takes in a `value` props.

This `value` prop is useful if you want to provide a value other than the initial value passed in at the context object creation time via `createContext(initialStateValue)`

Here's an example where a new set of values are passed in to the `Provider` component.

```
<Provider value={x: 100, y: 150}>
   <Scene>
     <Benny />
   </Scene>
</Provider>
```

Now that we have values provided by the `Provider` component, how can a nested component such as `Benny` consume this value?

```
...

<Provider>
  <Scene>
    <Benny />
  </Scene>
</Provider>
```



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

The simple answer is by using the `Consumer` component.

Consider the `Benny` component being a simple component that renders some SVG.

```
const Benny = () => {
  return <svg />
}
```

Now, within `Benny` we can go ahead and use the `Consumer` component like this:

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

```
}
```

Okay, what's going on there?

The `Consumer`component exposes a render prop API i.e `children` is a function. This function is then passed arguments corresponding to the values saved in the context object. In this case, the `position` object with the `x` and `y` coordinate values.

It is worth noting that whenever the value from a `Provider` component changes, the associated `Consumer` component and the children will be re-rendered to keep the value(s) consumed in sync.

Also, a `Consumer` will receive values from the closest `Provider` above it in the tree.

Consider the situation below:

```
// create context object
const BennyPositionContext = createContext({
    x: 50,
    y: 50
})
// get provider and consumer
const { Provider, Consumer } = BennyPositionContext
// wrap Root component in a Provider
<Provider>
    <Root />
</Provider>
// in Benny, within Root.
const Benny = () => (
    <Provider value={x: 100, y: 100}>
        // do whatever
    </Provider>
)
```

Now, with a new provider component introduced in `Benny`, any `Consumer` within `Benny` will receive the value `{x: 100, y: 100}` NOT the initial value of `{x: 50, y: 50}`.

This is a contrived illustrated example, but it helps solidify the foundations of using the Context API.

Having understood the necessary building blocks for using the Context API, let's build an application utilizing all you've learned so far, with extra use cases discussed.

Example: The Mini-Bank Application.

John's an all round focused guy. When he's not working on projects from his work place, he dabbles into side projects.

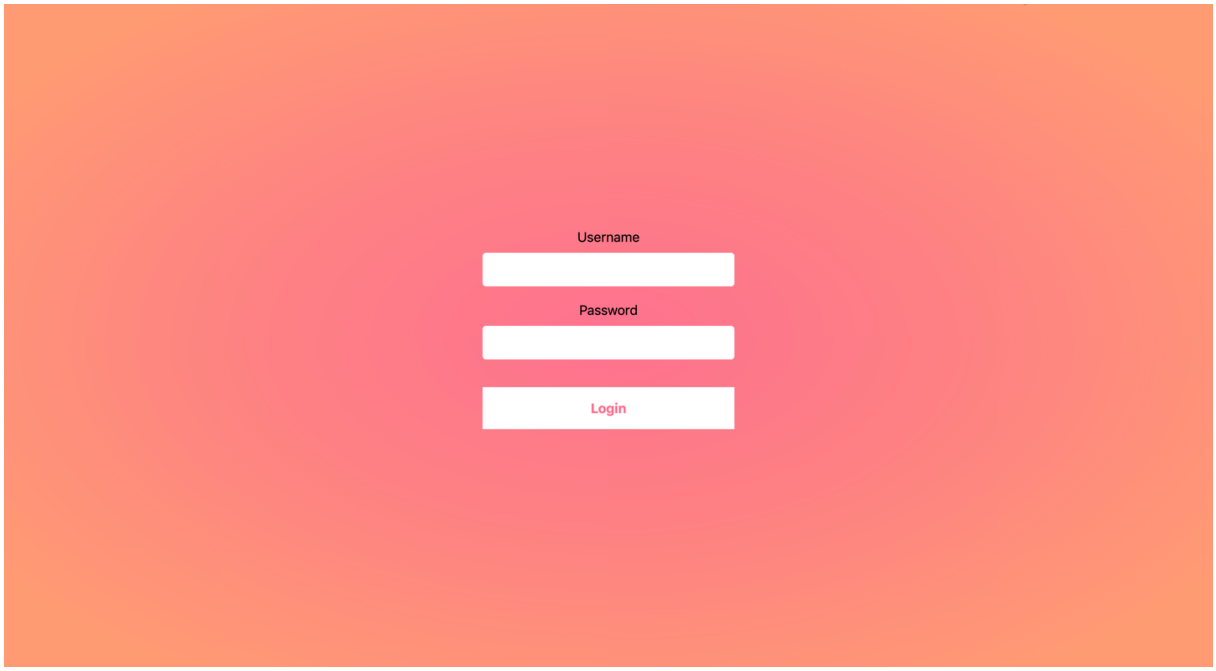
One of his many side projects is a bank application he thinks could shape the future of banking. How so.

I managed to get the source code for this application. You'll find it in the code repository for the book as well.

To get started, please Install the dependencies and run the application by following the commands below:

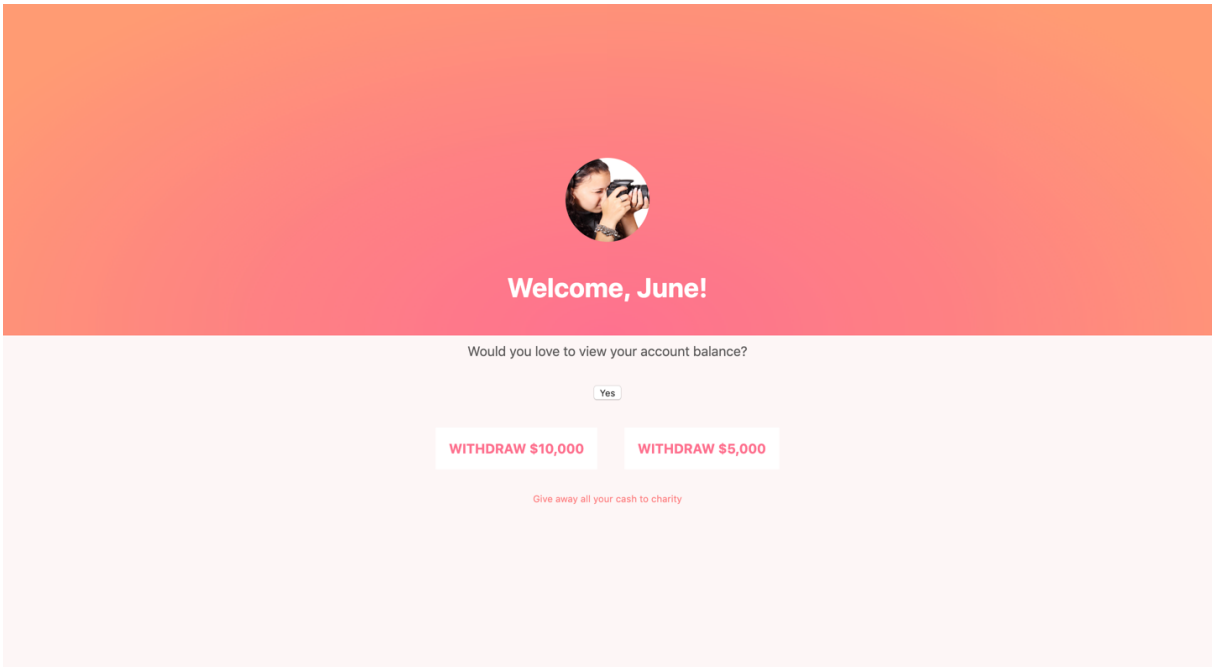
```
cd 02-Context-API/bank-app

yarn install

yarn start
```

Once that's done, the application starts off with a login screen as seen below:

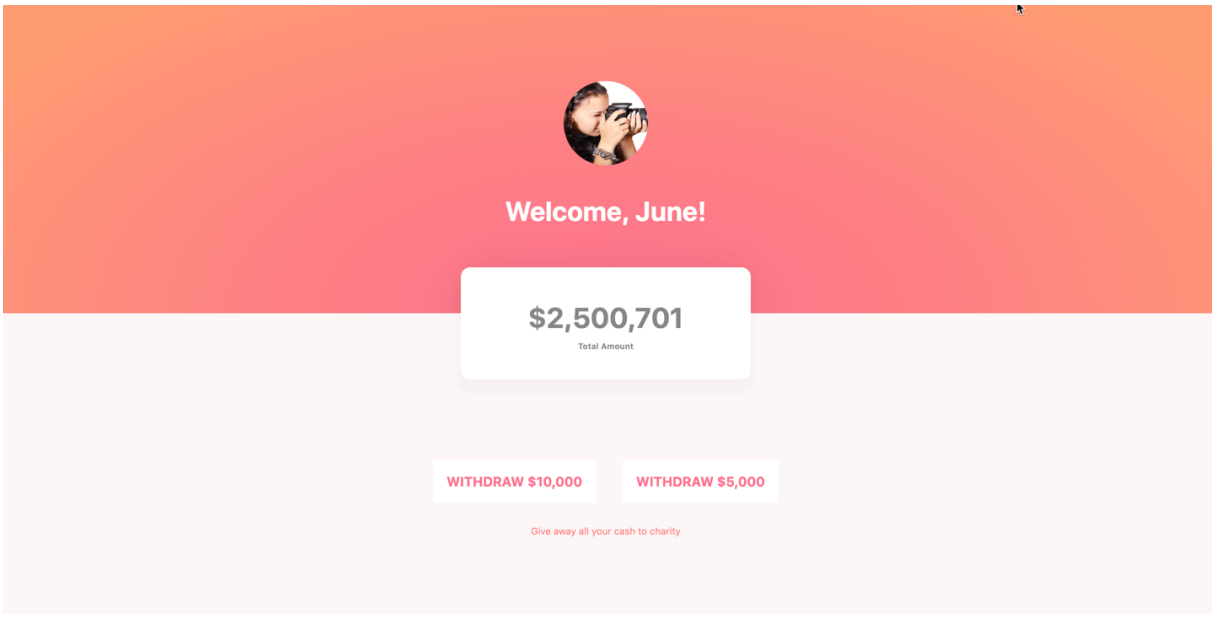


You can enter whatever username and password combination of your choosing.

Upon login in you'll be presented with the application's main screen shown below:



In the main screen you can perform actions such as viewing your bank account balance and making withdrawals as well.



Our goal here is to manage the state in this application a lot better by introducing React's Context.

Identifying Props being Drilled.

The root component of the application is called `Root`, and has the implementation below:

```
...
import { USER } from './api'
class Root extends Component {
  state = {
    loggedInUser: null
  }
  handleLogin = evt => {
    evt.preventDefault()
    this.setState({
      loggedInUser: USER
    })
  }
```

```
  render () {
    const { loggedInUser } = this.state

    return loggedInUser ? (
      <App loggedInUser={loggedInUser} />
    ) : (
      <Login handleLogin={this.handleLogin} />
    )
  }
}
```

If the user is logged in, the main component `App` is rendered, else we show the `Login` component.

```
...
loggedInUser ? (
      <App loggedInUser={loggedInUser} />
    ) : (
   <Login handleLogin={this.handleLogin} />
)
...
```

Upon a successful login (which doesn't require any particular username and password combinations), the `state` of the `Root` application is updated with a `loggedInUser`.

```
...
handleLogin = evt => {
    ...
    this.setState({
      loggedInUser: USER
    })
  }
```

```
...
```

In the real world, this could come from an `api` call.

For this application, I've created a fake user in the `api` directory that exports the following user object.

```
export const USER = {
  name: 'June',
  totalAmount: 2500701
}
```

Basically, the `name` and `totalAmount` of the user's bank account are retrieved and set to state when you log in.

How's the user object used in the application?

Well, consider the main component, `App`. This is the component responsible for rendering everything in the app other than the `Login` screen.

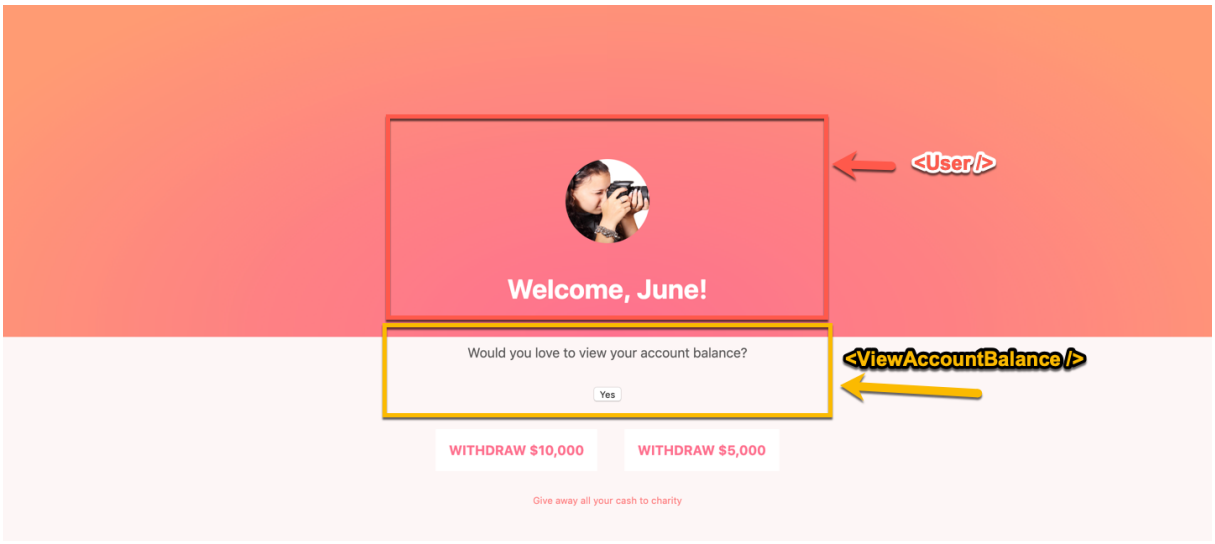
Here's the implementation of that:

```
class App extends Component {
  state = {
    showBalance: false
  }

  displayBalance = () => {
    this.setState({ showBalance: true })
  }
  render () {
    const { loggedInUser } = this.props
    const { showBalance } = this.state

    return (
      <div className='App'>
```

```
        <User loggedInUser={loggedInUser} profilePic={photographer} />

        <ViewAccountBalance
          showBalance={showBalance}
          loggedInUser={loggedInUser}
          displayBalance={this.displayBalance}
        />

        <section>
          <WithdrawButton amount={10000} />
          <WithdrawButton amount={5000} />
        </section>

        <Charity />

      </div>
    )
  }
}
```

It's a lot simpler that it seems. Have a second look!

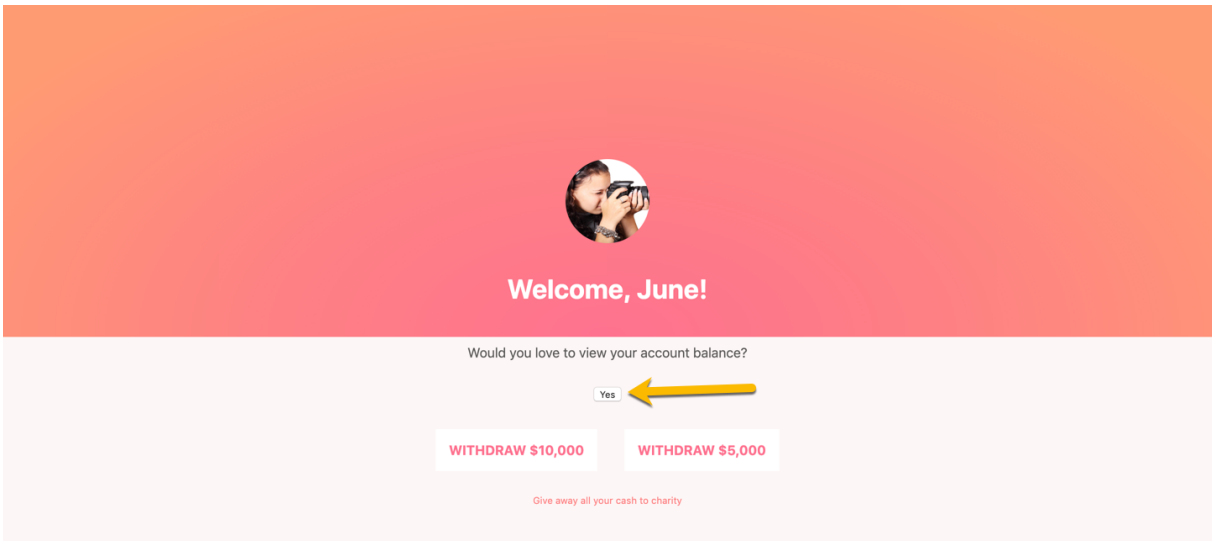
The `loggedInUser` is passed as props to `App` from `Root`, and is also passed down to both `User` and `ViewAccountBalance` components.



The `User` component receives the `loggedInUser` prop and passes it down to another child component, `Greeting` which renders the text, "*Welcome, June*".

```
//User.js


const User = ({ loggedInUser, profilePic }) => {
  return (
    <div>
      <img  src={profilePic} alt='user' />
      <Greeting loggedInUser={loggedInUser} />
    </div>
  )
}
```

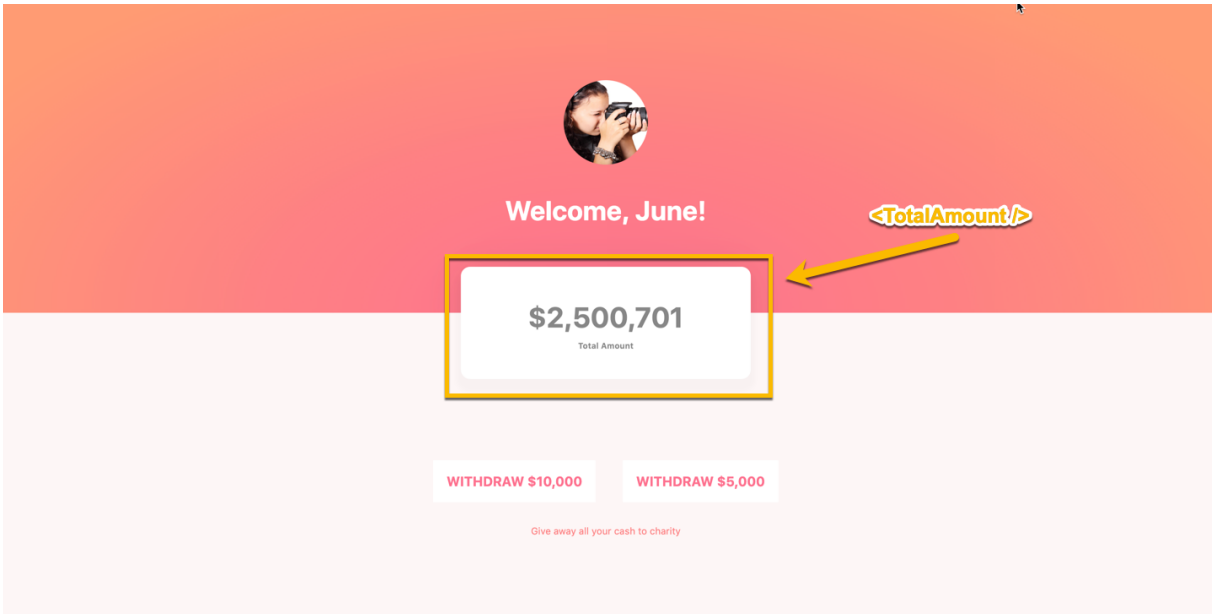
Also, `ViewAccountBalance` takes in a boolean prop `showBalance` which decides whether to show the account balance or not. This is toggled to `true` when you click the "yes" button.




```
//ViewAccountBalance.js


const ViewAccountBalance = ({ showBalance, loggedInUser,
displayBalance }) => {
  return (
    <Fragment>
      {!showBalance ? (
        <div>
          <p>
            Would you love to view your account balance?
          </p>
          <button onClick={displayBalance}>
            Yes
          </button>
        </div>
      ) : (
        <TotalAmount totalAmount={loggedInUser.totalAmount} />
      )}
    </Fragment>
  )
```

```
}
```

From the code block above, do you also see that `ViewAccountBalance` receives the `loggedInUser` prop only to pass it to `TotalAmount`?



`TotalAmount` receives this prop, retrieves the `totalAmount` from the user object and renders the total amount.

I'm pretty sure you can figure out whatever else is going on in the code snippets above.

Having explained the code so far, do you see the obvious props drilling here?

`loggedInUser` is passed down way too many times to components that don't even need to know about it.

Let's fix that with the `Context` API.

Avoid Props Drilling with Context.

One easy solution is to look at the `Root` component where we began passing props down and finding a way to introduce a context object in there.

Going by that solution, we could create a context object with no initial default values above the `Root` class declaration:

```
const { Provider, Consumer } = createContext()
class Root extends Component {
  state = {
    loggedInUser: null
  }

  ...
}
```

Then we could wrap the main `App` component around the `Provider` with a value prop.

```
class Root extends Component {
  state = {
    loggedInUser: null
  }
  ...
  render () {
    ...
    return loggedInUser ? (
      <Provider value={this.state.loggedInUser}>
        <App loggedInUser={loggedInUser} />
      </Provider>
    )
...
```

Initially, the `Provider value` prop will be `null`, but as soon as a user logs in and the `state` is updated in `Root`, the `Provider` will also receive the current `loggedInUser`.

With this done we can import the `Consumer` wherever we want and do away with passing props needlessly down the component tree.

For example here's the `Consumer` used in the `Greeting` component:

```
import { Consumer } from '../Root'
const Greeting = () => {
  return (
    <Consumer>
      {user => <p>Welcome, {user.name}! </p>}
    </Consumer>
  )
}
```

We could go ahead and do the same everywhere else we've passed the `loggedInUser` prop needlessly.

And the app works just as before, only we got rid of passing down `loggedInUser` over and over again .

Isolating Implementation Details.

The solution highlighted above works but not without some caveat.

A better solution will be to centralise the logic for the user state and Provider in one place.

This is pretty common practice. Let me show you what I mean.

Instead of having the `Root` component manage the state for `loggedInUser`, we will create a new file called `UserContext.js`.

This file will have the related logic for updating `loggedInUser` as well as expose a context `Provider` and `Consumer` to make sure `loggedInUser` and any updater functions are accessible from anywhere in the component tree.

This sort of modularity becomes important when you have many different context objects. For example, you could have a `ThemeContext` and `LanguageContext` object in the same app.

Extracting these into separate files and components proves more manageable and effective over time.

Consider the following:

```
// UserContext.js
import React, { createContext, Component } from 'react'
import { USER } from '../api'
const { Provider, Consumer } = createContext()
class UserProvider extends Component {
  state = {
    loggedInUser: null
  }
  handleLogin = evt => {
    evt.preventDefault()
    this.setState({
      loggedInUser: USER
    })
  }

  render () {
    const { loggedInUser } = this.state
    return (
      <Provider
        value={{
          user: loggedInUser,
          handleLogin: this.handleLogin
        }}
      >
```

```
        {this.props.children}
      </Provider>
    )
  }
}
export { UserProvider as default, Consumer as UserConsumer }
```

This represents the content of the new `context/UserContext.js` file. The logic previously handled by the `Root` component as been moved here.

Note how it handles every logic regarding the `loggedInUser` state value, and passes the needed values to `children` via a `Provider`.

```
...
<Provider
     value={{
       user: loggedInUser,
       handleLogin: this.handleLogin
      }}
     >
      {this.props.children}
</Provider>
...
```

In this case, the `value` prop is an object with the `user` value, and function to update it, `handleLogin`.

Also note that the Provider and Consumer are both exported. This makes it easy to consume the values from any components in the application.

```
export { UserProvider as default, Consumer as UserConsumer }
```

With this decoupled setup, you can use the `loggedInUser` state value anywhere in your component tree, and have it updated from anywhere in your component tree as well.

Here's an example of using this in the `Greeting` component:

```
import React from 'react'
import { UserConsumer } from '../context/UserContext'
const Greeting = () => {
  return (
    <UserConsumer>
      {({ user }) => <p>Welcome, {user.name}! </p>}
    </UserConsumer>
  )
}
export default Greeting
```

How easy.

Now, I've taken the effort to delete every reference to `loggedInUser` where the prop had to be passed down needlessly. Thanks, Context!

For example:

```
// before
const User = ({ loggedInUser, profilePic }) => {
  return (
    <div>
      <img src={profilePic} alt='user' />
      <Greeting loggedInUser={loggedInUser} />
    </div>
  )
}
// after: Greeting consumes UserContext
const User = ({profilePic }) => {
```

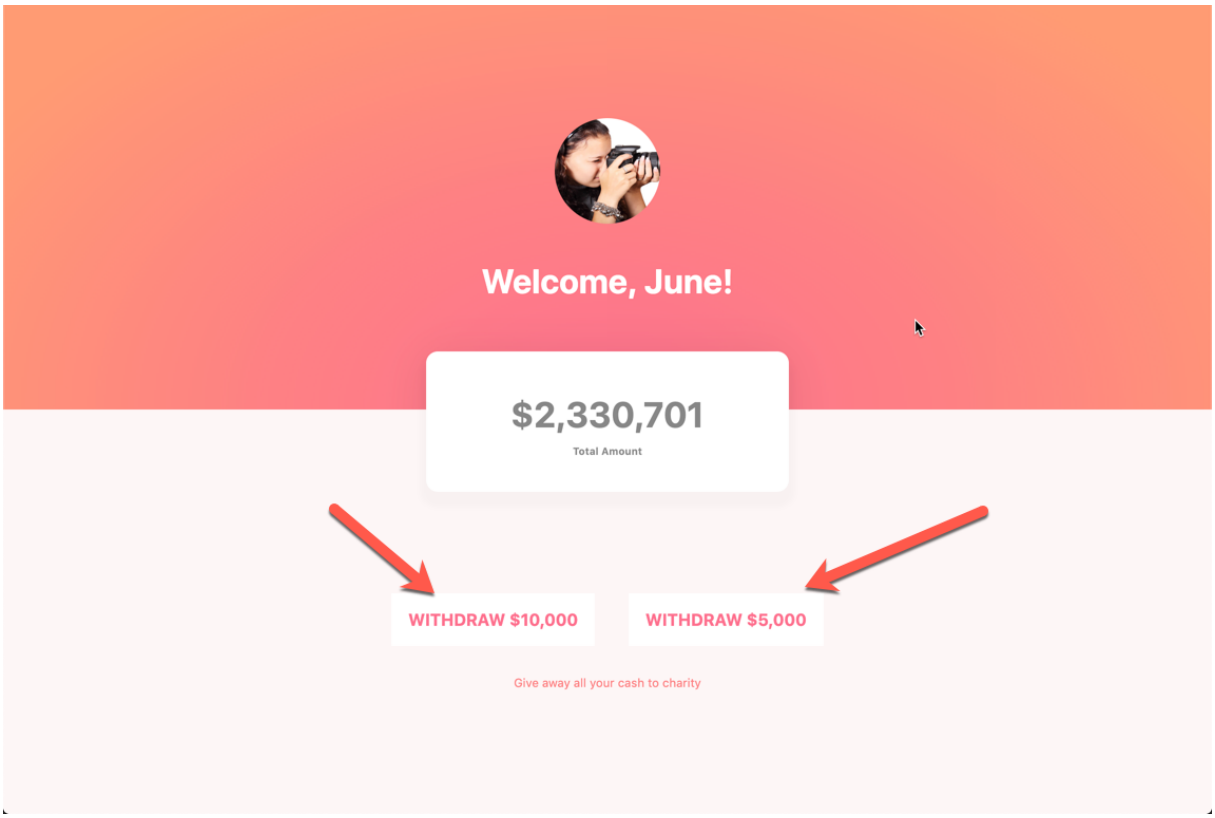
```
  return (
    <div>
      <img src={profilePic} alt='user' />
      <Greeting />
    </div>
  )
}
export default User
```

Be sure to look in the accompanying code folder for the final implementation I.e after stripping off the `loggedInUser` from being passed down needlessly.

Updating Context Values.

What's a bank app if you can't make withdrawals, huh?

Well, this app has some buttons. You click them and voila, a withdrawal is made.



Since the `totalAmount` value resides in the `loggedInUser` object, we may as well have the logic to make withdrawals in the `UserContext.js` file.

Remember we're trying to centralise all logic related to the user object in one place.

To do this, we'll extend the `UserProvider` in `UserContext.js` to include a `handleWithdrawal` method.

```
// UserContext.js
...
handleWithdrawal = evt => {
    const { name, totalAmount } = this.state.loggedInUser
    const withdrawalAmount = evt.target.dataset.amount

    this.setState({
      loggedInUser: {
        name,
```

```
        totalAmount: totalAmount - withdrawalAmount
      }
    })
  }
```

When you click any of the buttons, we will invoke this `handleWithdrawal` method.

From the `evt` click object passed as an argument to the `handleWithdrawal` method, we then pull out the amount to be withdrawn from the `dataset` object.

```
const withdrawalAmount = evt.target.dataset.amount
```

This is possible because both buttons have a `data-amount` attribute set on them. For example:

```
<button data-amount=1000 />
```

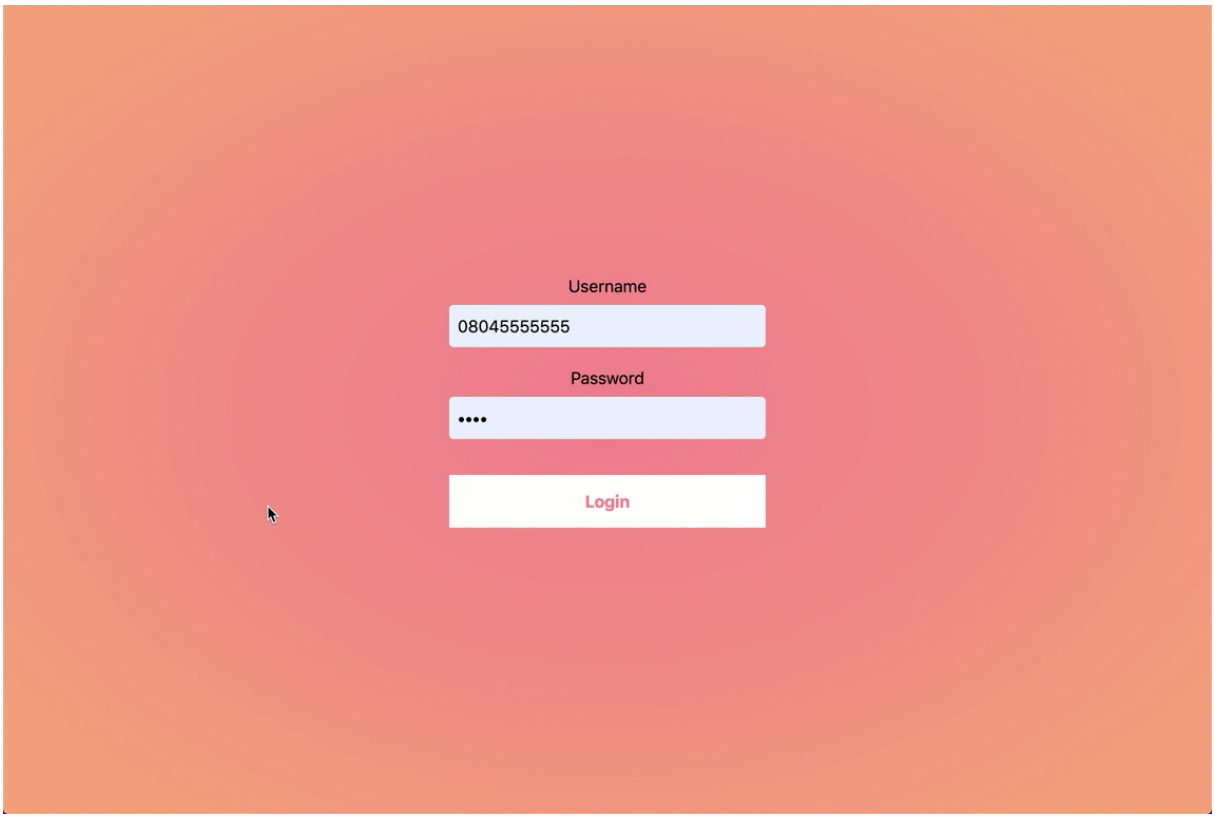
Now that we have the `handleWithdrawal` method written out, we can expose it via the `values` prop passed to `Provider` as shown below:

```
<Provider
    value={{
       user: loggedInUser,
       handleLogin: this.handleLogin
       handleWithdrawal: this.handleWithdrawal
     }}
   >
 {this.props.children}
</Provider>
```

Now, we're all set to consume the `handleWithdrawal` method from anywhere in the component tree.

In this case, our focus is on the `WithdrawButton` component. Go ahead and wrap that in a `UserConsumer`, deconstruct the `handleWithdrawal` method and make it the click handler for the buttons as shown below:

```
const WithdrawButton = ({ amount }) => {
  return (
    <UserConsumer>
      {({ handleWithdrawal }) => (
        <button
          data-amount={amount}
          onClick={handleWithdrawal}
        >
          WITHDRAW {amount}
        </button>
      )}
    </UserConsumer>
  )
}
```



And that works!

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>
}
```

When you have multiple `Consumers` within a component, it results to having a lot of nested, not-so-pleasant code.

Here's an example.

While working on the *Benny Home Run* application, John had to create a new context object to hold the game level state of the current user.

```
// initial context object
const BennyPositionContext = createContext({
    x: 50,
    y: 50
})

// another context object for game level i.e Level 0 - 5
const GameLevelContext = createContext(0)
```

Remember, it's common practice to split related data into different context objects for reusability and performance (owing to the fact the every consumer is re-rendered when values within a `Provider` change)

With these context objects, John goes ahead to use both `Consumer` components within the `Benny` component as follows.

```
//grab consumer for PositionContext
const { Consumer: PositionConsumer } = BennyPositionContext

// grab consumer for GameLevelContext
const { Consumer: GameLevelConsumer } = GameLevelContext

// use both Consumers here.
const Benny = () => {
  return <PositionConsumer>
    {position => <GameLevelConsumer>
        {gameLevel => <svg />}
    </GameLevelConsumer>}
  </PositionConsumer>
}
```

Do you notice that consuming values from both context objects result in very nested code?

Well, this is one of the more common problem with consuming data with the `Consumer` component. With multiple consumer components, you begin to have a lot of nesting.

So, what can we do about this?

Firstly, when we learn about Hooks in a later chapter, you'll come to see an almost perfect solution to this. In the mean time, let's consider the solution available to `Class` components via something called`contextType`.

Using a Class Component with contextType.

To take advantage of `contextType` you're required to work with a class component.

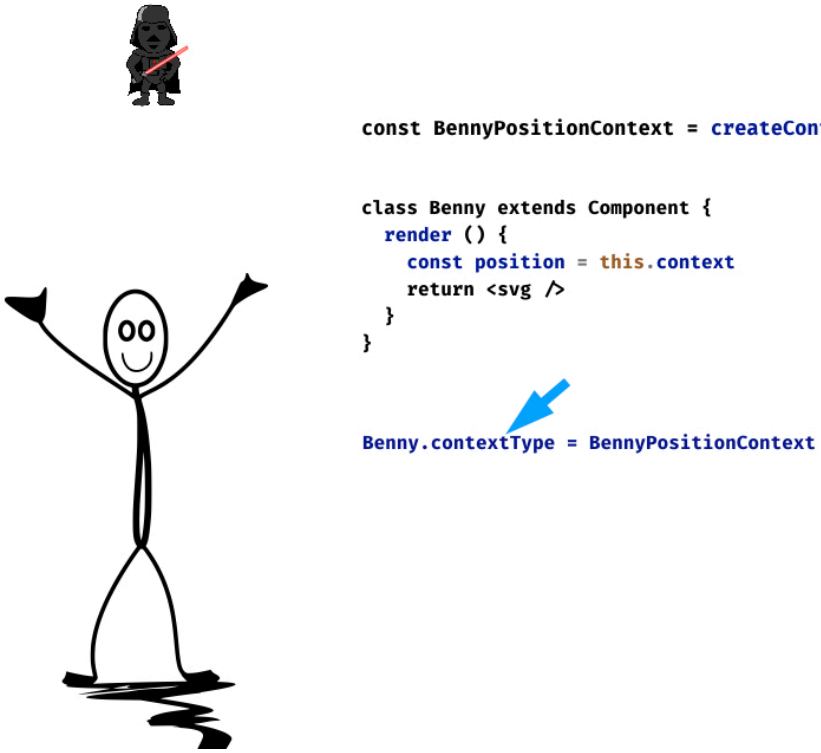
Consider the `Benny` component rewritten as a class component.

```
// create context object
const { Provider, Consumer } = createContext({ x: 50, y: 50 })
// Class component
class Benny extends Component {
  render () {
    return <Consumer>
    {position => <svg />}
  </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.



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 ...
// look here 👇
Benny.contextType = BennyPositionContext
```

After setting the `contextType` property, you can go ahead to consume values from the context object by using `this.context`.

For example, to retrieve the position values `{ x: 50, y: 50 }`:

```
class Benny extends Component {
  render () {
```

```
    // look here. No nesting!
    const position = this.context
    return <svg />
  }
}
```

The Perfect Solution?

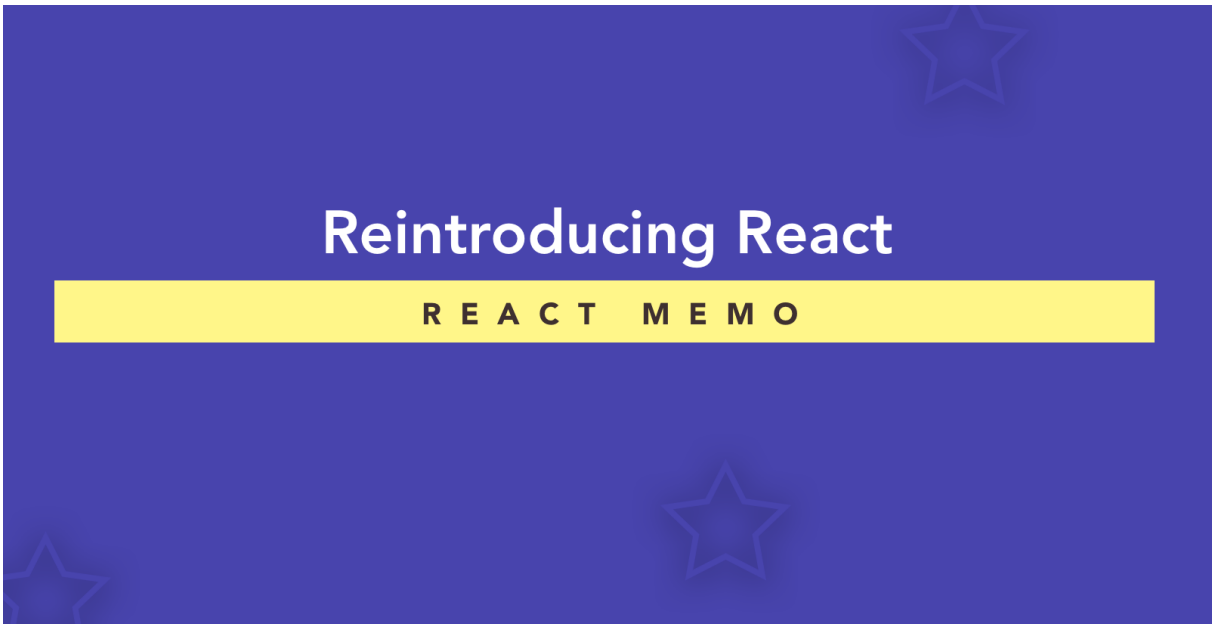
Using the `contextType` class property is great, but not particularly the best solution in the world. You can only use one `contextType` within a class component. This means if you need to introduce multiple `Consumers` you'll still have some nested code.

Conclusion

The `contextType` property is does solve the nesting problem a little bit.

However, when we discuss Hooks in a later chapter, you'll see how much better the solution hooks offer is.

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



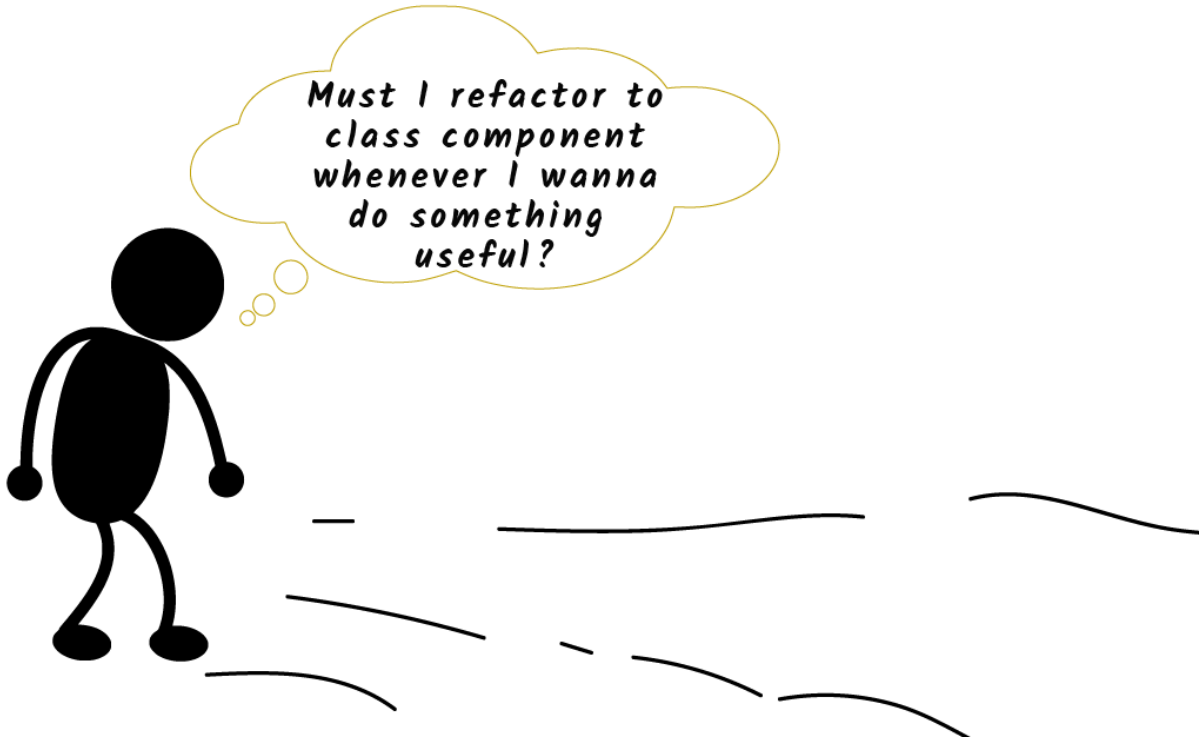
A few weeks ago John refactored the `Benny` component to a `PureComponent`.

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`.



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>
        Hello {name}.
     </div>
    )
}
export default MyComponent
// after
import React, { memo } from 'react'
export default  memo(function MyComponent ({name}) {
    return ( <div>
        Hello {name}.
     </div>
    )
})
```

So simple, it couldn't get any easier.

Technically, `React.memo` is a higher order function that takes in a functional component and returns a new memoized component.

So, if props do not change, `react` will skip rendering the component and just use the previously memoized value.

With this new found information, John refactors the functional component, `Benny` to use `React.memo`.

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

- class Benny extends PureComponent {
-   render () {

+ const Benny = memo(() => {
   return <Consumer>
     {position => <svg />}
   </Consumer>
})
```

Handling Deeply Nested Props.

`React.memo` does a props shallow comparison. By implication, if you have nested props objects, the comparison will fail.

To handle such cases, `React.memo` takes in a second argument, an equality check function.

Here's a basic example:

```
import React, { memo } from 'react'
export default  memo (function MyComponent (props) {
    return ( <div>
        Hello World from {props.name.surname.short}
     </div>
    )
}, equalityCheck)
function equalityCheck(prevProps, nextProps) {
  // return perform equality check & return true || false
}
```

If the `equalityCheck` function returns `true`, no re-render will happen. This would mean that the current props and previous props are the same. If you return `false`, then a re-render will occur.

If you're concerned about incurring extra performance hits from doing a deep comparison, you may use the lodash isEqual utility method.

```
import { isEqual } from 'lodash'

function equalityCheck(prevProps, nextProps) {
    return isEqual(prevProps, nextProps)
}
```

Conclusion.

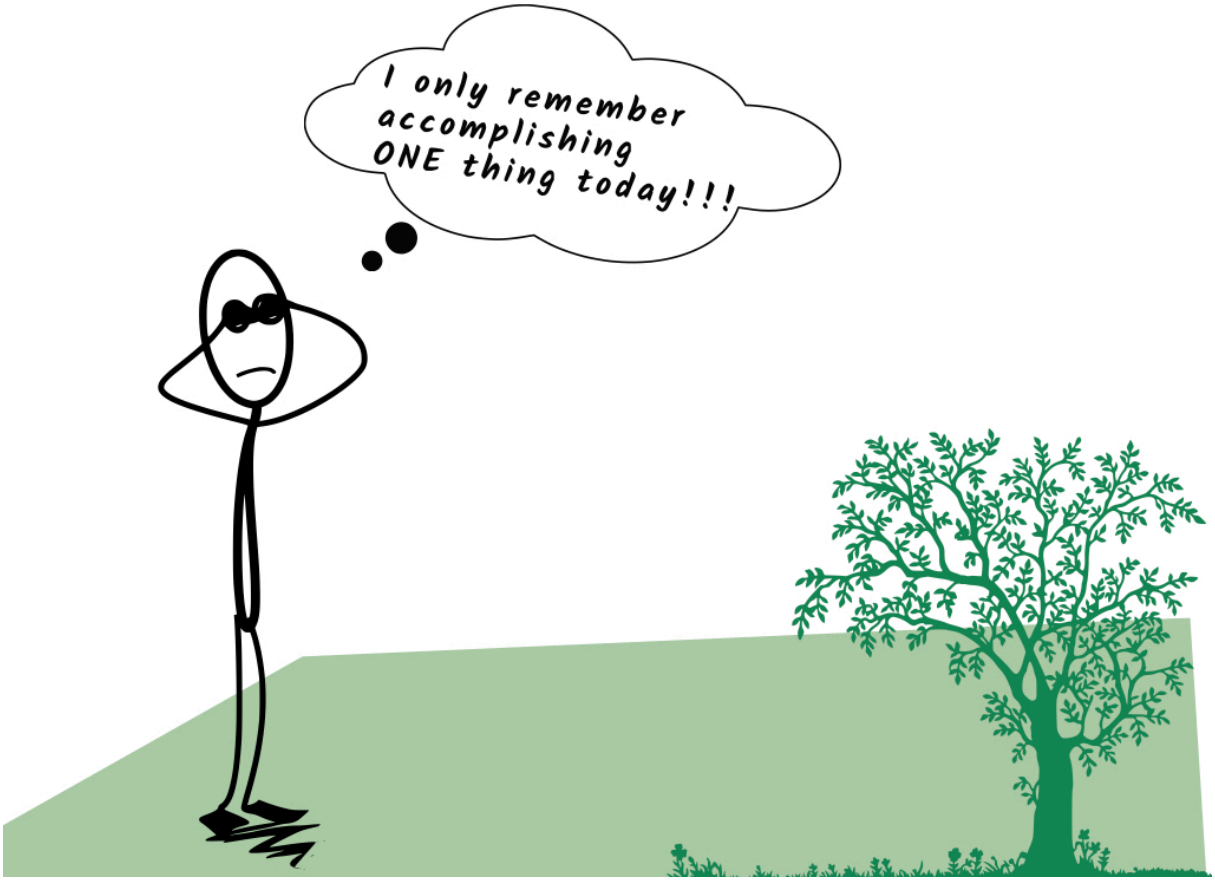
`React.memo` brings the class `PureComponent` behaviour to functional components. We've also had a look at using the `equalityCheck` function as well. If you use the `equalityCheck` function, be sure to include checks for function props where applicable. Not doing so is a common source of bugs in many React applications.

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.

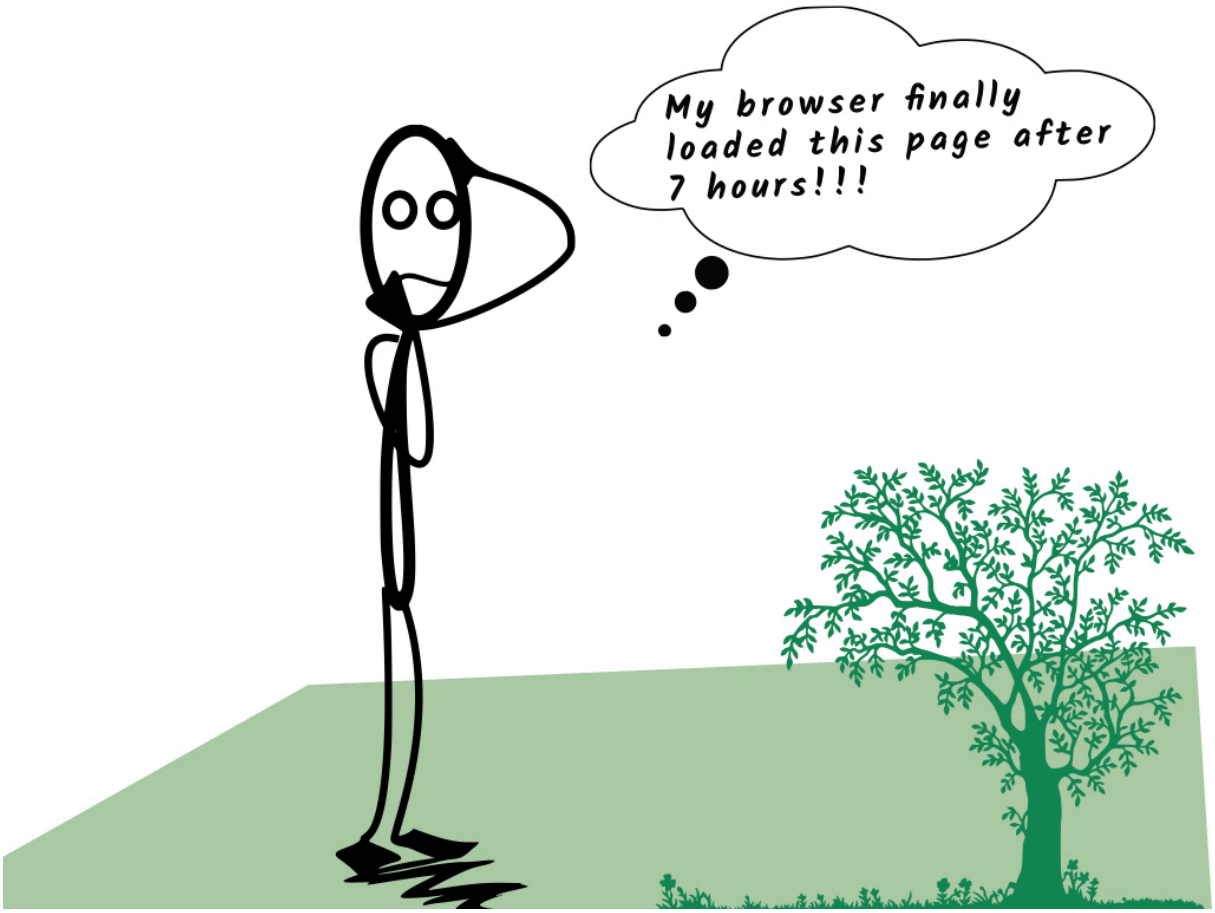
After a long careful thought, she comes to the conclusion that she accomplished just one thing the entire day.



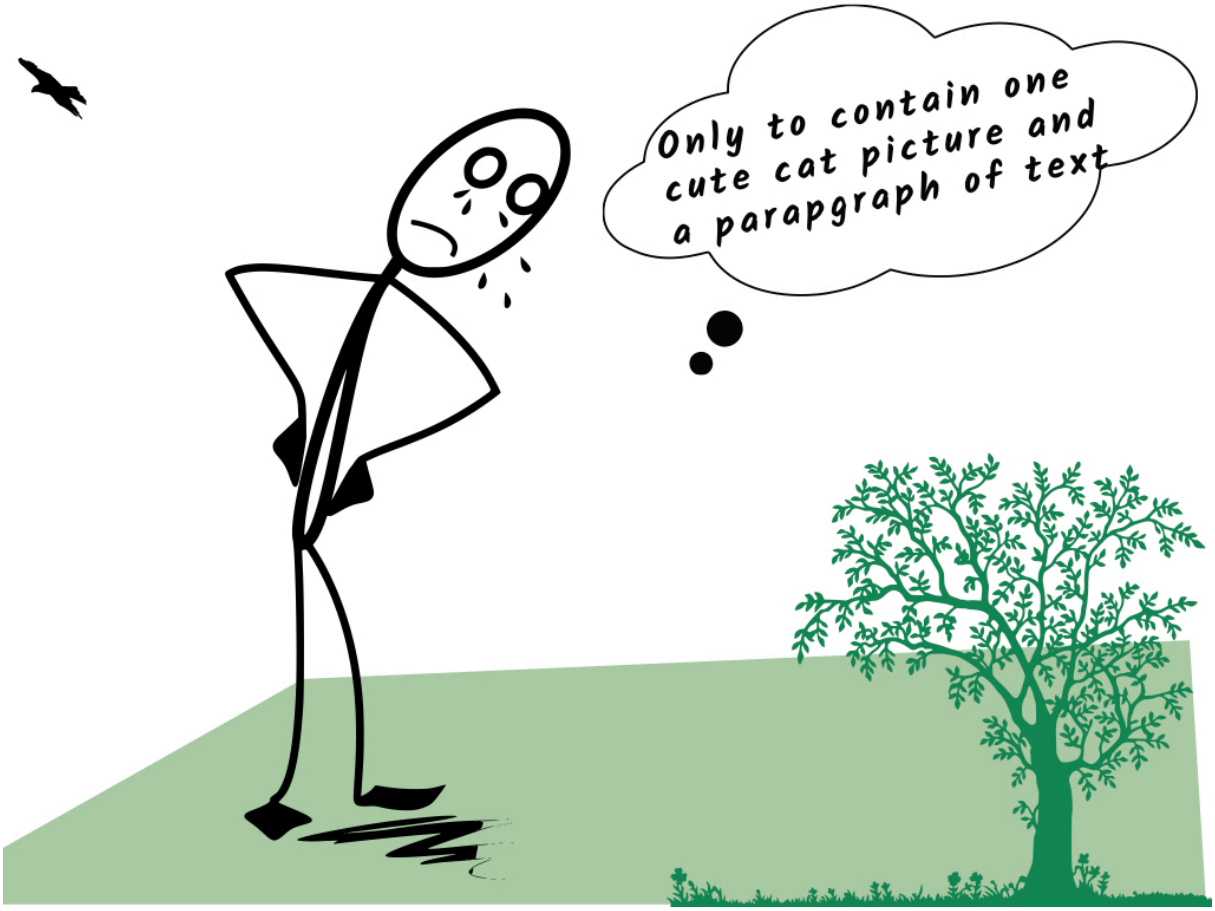
Well, how is it possible that Mia only achieved one thing the entire day?

That 'one thing' had better be a worthy feat!

So, what was Mia's accomplishment for the day?



It turns out that all Mia accomplished today was seating helplessly as her browser attempted to load a page for 7 hours!



What???

Measuring Performance in React Apps.

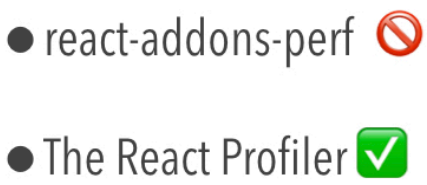
Web performance is a big deal. Nobody has all the time in the world to seat through minutes waiting for your webpage to load.

In order to measure performance and identify performance bottlenecks in your apps, it's crucial to have some way to inspect how long it took your app's components to render, and why they were rendered.

This is exactly what the Profiler exists for.

If you've been writing react for sometime, you might remember the `react-addons-perf` module.

Well, that has been deprecated in favour of the Profiler.



With the Profiler, you can :

- Collect timing information about each component
- Easily identify performance bottlenecks
- Be sure to have a tool compatible with concurrent rendering

Getting Started.

To keep this as pragmatic as possible, I have set up a tiny application we're going to profile together. i.e measure performance. We'll do this with the aid of the Profiler.

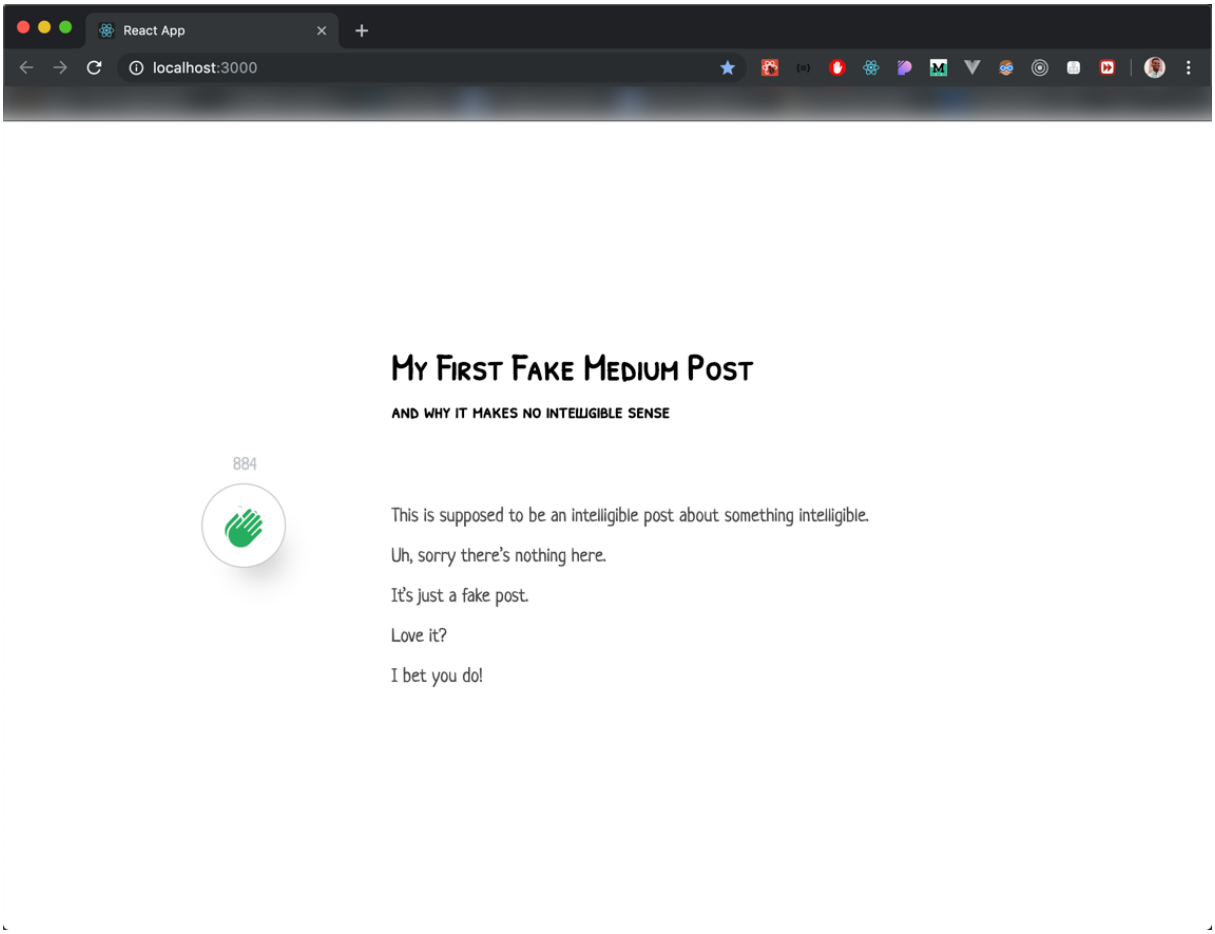
I call the application *fake-medium*, and it looks like this:

You'll find the source code for the application in the code repository for this book.

To Install the dependencies and run the app, run the following from the`04-The-Profiler` directory:

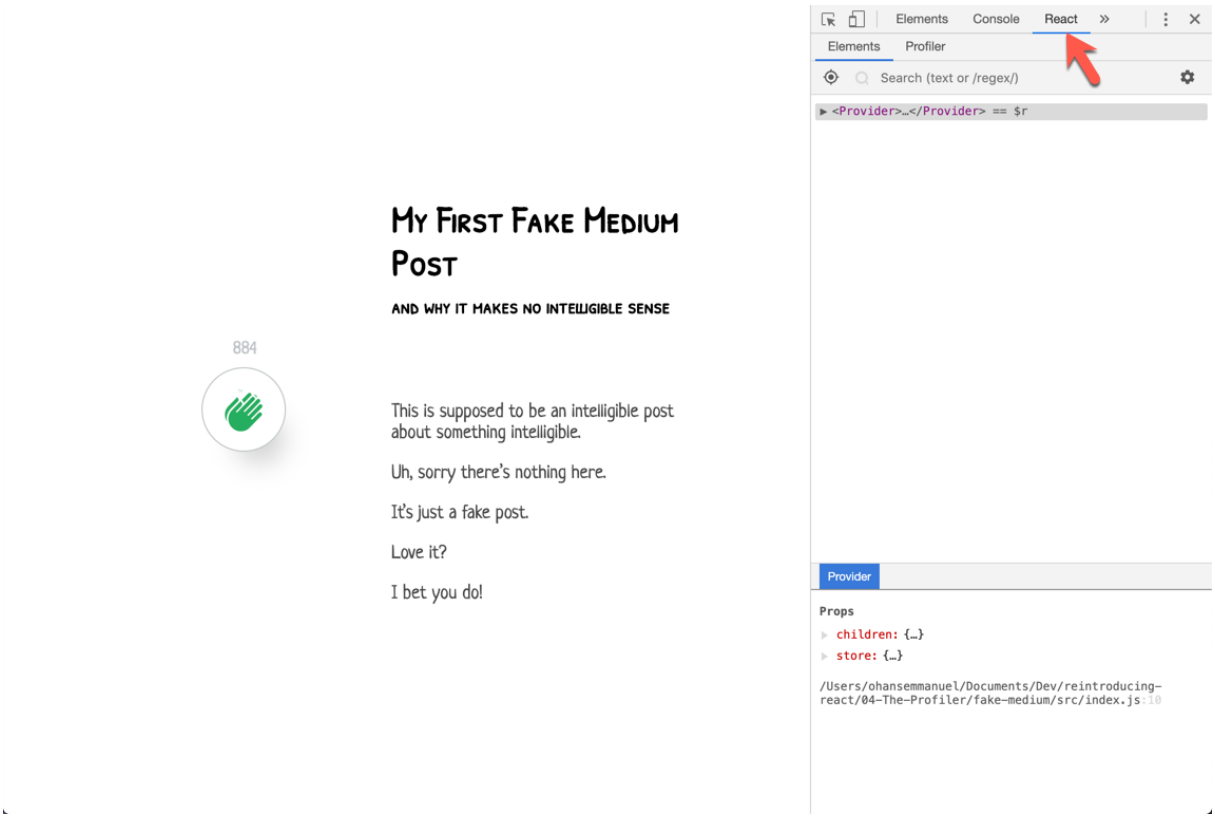
```
cd fake-medium
yarn install
yarn start
```

If you ran those commands, you should have the application running in your default browser, on port `3000` or similar.



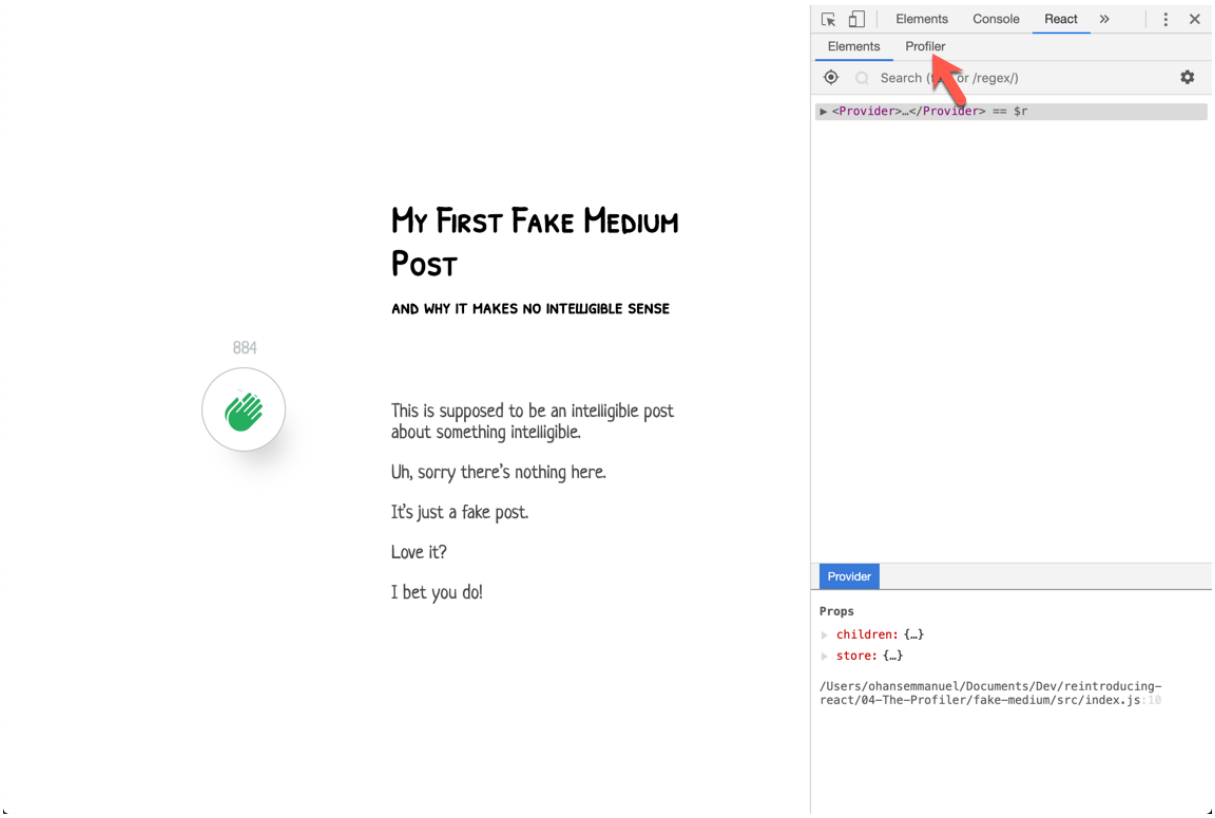
Finally, open your chrome devtools by pressing Command+Option+J (Mac) or Control+Shift+J (Windows, Linux, and Chrome OS).

Then find the React chrome devtools extension tab and click it.

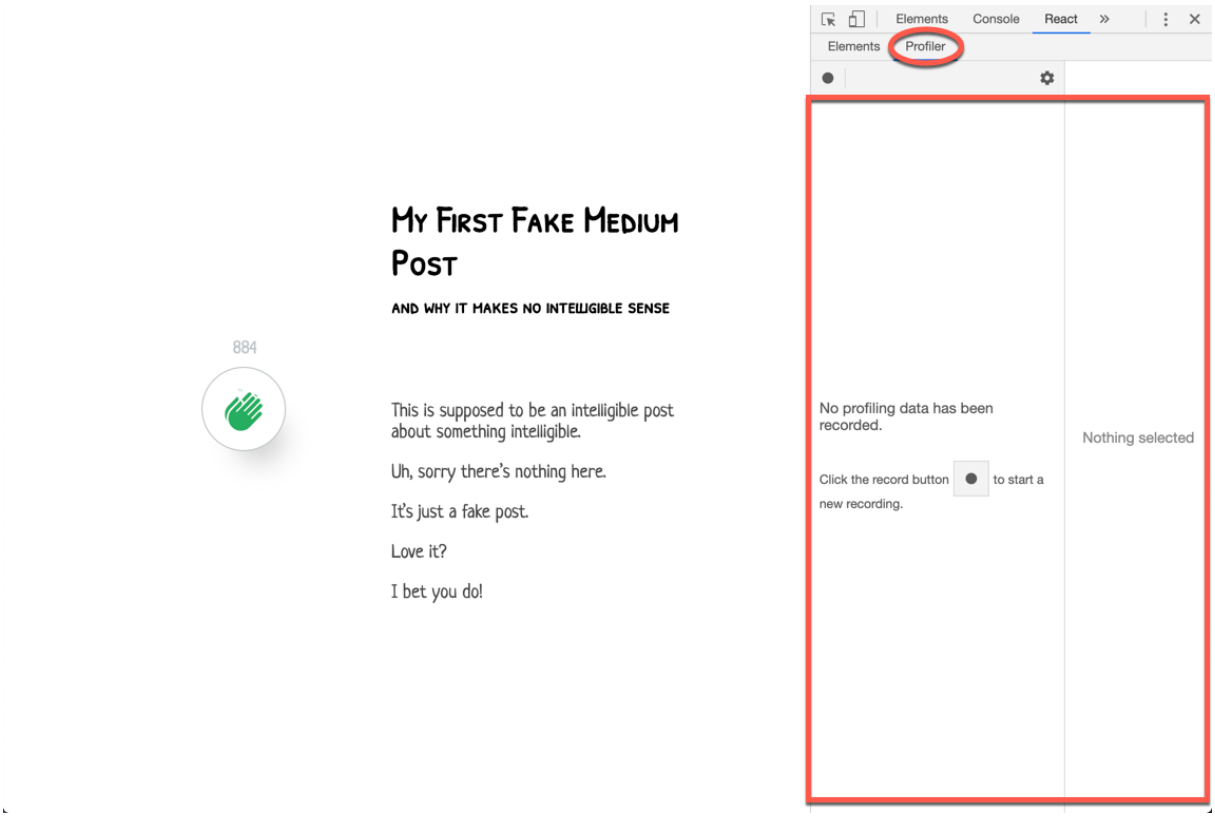


You'll be presented with two tabs, elements and profiler.

You guessed right, our focus is on the profiler tab, so please click it.



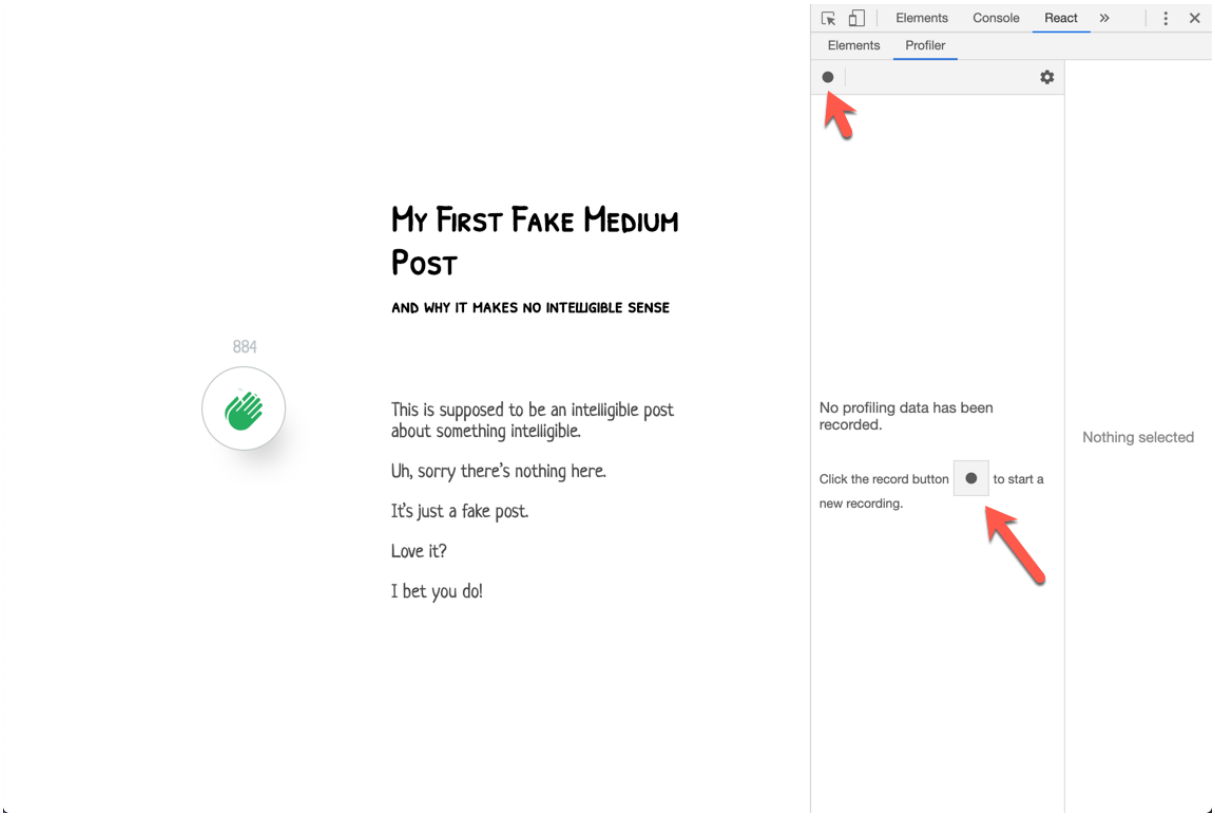
Doing so will lead you to the following page:



How does the Profiler Work?

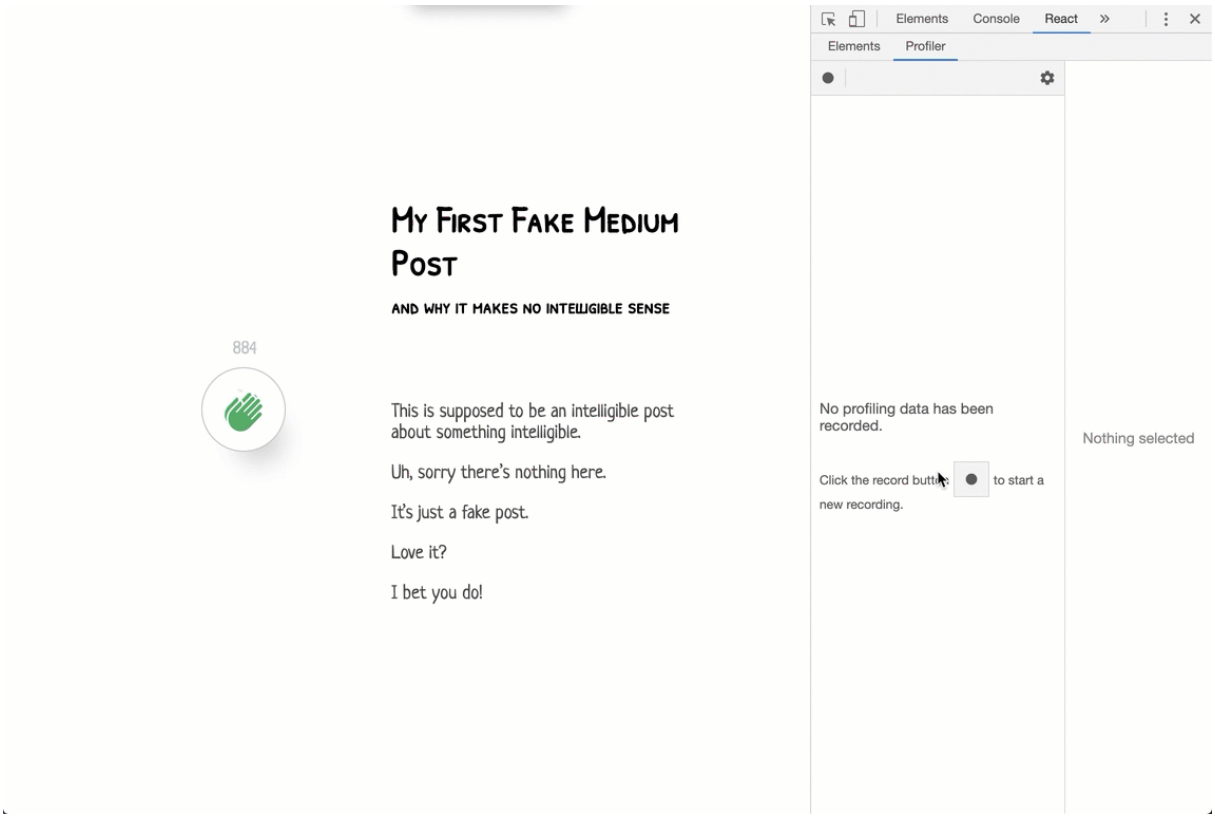
The Profiler works by recording a session of actual usage of your application. In this recording session it gathers information about the components in your application and displays some interesting information you can exploit to find performance bottlenecks.

To get started, click the record button.



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

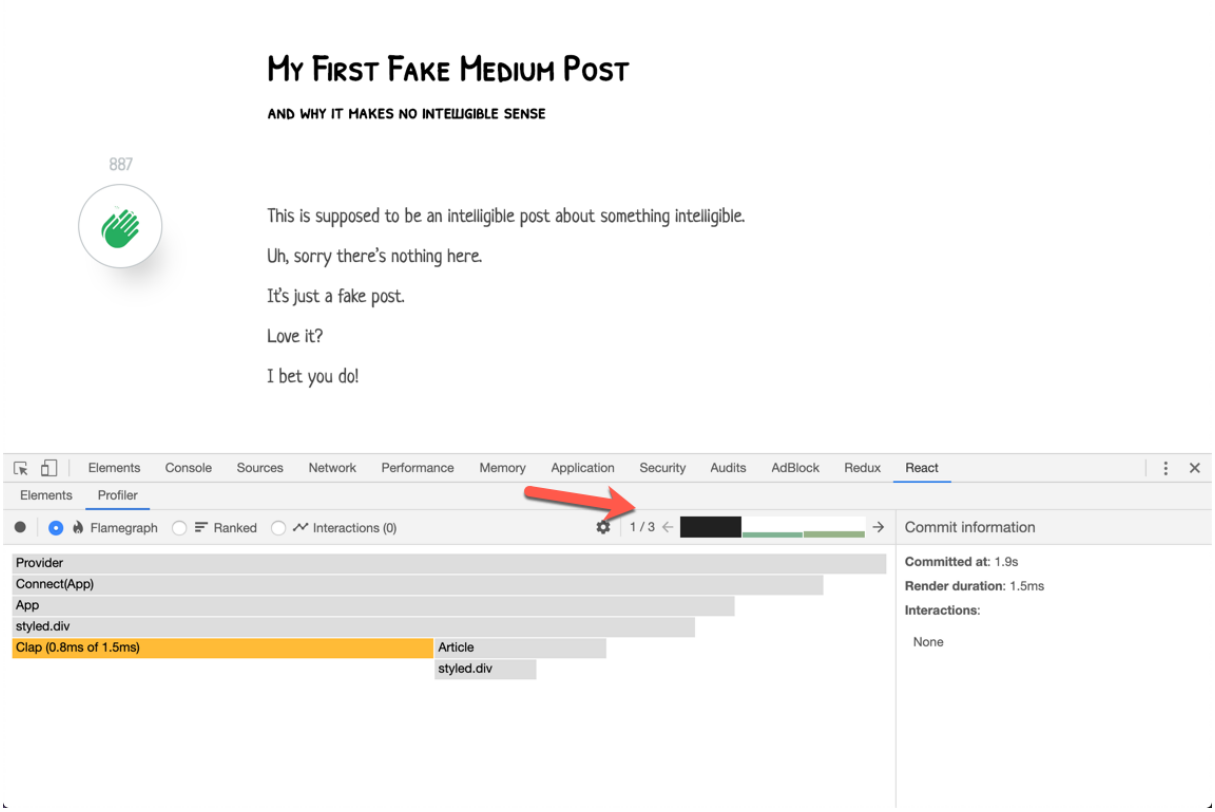
In this case, I've gone ahead to click the medium clap button 3 times!



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

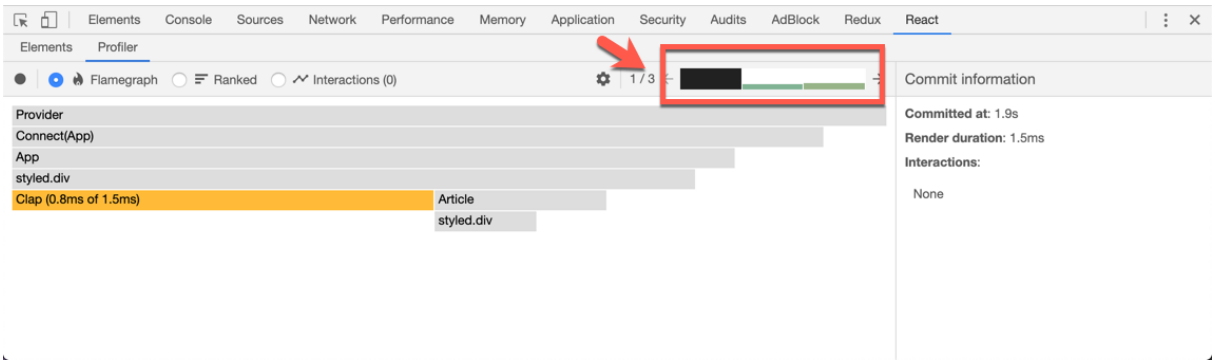
Making Sense of the Profiler Results.

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.



Conceptually, react does work in 2 phases. The render phase where components are rendered and the virtual DOM *diffed*, and the commit phase where actual changes in the virtual DOM are committed to the DOM.

The graphical representation you see on the far right of the profiler represents the number of commits that were made to the DOM during your interaction with the app.

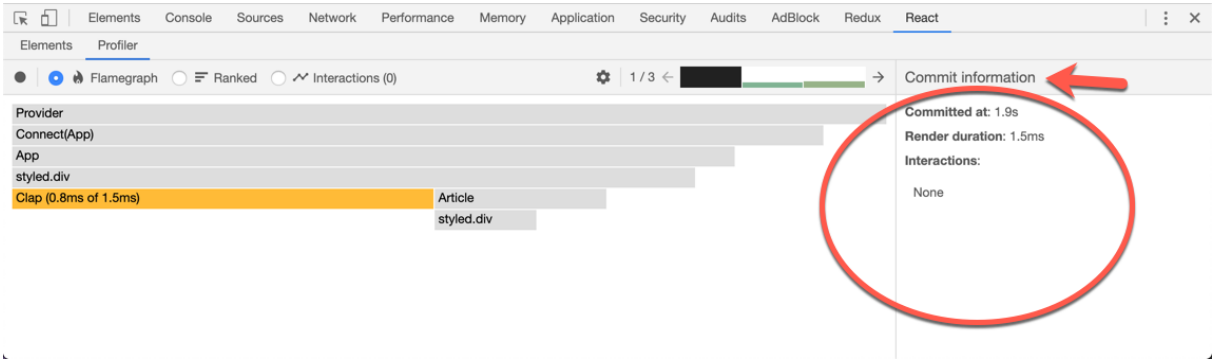


The taller the bar is, the longer it took `React` to render the components in this commit.

In the example above, the Profiler recorded three commits. That make sense since I clicked the button only 3 times. So, there should be only 3 commits made to the DOM.

Also the first commit took much longer than the subsequent two commits.

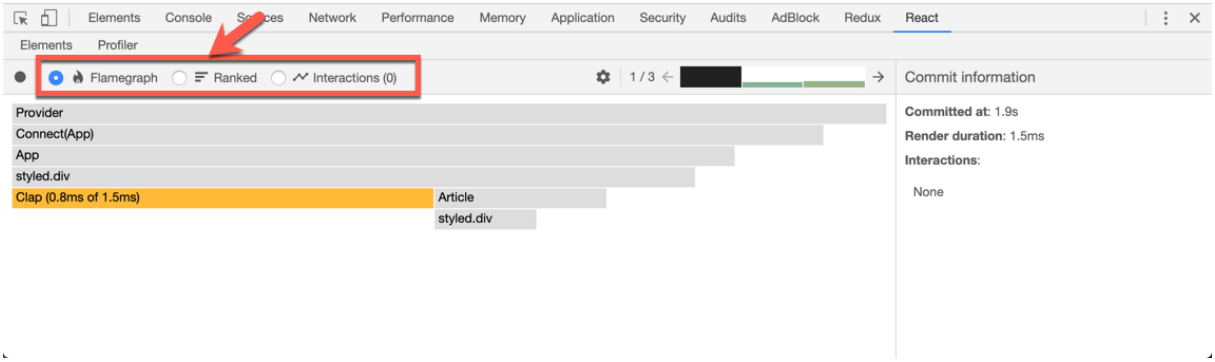
The three bars represent the different commits made to the DOM, and you can click on any to investigate performance metrics for the particular commit.



The Flame Chart.

After a successful recording session, you'll be presented with a couple different information about your components.

First, you have 3 tabs representing different groups of information - each relating to the selected commit on the right.



The first tab represents a flame chart.

The flame chart displays information on how long it took your component tree to render.

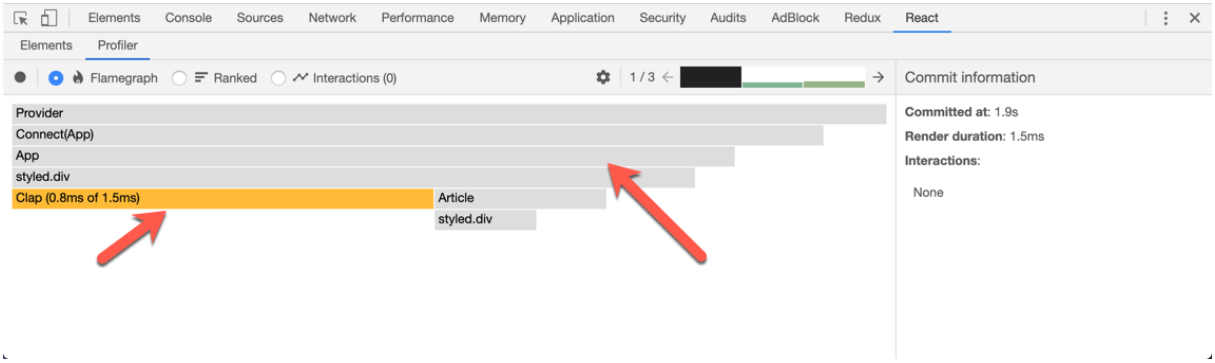
You'll notice that each component in your application tree is represented by bars of varying lengths and colors.

The length of a bar defines how long it took the component (and its children) to render.

Judging by the bar length, it appears the `Provider` component took the longest time to render. That make sense since the `Provider` is the main root component of the app, so the time represented here is the time taken for `Provider` and all its children to render.

That's half the story.

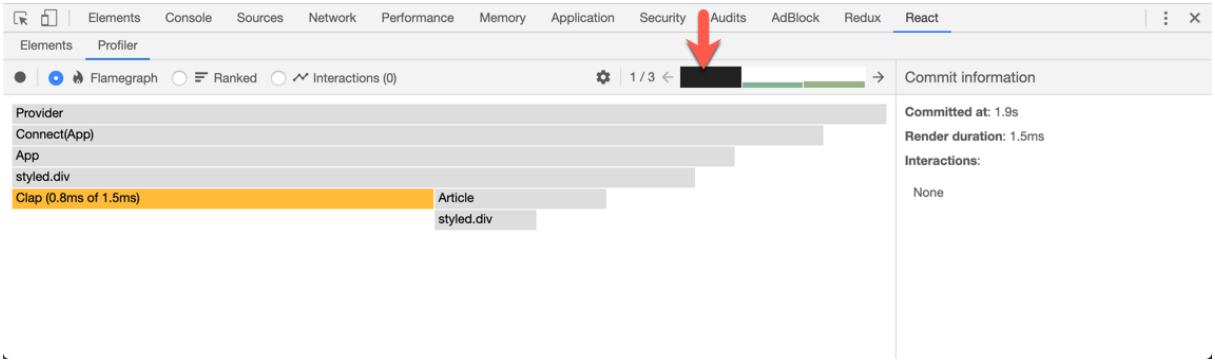
Note that the colors of the bars are different.



For example, `Provider` and a couple other components have a *grey* color.

What does that mean?

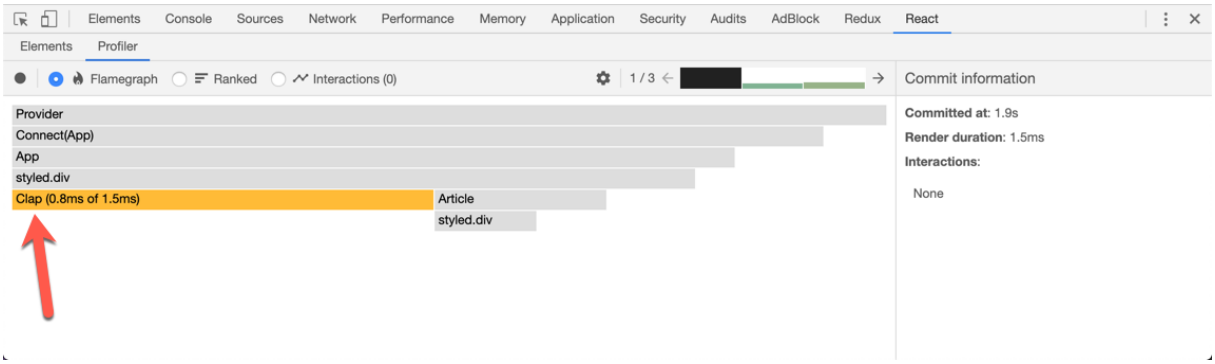
Well, first we are investing the first commit made to the DOM during the interaction with the application.



The components with a grey color means they weren't rendered in this commit. If the component is greyed out, it wasn't rendered in this commit. So, the length of the bar only represents how long it took the component to render *previously* before this commit i.e before the interaction with the application.

If you think about it, that's reasonable.

On careful examination, you'll see that the only component with a different flame chart color here is the `Clap` component.



This component represents the Medium clap button that was clicked.

A yellow bar means the component took the most time to render in this commit.

Well, no other component is coloured which means the `Clap` button was the only component re-rendered in this commit.

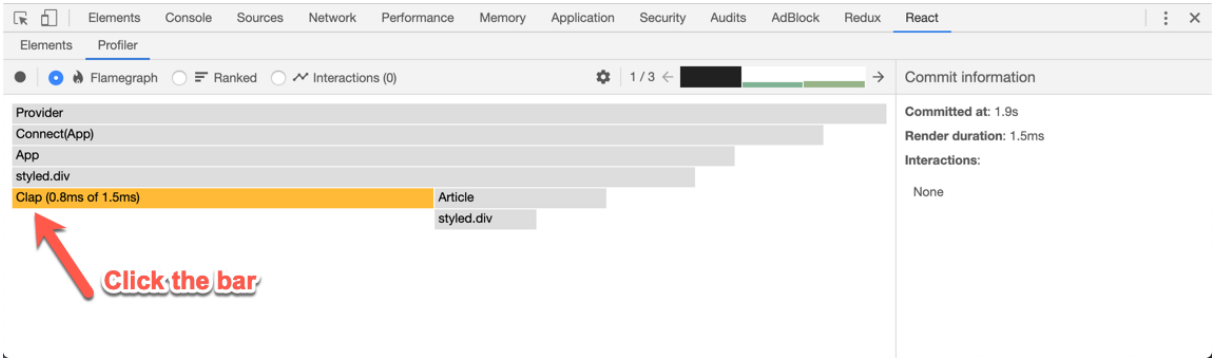
That's perfect!

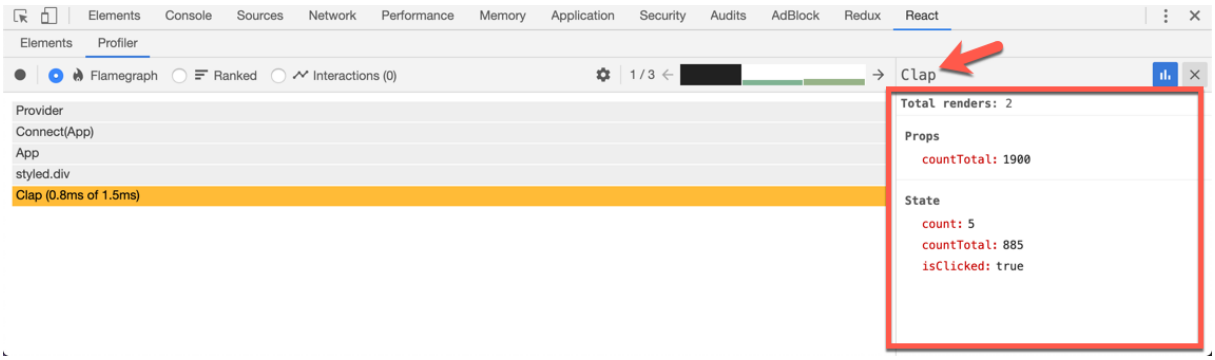
You don't want to click the `Clap` button and have a different component being re-rendered. That'll be a performance hit right there.

In more complex applications, you'll find flame charts with not just yellow and grey bars. You'll find some with blue bars.

What's worth noting is that, yellow longer bars took more time to render, followed by the blue ones, and finally grey bars weren't re-rendered in the particular commit being viewed.

It's also possible to click on a particular bar to view more information on why it rendered or not i.e the props and state passed to the component.



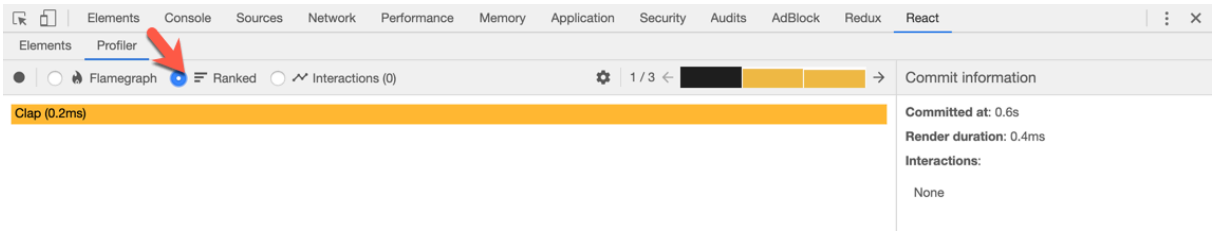


While zoomed in, you can also click the commit bars on top to see the difference in `props` or `state` across each commit render.

The Ranked Chart.

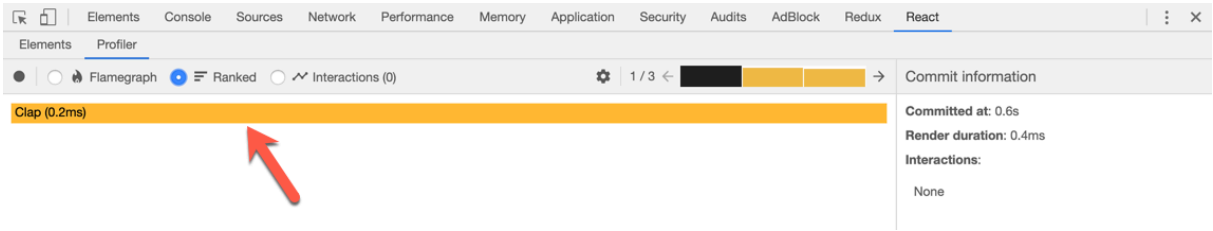
Once you understand how the flame chart works, the ranked chart becomes a walk over.

The second tab option refers to the ranked chart.



The ranked chart displays every component that was rendered in the commit being viewed. It displays this components ranked from top to bottom — with component which took more time to render at the top.

In this example, we have just the `Clap` component displayed in the ranked chart view. That's okay as we only expect the `Clap` component to be re-rendered upon clicking.



A more complex ranked chart may look like this:

You can see how the longer yellow bars are right there at the top, and shorter blue bars at the bottom. If you look carefully, you'll notice that the colors fade as you go

from top to bottom. From more yellow bars to pale yellow bars, to light blue bars and finally blue bars.

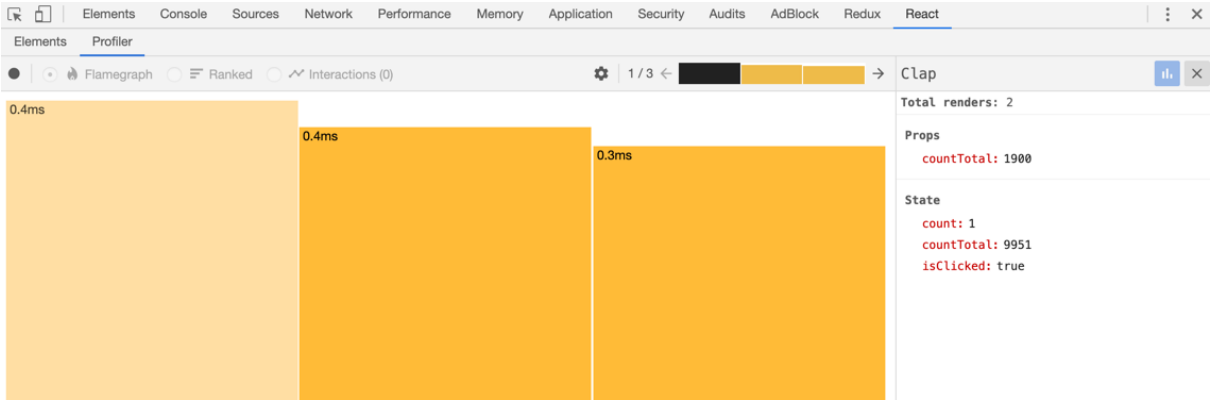
Component Chart.

Whenever you zoom into a component within the Profiler i.e by clicking its associated bar, a new option on the right pops up.

Clicking this button will bring you to what's called the component chart.

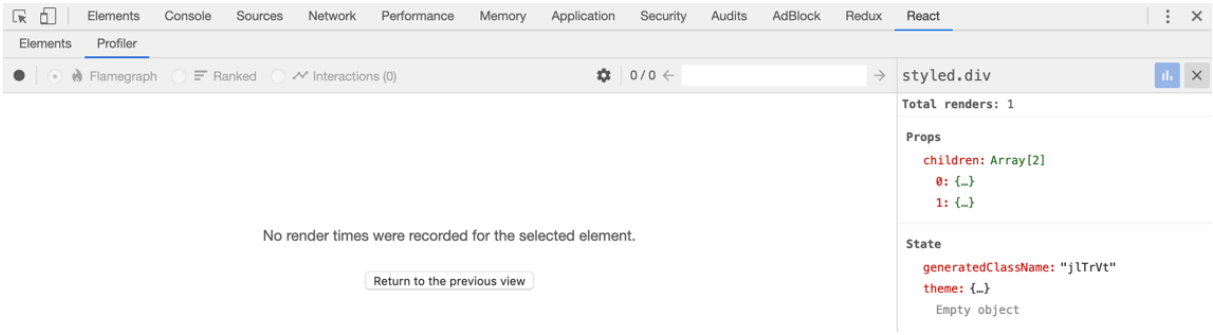
The component chart displays how many times a particular component was re-rendered during the recorded interaction.

In this example, I can click the button to view chart for the `Clap` component.



This shows three bars representing the three times the `Clap` component was re-rendered. If I saw a fourth bar here, I'd begin to get worried as I only clicked three times and expected only three re-renders.

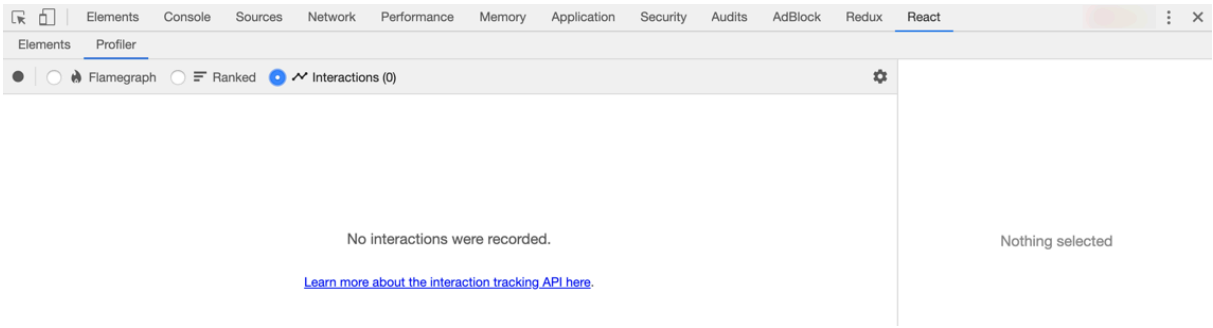
If the selected component didn't re-render at all, you'll be presented with the empty screen below:



NB: you can either view the component chart by clicking the button on the far right, or by double clicking a component bar from the flame or ranked chart.

Interactions.

There's one final tab in the profiler, and by default it displays an empty screen:



Interactions let you tag actions performed during a recording session with a string identifier so they can be monitored and tracked within the profiling results.

The API for this is unstable, but here's how to enable interactions in your profiling results.

First install the `scheduler` module. From your terminal, run `yarn add scheduler` within the application directory .

Once the installation is done, you need to use `unstable_trace` function exported by the module as shown below:

```
import { unstable_trace as trace } from 'scheduler/tracing'
```

The function is exported as `unstable_trace` but you can rename it in the import statement as I have done above.

The `trace` function has a signature similar to this:

```
trace("string identifier", timestamp, () = {
})
```

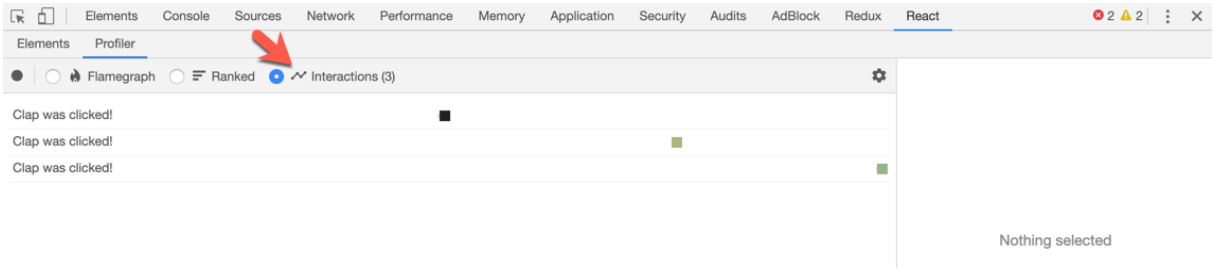
It takes a string identifier, timestamp and a callback function. Ideally, you track whatever interaction you want by passing it into the callback.

For example, I have gone ahead to do this in the `fake-medium` application:

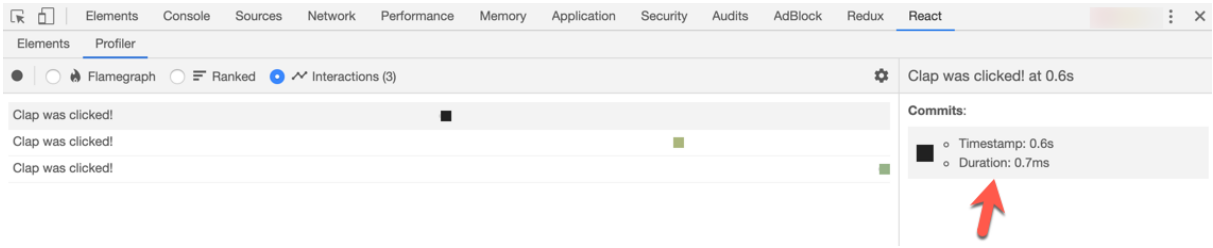
```
// before
_handleClick () {
   // do something
}
// after
_handleClick () {
   trace("Clap was clicked!", window.performace.now(), () => {
      // do something
   })
}
```

The medium clap when clicked calls the `_handleClick` method. Now, I've wrapped the functionality within the `trace` method.

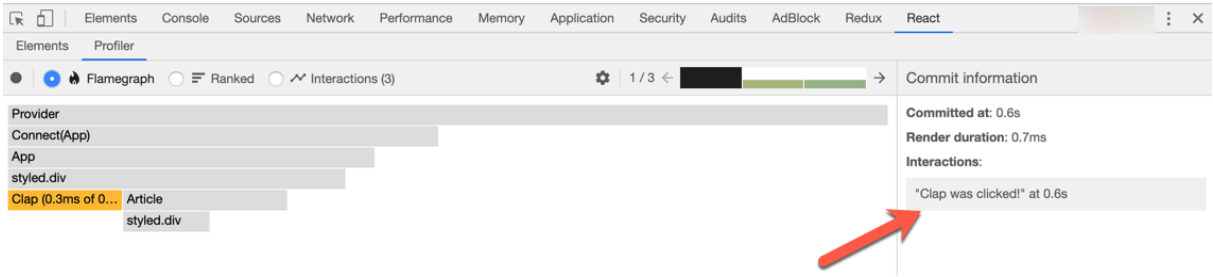
Here's what happens when the profiling result is now viewed:



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



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



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.

"*Okay, I'll have a look and fix whatever bottlenecks I find. Can we do that together?*", John said while thinking to himself how much help Mia would be since she spotted the problem anyway.

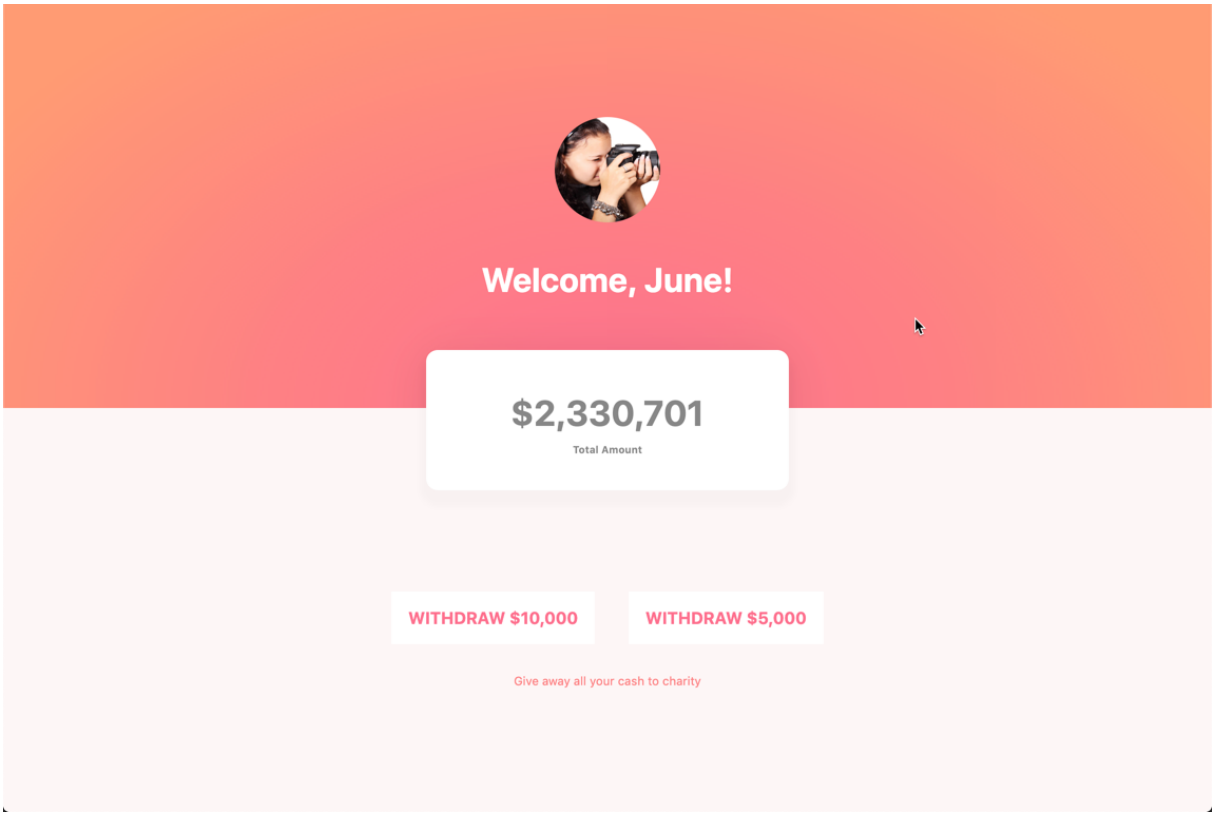
"*Oh, sure*", she retorted.

After spending a couple hours, they found and fixed a coupe of performance bottlenecks in the application.

What did they do? What measures were taken?

In this example, we'll spin up the bank application and pick up from where we stopped in the earlier chapter. This time we'll fix the performance bottlenecks within the application.

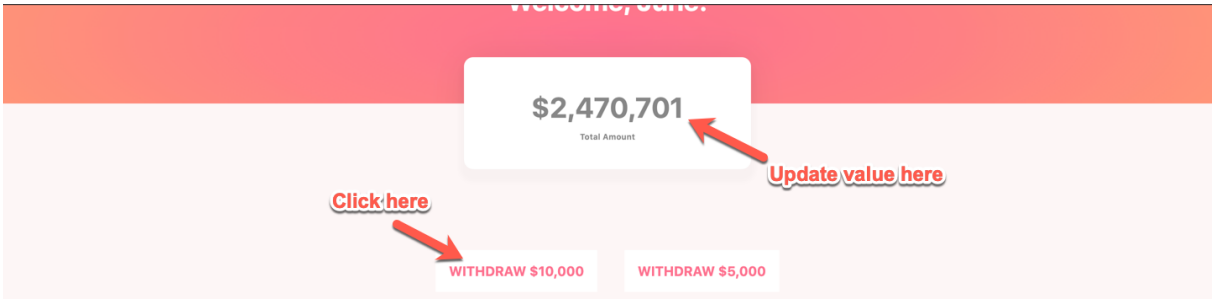
Here's what the bank app looks like again:



Noting the Expected Behaviour.

When I need to profile an application, specifically during a certain interaction with an app, I like to set the baseline for what I expect in terms of performance. This sorts of expectation helps you retain your focus as you delve into interpreting the results from the Profiler.

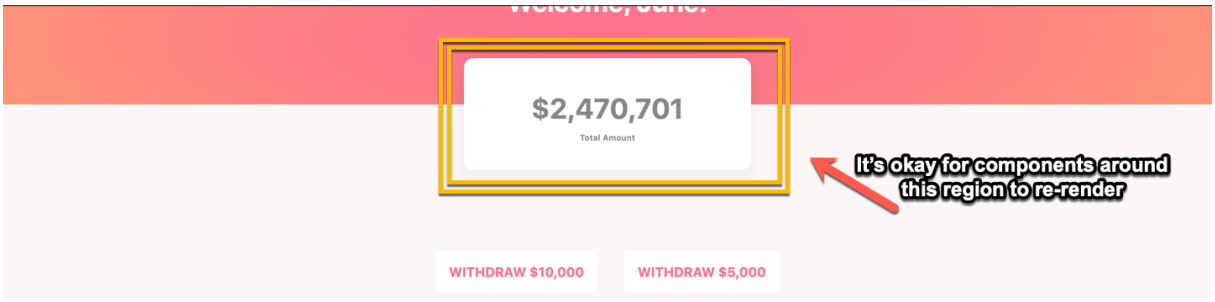
Let's consider the bank application we want to profile. The interaction in the bank app is simple. You click a set of buttons, and the withdrawal amount is updated.



Now, what would you consider the expected behaviour of this app with respect to re-renders and updates?

Well, for me, it's quite simple.

The only part of the app visually changing is the withdrawal amount. Before going into the profiling session, my expectation for a performant application will be that no unnecessary components are re-rendered, just the component responsible for updating the total amount.

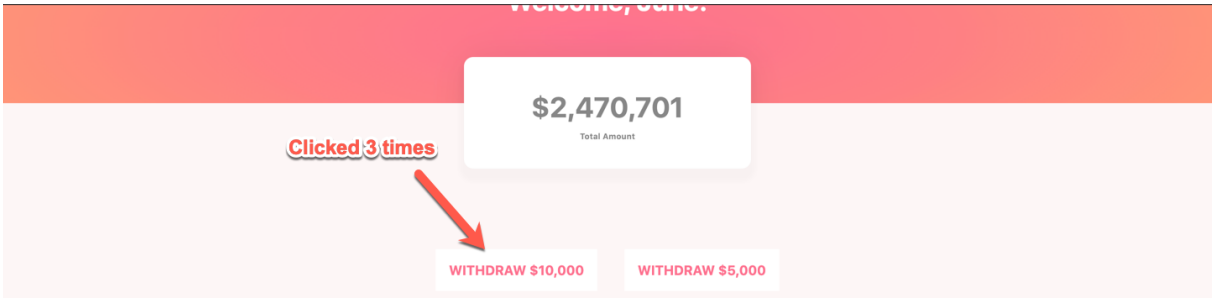


Give or take, I'll expect just the `TotalAmount` component to be re-rendered, or any other component directly connected with that update.

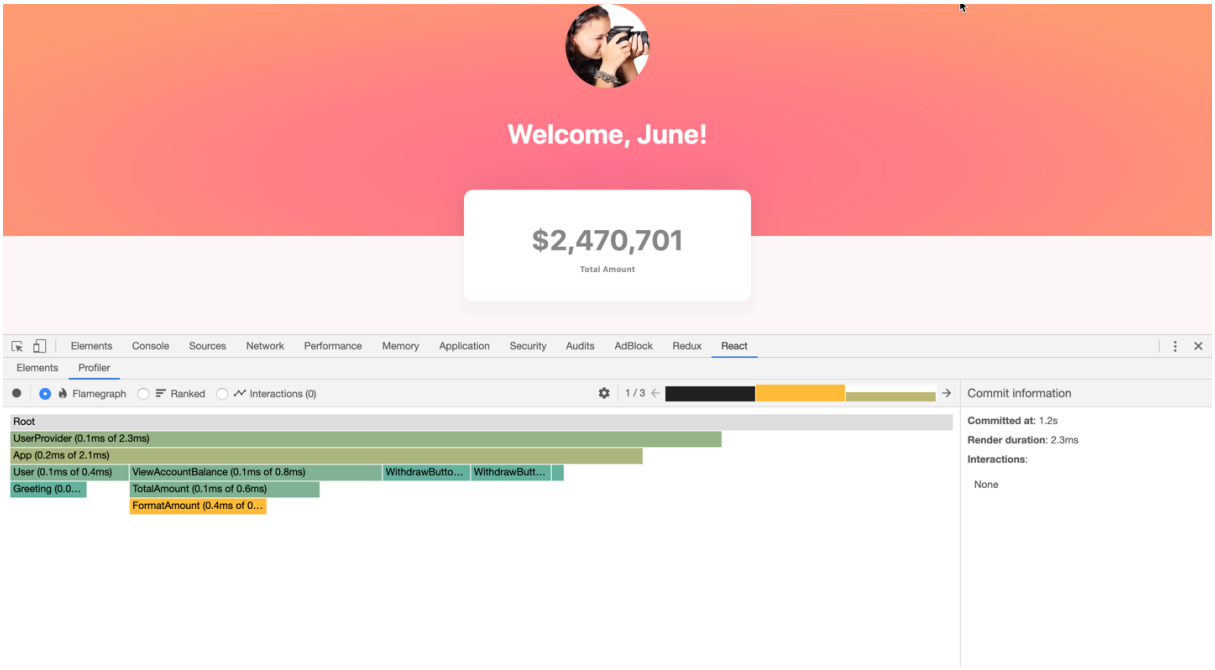
With this expectation set, let's go on and profile the application.

The steps are the same as discussed earlier. You open your `devtools`, record an interaction session, and begin to interpret the results.

Now, I have gone ahead to record a session. In this session, all I did was click the "withdraw $10,000" button 3 times.



Here's the flame chart from the profiling session:



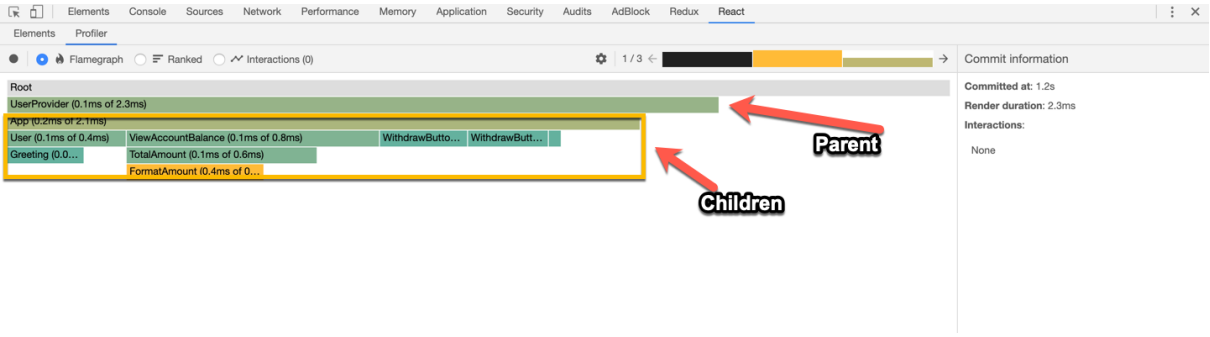
Oh my! From the chart above, so many components were re-rendered. You see the many bar colours represented in the flame chart ?

This is far from ideal, so let's begin to fix the problem.

Interpreting the Flame chart.

First let's consider what's likely the root of the problem here. By default, whenever a `Provider` has its `value` changed, every child component is forced to re-render. That's how the `Consumer` gets the latest values from the `context` object and stays in sync.

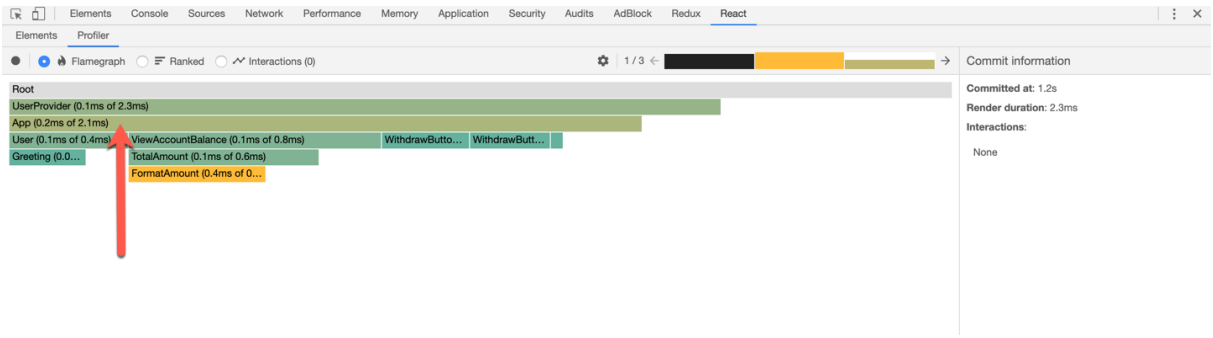
The problem here is that every component apart from `Root` is a child of `Provider` – and they all get re-rendered needlessly!



So, what can we do about this?

Some of the child components don't need to be re-rendered as they are not directly connected with the change.

First, let's consider the first child component, `App`.



The `App` component doesn't receive any `prop` and it only manages the state value `showBalance`.

```
class App extends Component {
  state = {
```

```
    showBalance: false
  }
  displayBalance = () => {
    this.setState({ showBalance: true })
  }
  render () {
    const { showBalance } = this.state
    ...
  }
}
```

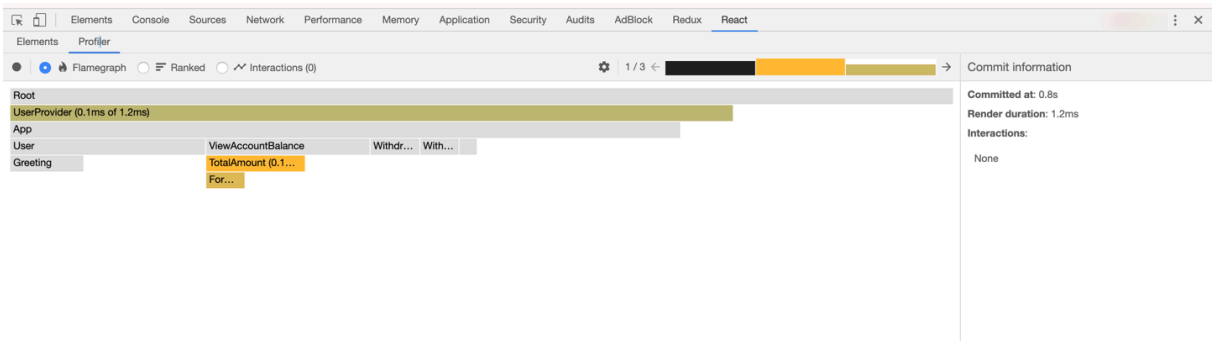
It isn't directly connected with the change, and it's pointless to re-render this component.

Let's fix this by making it a `PureComponent`.

```
// before
class App extends Component {
  state = {
    showBalance: false
  }
 ...
}
// after
class App extends PureComponent {
  state = {
    showBalance: false
  }
 ...
}
```

Having made `App` a `PureComponent`, did we make any decent progress?

Well, have a look at the new flame chart generated after that simple (one-liner) change.



Can you see that?

A lot of App's children aren't re-rendered needlessly, and we have a more sane flame graph now.

Great!

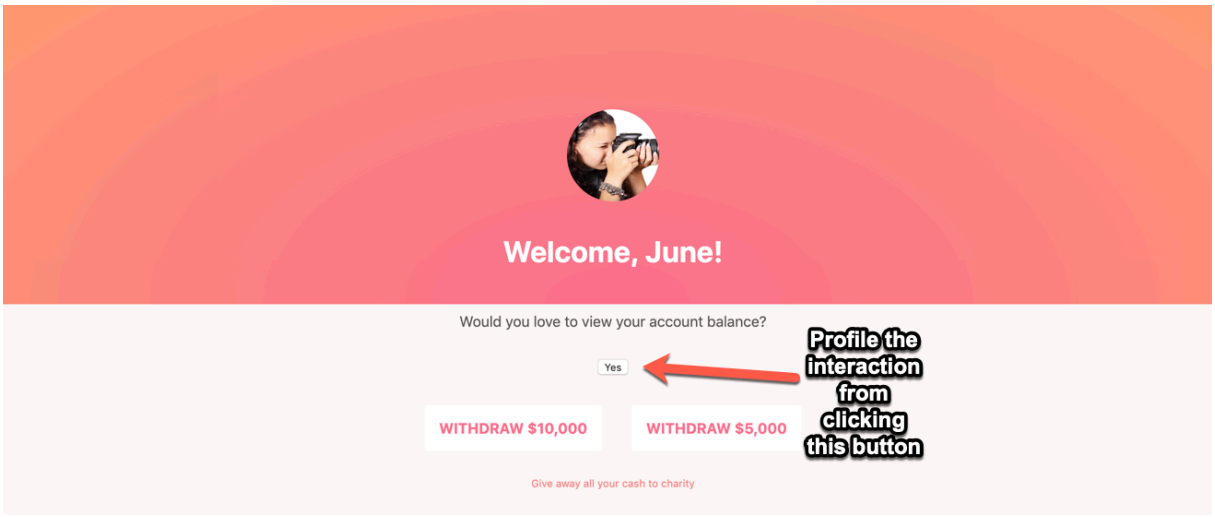
Profile Different Interactions.

It's easy to assume that because we had fewer re-renders in the "withdraw amount" interaction we now have a performant app.

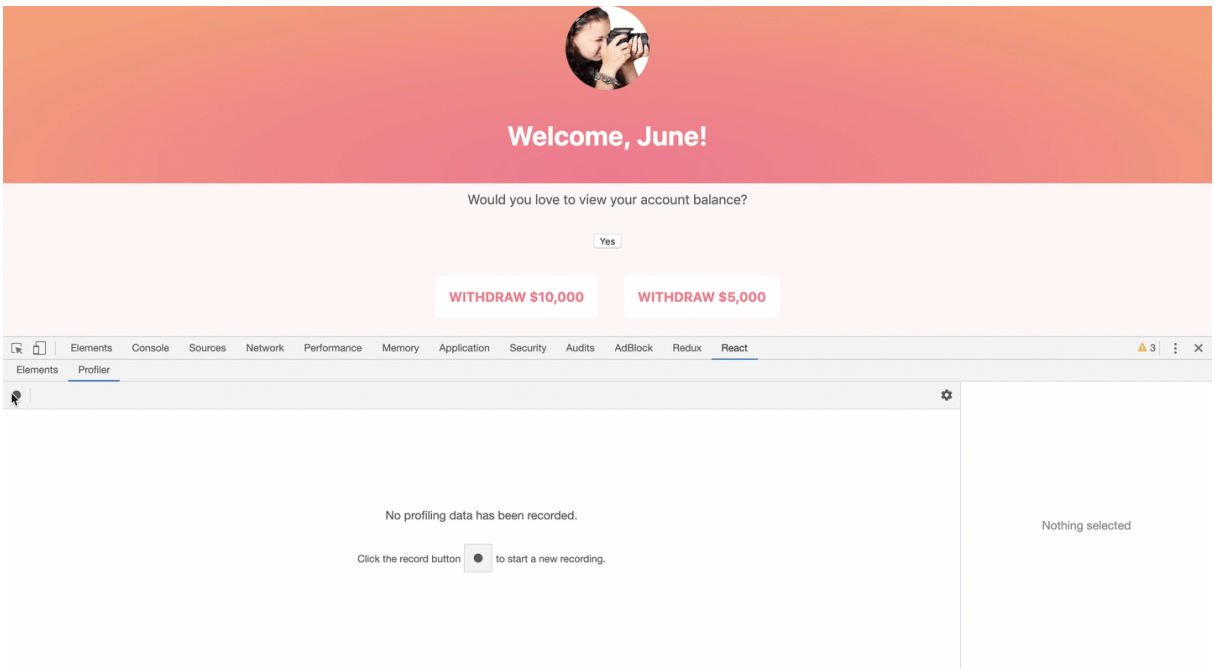
That's not correct.

App's now a `PureComponent`, but what happens when `App` gets rendered owing to a `state` change?

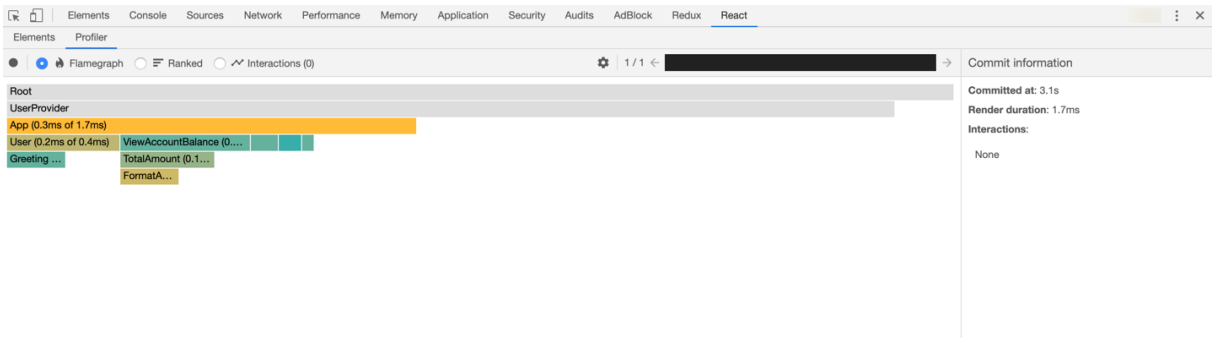
Well, let's profile a different interaction. This time, load up the application and profile the interaction for viewing an account balance.



If you go ahead and profile the interaction, we get a completely different result.

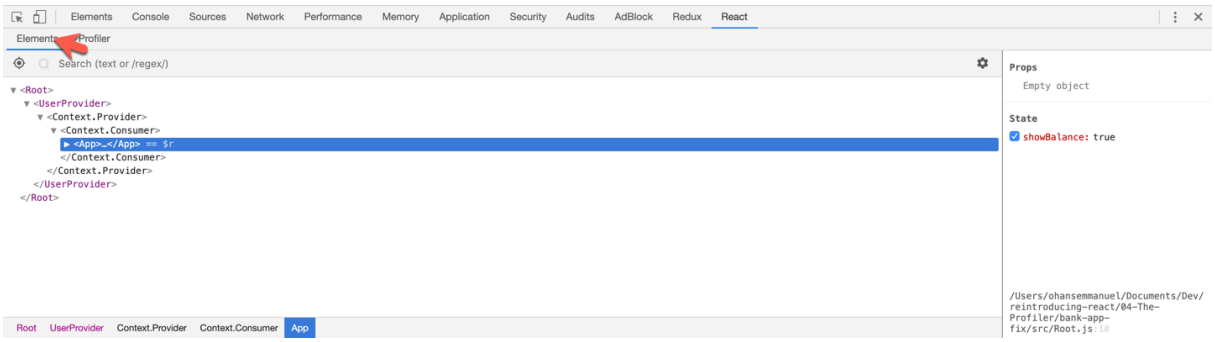


Now, what's changed?



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:

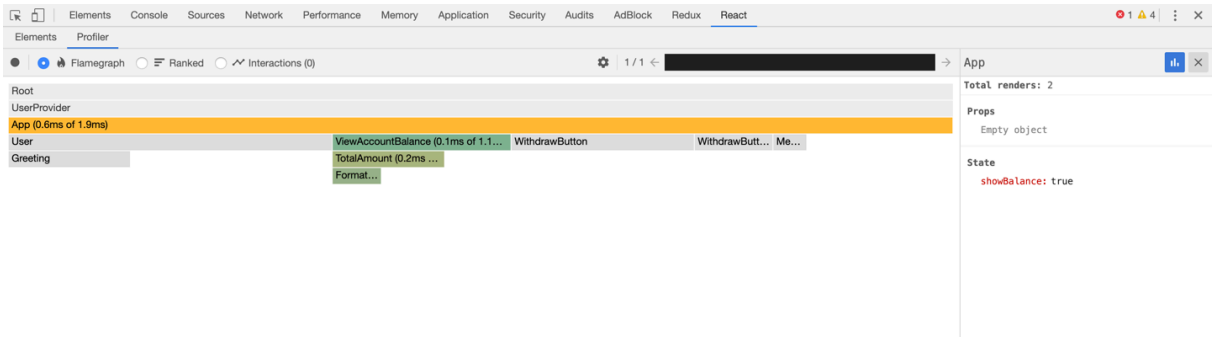


Well, since these child components are functional components, let's use `React.memo` to memoize the render results so they don't change except there's a change in props.

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

```
// WithdrawButton.js
import { memo } from 'react'
const WithdrawButton = memo(({ amount }) => {
    ...
  )
})
```

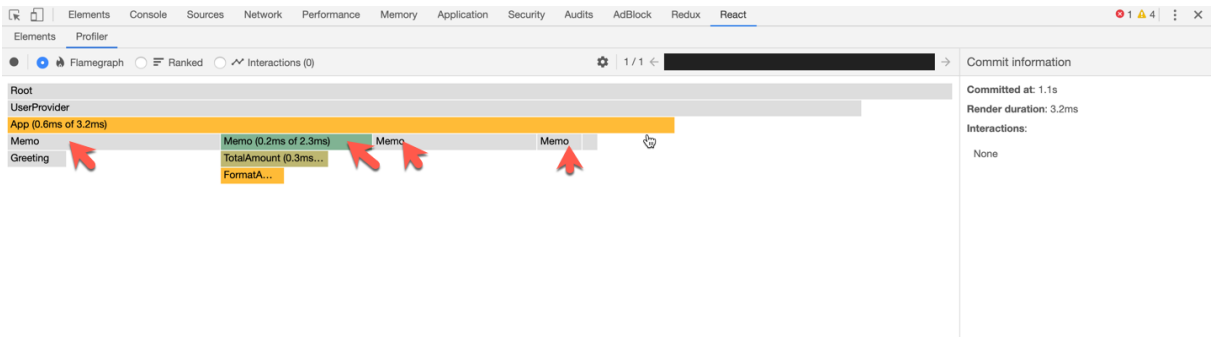
Now, when you do that and re-profile the interaction, we get a much nicer flame chart:



Now, only `ViewAccountBalance` and other child components are re-rendered. That's okay.

When you view your flame chart i.e if you're following along, you may see something slightly different.

The names of the component may not be shown. You get the generic name `Memo` and it becomes difficult to track which component is what.



To change this, set the `displayName` property for the memoized components.

Below's an example.

```
// ViewAccountBalance.js
const ViewAccountBalance = memo(({ showBalance, displayBalance }) => {
  ...
})
// set the displayName here
ViewAccountBalance.displayName = 'ViewAccountBalance'
```

You go ahead and do this for all the memoized functional components.

The Provider Value.

We're pretty much done with resolving the performance leaks in the application, however, there's one more thing to do.

The effect isn't very obvious in this application, but will come handy as you face more cases in the real world such as in situations where a `Provider` is nested within other components.

The implementation of the `Provider` in the bank application had the following:

```
...
<Provider
    value={{
       user: loggedInUser,
       handleLogin: this.handleLogin
       handleWithdrawal: this.handleWithdrawal
     }}
   >
 {this.props.children}
</Provider>
...
```

The problem here is that we're passing a new object to the `value` prop every single time. A better solution will be to keep a reference to these values via state. e.g.

```
<Provider value={this.state}>
    {this.props.children}
</Provider>
```

Doing this requires a bit of refactoring as shown below:

```
// context/UserContext.js
class UserProvider extends Component {
  constructor () {
    super()
    this.state = {
      user: null,
      handleLogin: this.handleLogin,
      handleWithdrawal: this.handleWithdrawal
    }
  }
  ...
  render () {
    return <Provider value={this.state}>
        {this.props.children}
    </Provider>
  }
}
```

Be sure to look in the associated code folder if you need more clarity on this.

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.

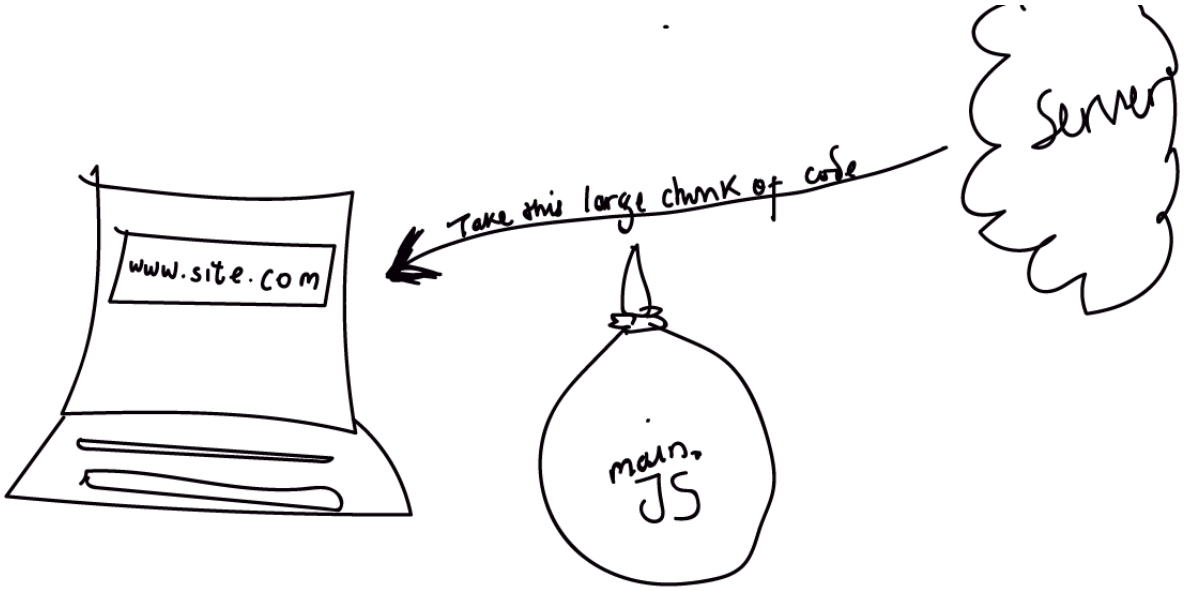
John's had great feedback from his manager for the past few months. Every now and then Tunde storms into the office with a new project idea. Today, it's lazy loading with `React.Lazy` and `Suspense`.

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.



To understand lazy loading, here's the specific use case Tunde had in mind when he discussed with John.

"*Hey John, do you remember the Benny app has an initial home screen?*", said Tunde.

By initial home scree, Tunde was referring to this:

This is the first screen the user encounters when they visit the Benny game. To begin playing the game, you must click the "Start Game" button to be redirected to the actual game scene.

"*John, the problem here is that we've bundled all our React components together and are all served to the user on this page*".

"*Oh, I see where you're going*", said John.

"*Instead of loading the* `GameScene` *component and its associated assets, we could defer the loading of those until the user actually clicks 'Start Game', huh?*", said John.

And Tunde agreed with a resounding "*Yes, that's exactly what I mean*".

Lay loading refers to deferring the loading of a particular resource until much later, usually when a user makes an interaction that demands the resource to be actually loaded. In some cases it could also mean preloading a resource.

Essentially, the user doesn't get the lazy loaded bundle served to them initially, rather it is fetched much later at runtime.

This is great for performance optimisations, initial load speed etc.

React makes lazy loading possible by employing the dynamic import syntax.

Dynamic imports refer to a `tc39` syntax proposal for javascript, however, with transpilers like Babel, we can use this syntax today.

The typical, static way of importing a module looks like this:

```
import { myModule } from 'awesome-module'
```

While this is desirable in many cases, the syntax doesn't allow for dynamically loading a module at runtime.

Been able to dynamically load a part of a Javascript application at runtime makes way for interesting use cases such as loading a resource based on a user's language (a factor that can only be determined at runtime), or only loading some code just when it is likely to be used by the user (performance gains).

For these reasons (and more) there's a proposal for introducing the dynamic import syntax to Javascript.

Here's what the syntax looks like:

```
import('path-to-awesome-module')
```

It has a syntax similar to a function, but is really not a function. It doesn't inherit from `Funtion.proptotype` and you can't invoke methods such as `call` and `apply`.

The returned result from the dynamic import call is a promise which is resolved with the imported module.

```
import('path-to-awesome-module')
    .then(module => {
     // do something with the module here e.g. module.default() to
invoke the default export of the module.
 })
```

Using React.lazy and Suspense.

`React.lazy` and `Suspense` make using dynamic imports in a React application so easy.

For example, consider the demo code for the Benny application below:

```
import React from 'react'
import Benny from './Benny'
import Scene from './Scene'
import GameInstructions from './GameInstructions'

class Game extends Component {
  state = {
    startGame: false
  }
  render () {
    return !this.state.startGame ?
        <GameInstructions /> :
        <Scene />
  }
}
export default Game;
```

Based on the state property `startGame`, either the `GameInstructions` or `Scene` component is rendered when the user clicks the "Start Game" button.

`GameInstructions` represents the home page of the game and `Scene` the entire scene of the game itself.

In this implementation, `GameInstructions` and `Scene` will be bundled together in the same Javascript resource.

Consequently, even when the user hasn't shown intent to start playing the game, we would have loaded and sent to the user, the complex `Scene` component which contains all the logic for the game scene.

So, what do we do?

Let's defer the loading of the `Scene` component.

Here's how easy `React.lazy` makes that.

```
// before
import Scene from './Scene'
// now
const Scene = React.lazy(() => import('./Scene'))
```

`React.lazy` takes a function that must call a dynamic import. In this case, the dynamic import call is `import('./Scene')`.

Note that `React.lazy` expects the dynamically loaded module to have a `default` export containing a React component.

With the `Scene` component now dynamically loaded, when the application is bundled for production, a separate module (or javascript file) will be created for the `Scene` dynamic import.

When the app loads, this javascript file won't be sent to the user. However, if they click the "Start Game" button and show intent to load the `Scene` component, a network request would be made to fetch the resource from the remote server.

Now, fetching from the server introduces some latency. To handle this, wrap the `Scene` component in a `Suspense` component to show a fallback for when the resource is being fetched.

Here's what I mean:

```
import { Suspense } from 'react'
const Scene = React.lazy(() => import('./Scene'))

class Game extends Component {
```

```
  state = {
    startGame: false
  }
  render () {
    return !this.state.startGame ?
        <GameInstructions /> :
        // look here
        <Suspense fallback="<div>loading ...</div>">
          <Scene />
        </Suspense>
  }
}
export default Game;
```

Now, when the network request is initiated to fetch the `Scene` resource, we'll show a "loading…" fallback courtesy the `Suspense` component.

`Suspense` takes a `fallback` prop which can be a markup as shown here, or a full blown React component e.g. a custom loader.

With `React.lazy` and `Suspense` you can suspend the fetching of a component until much later, and show a fallback for when the resource is being fetched.

How convenient.

Also, you can place the `Suspense` component anywhere above the lazy loaded component. In this case the `Scene` component.

If you also had multiple lazy loaded components, you could wrap them in a single `Suspense` component or multiple, depending on your specific use case.

Handling Errors.

In the real-world, things break often, right?

It's possible that in the process of fetching the lazy loaded resource, a network error occurs.

To handle such case, be sure to wrap your lazy loaded components in a *Error Boundary*.

Remember error boundaries from the Lifecycle method chapter?

Here's an example:

```
import { Suspense } from 'react'
import MyErrorBoundary from './MyErrorBoundary'
const Scene = React.lazy(() => import('./Scene'))
class Game extends Component {
  state = {
    startGame: false
  }

  render () {
    return <MyErrorBoundary>
         {!this.state.startGame ?
           <GameInstructions /> :
           <Suspense fallback="loading ...">
             <Scene />
           </Suspense>}
        </MyErrorBoundary>
  }
}
export default Game;
```

Now, if an error occurs while fetching the lazy loaded resource, it'll be graciously handled by the error boundary.

No named exports.

If you remember from the section above, I did mention that `React.lazy` expects the dynamic import statement to include a module with a **default export** being a React component.

At the moment, `React.lazy` doesn't support named exports. That's not entirely a bad thing, as it keeps tree shaking working so you don't import actual unused modules.

Consider the following module:

```
// MyAwesomeComponents.js
export const AwesomeA = () => <div> I am awesome </div>
export const AwesomeB = () => <div> I am awesome </div>
export const AwesomeC = () => <div> I am awesome </div>
```

Now, if you attempt to use `React.lazy` with a dynamic import of the module above, you'll get an error.

```
// SomeWhereElse.js
const Awesome = React.lazy(() => import('./MyAwesomeComponents'))
```

That won't work since there's no default export in the `MyAwesomeComponents.js` module.

A workaround would be to create some other module that exports one of the components as a default.

For example, if I was interested in lazy loading the `AwesomeA` component from the `MyAwesomeComponents.js` module, I could create a new module like this:

```
// AwesomeA.js
export { AwesomeA as default } from './MyAwesomeComponents'
```

Then I can can go ahead to effectively use `React.lazy` as follows:

```
// SomeWhereElse.js
```

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

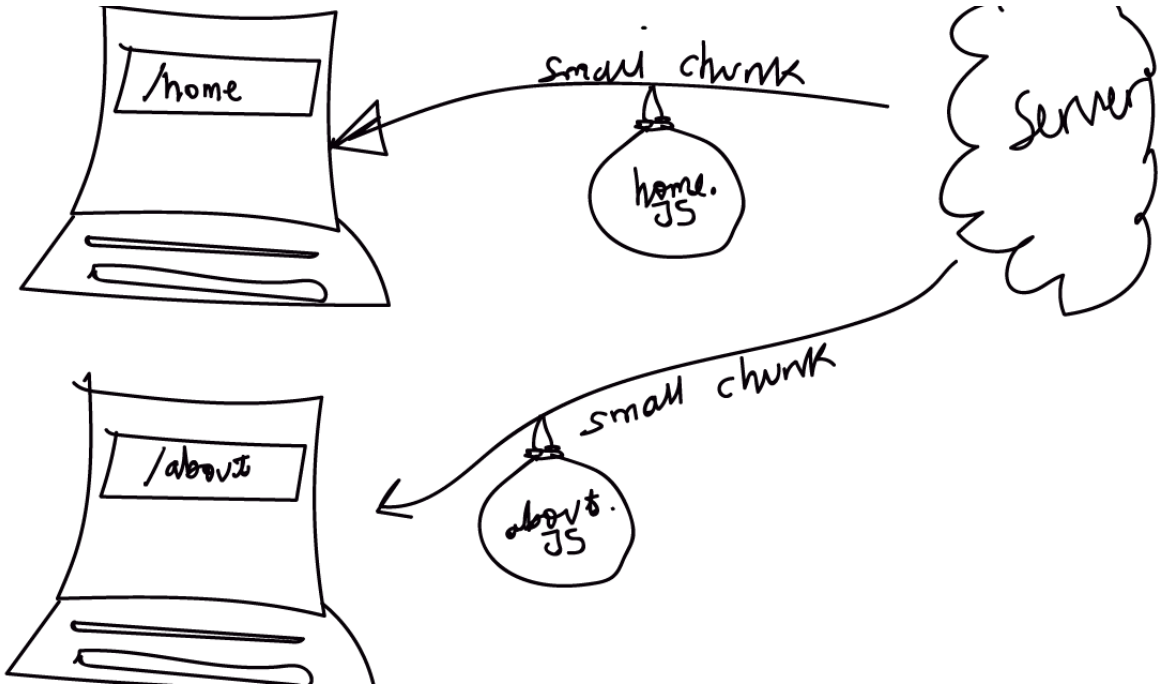
Problem solved!

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>
```

```
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
  </Router>
);
```

We could lazy load the `Home` and `About` components so they are only fetched when the user hits the associated routes.

Here's how with `React.Lazy` and `Suspense`.

```
// lazy load the route components
const Home = React.lazy(() => import('./Home'))
const About = React.lazy(() => import('./About'))
// Provide a fallback with Suspense
const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

Easy, huh?

We've discussed how `React.Lazy` and `Suspense` works, but under the hood, the actual code splitting i.e generating separate bundles for different modules is done by a bundler e.g. Webpack.

If you use `create-react-app`, `Gatsby` or `Next.js` then you have this already set up for you.

Setting this up yourself is also easy, you just need to tweak your `Webpack` config a little bit.

The official `Webpack` documentation has an entire guide on this. The guide may be worth checking if you're handling the budding configurations in your application yourself.

Example: Adding Lazy Loading to the Bank App.

We can add some lazy loading to the bank application we saw from Chapter 2.

Consider the `Root` component of the application:

```
const Root = () => (
  <UserProvider>
    <UserConsumer>
      {({ user, handleLogin }) =>
        user ? <App /> : <Login handleLogin={handleLogin} />
      }
    </UserConsumer>
  </UserProvider>
)
```

When a user isn't logged in we display the login page, and the `App` component only when the user is logged in.

We could lazy load the `App` component, right?

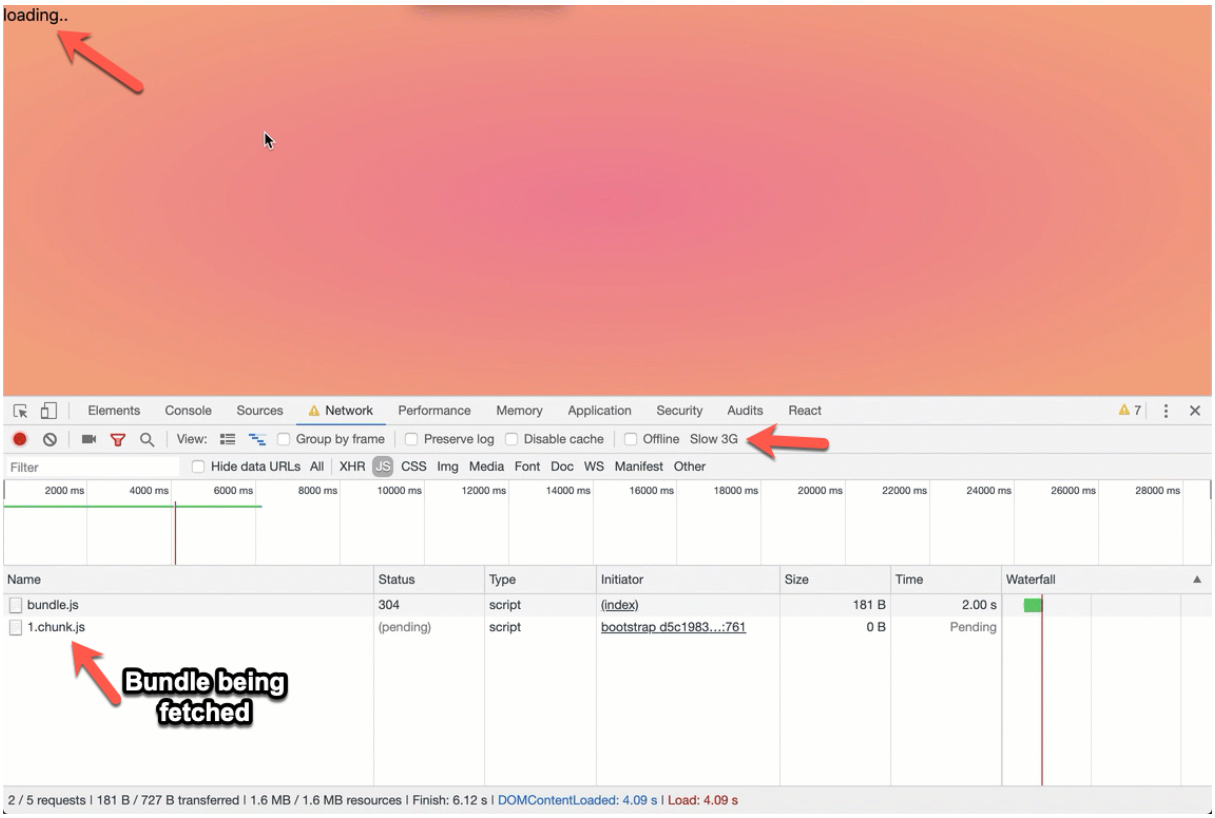
This is very easy. You use the dynamic import syntax with `React.lazy` and wrap the `App` component in a `Suspense` component.

Here's how:

```
...
```

```
const App = React.lazy(() => import('./containers/App'))
const Root = () => (
  ...
  <Suspense fallback='loading...'>
     <App />
  </Suspense>
)
```

Now, if you throttle your network connection to simulate Slow 3G, you'll see the intermediate *"loading..."* text after logging in.



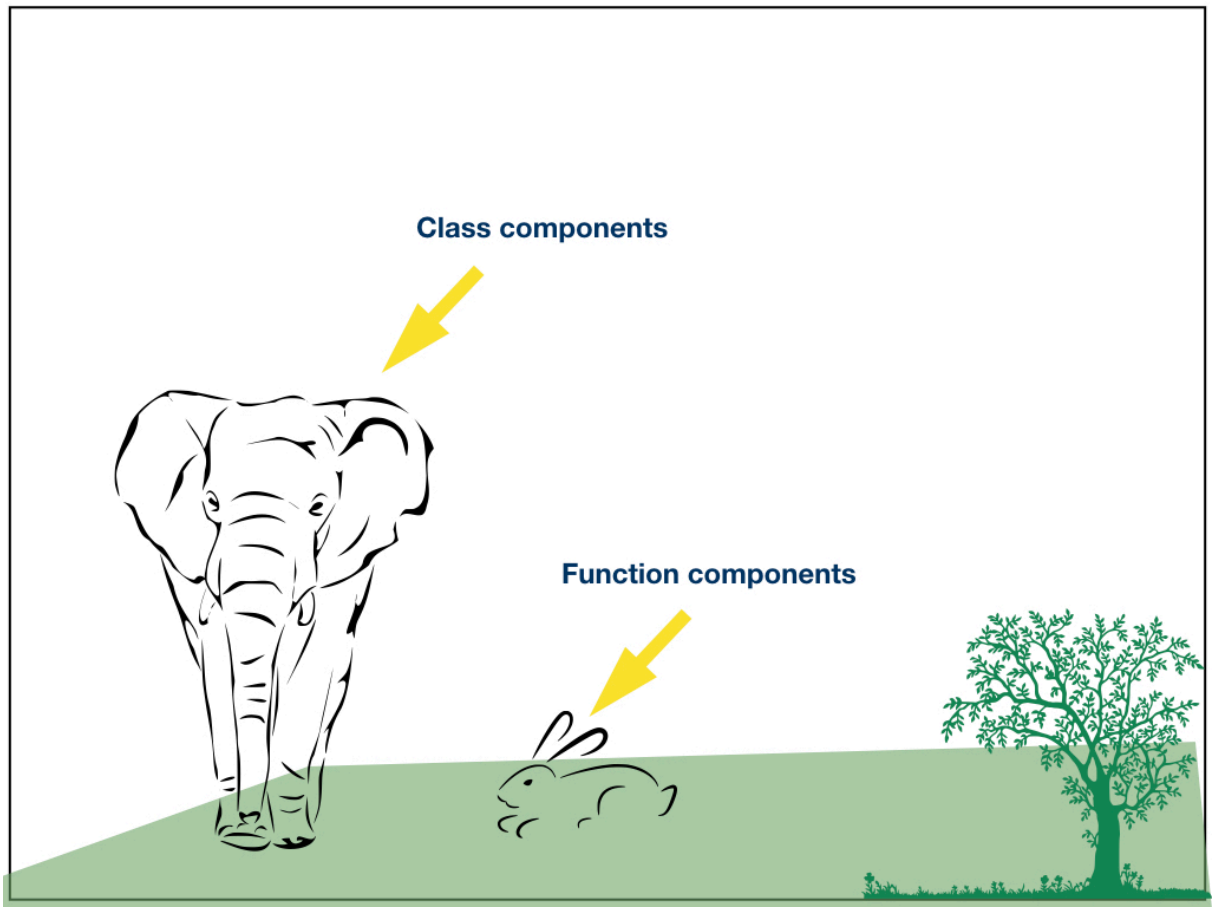
Conclusion.

`React.lazy` and `Suspense` are great, and so intuitive to work with, however, they do not support server side rendering yet.

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.

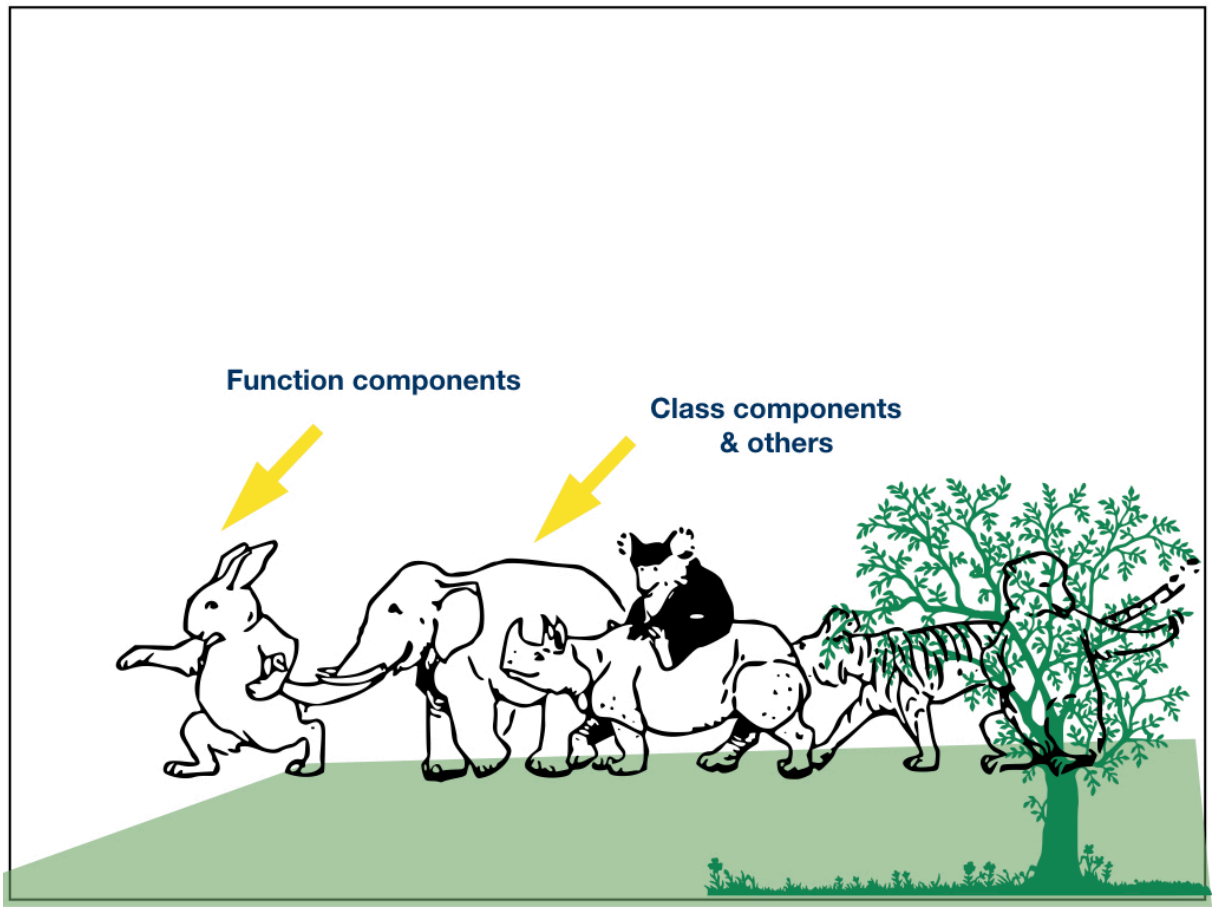


If you wanted local state or some other complex side effects, you had to reach out to class component. You either painfully refactor your functional components to class components or nothing else.

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

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.



That's a bold statement from Mia.

So, let's consider what hooks are.

Introducing Hooks.

Early this year, 2019, the React team released a new addition, hooks, to React in version `16.8.0`.

If React were a big bowl of candies, then hooks are the latest additions, very chewy candies with great taste!

So, what exactly do hooks mean? And why are they worth your time?

One of the main reasons hooks were added to React is to offer a more powerful and expressive way to write (and share) functionality between components.

> *In the longer term, we expect Hooks to be the primary way people write React components* — *React Team*

If hooks are going to be that important, why not learn about them in a fun way!

The Candy Bowl.

Consider React to be a beautiful bowl of candy.

The bowl of candy has been incredibly helpful to people around the world.

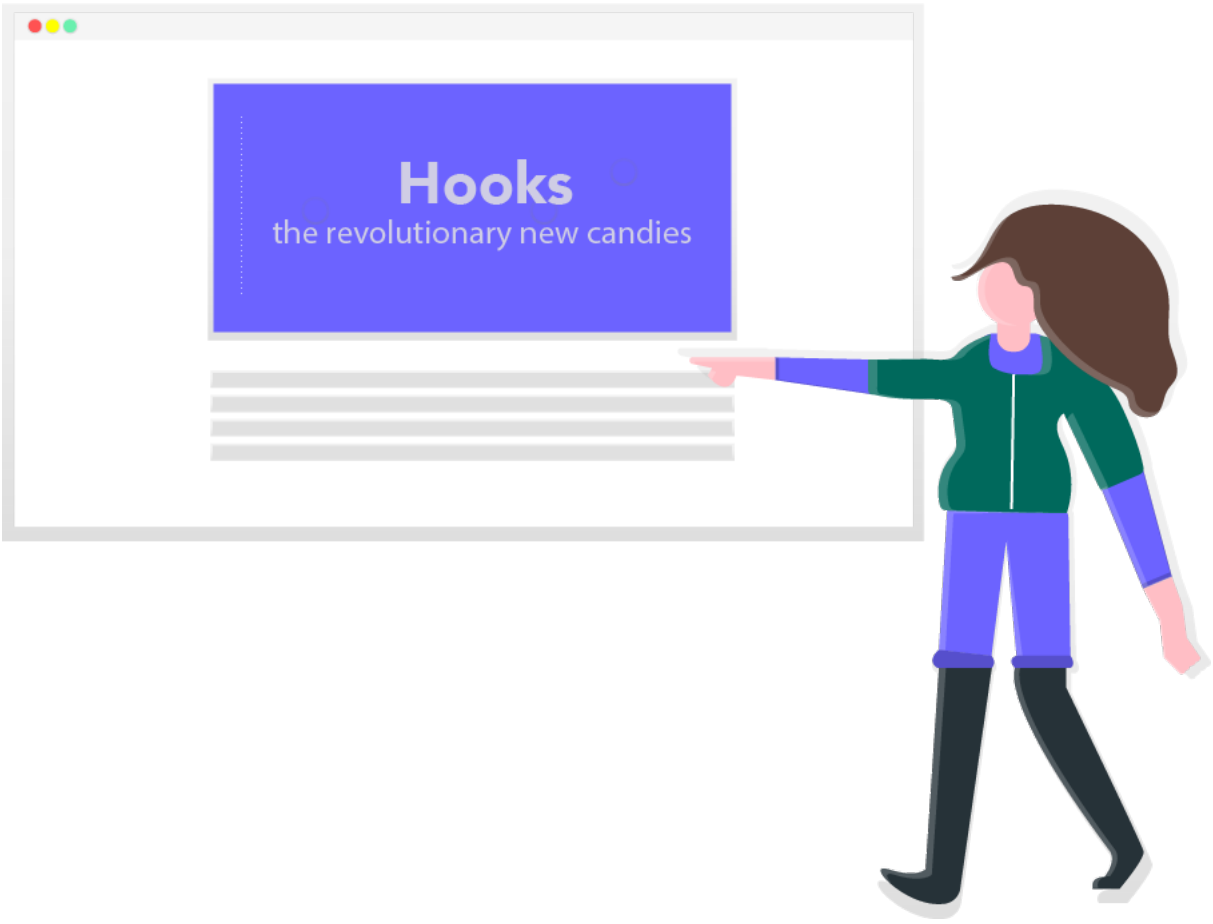
The people who made this bowl of candy realized that some of the candies in the bowl weren't doing people much good.

A couple of the candies tasted great, yes! But they brought about some complexity when people ate them—think render props and higher order components?

So, what did they do?

They did the right thing – not throwing away all the previous candies, but making new sets of candies.

These candies were called **Hooks**.



These candies exist for one purpose: **to make it easier for you to do the things you are already doing**.

These candies aren't super special. In fact, as you begin to eat them you'll realize they taste familiar—they are just **Javascript functions**!

As with all good candies, these 10 new candies all have their unique names. Though they are collectively called **hooks**.

Their names always begin with the three letter word, use … e.g. `useState`, `useEffect` etc.

Just like chocolate, these 10 candies all share some of the same ingredients. Knowing how one tastes, helps you relate to the other.

Sounds fun? Now let's have these candies.

The State Hook.

As stated earlier, hooks are functions. Officially, there are 10 of them. 10 new functions that exist to make writing and sharing functionalities in your components a lot more expressive.

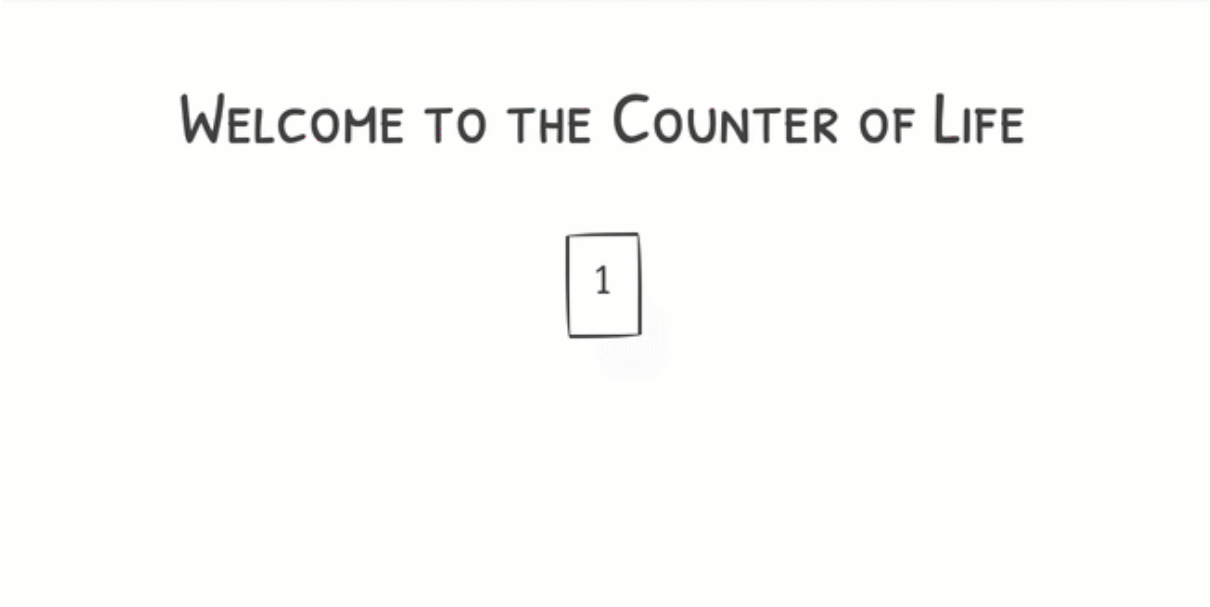
The first hook we'll take a look at is called, `useState`.

For a long time, you couldn't use the local state in a functional component. Well, not until hooks.

With `useState`, your functional component can have (and update) local state.

How interesting.

Consider the following counter application:



With the Counter component shown below:

```
import React, { Component } from 'react';
class Counter extends Component {
 state = {
    count: 1
 }
```

```
  handleClick = () => {
    this.setState(prevState => ({count: prevState.count + 1}))
  }
  render() {
    const { count } = this.state;
    return (
      <div>
        <h3 className="center">
            Welcome to the Counter of Life
        </h3>
        <button className="center-block" onClick={this.handleClick}>
            {count}
        </button>
      </div>
    );
  }
}
```

Simple, huh?

Let me ask you one simple question. Why exactly do we have this component as a `Class` component?

Well, the answer is simply because we need to keep track of some local state within the component.

Now, here's the same component refactored to a functional component with access to state via the `useState` hooks.

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

```
  }
  return (
      <div>
        <h3 className="center">
          Welcome to the Counter of Life
        </h3>
        <button
           className="center-block"
           onClick={handleClick}>
         {count}
        </button>
      </div>
    );
}
```

What's different?

I'll walk you through it step by step.

A functional component doesn't have all the `Class extend ...` syntax.

```
function CounterHooks() {
}
```

It also doesn't require a render method.

```
function CounterHooks() {
  return (
      <div>
        <h3 className="center">Welcome to the Counter of Life </h3>
        <button
          className="center-block"
          onClick={this.handleClick}> {count} </button>
      </div>
```

```
    );
}
```

There are two concerns with the code above.

- You're not supposed to use the `this` keyword in function components.
- The `count` state variable hasn't been defined.

Extract `handleClick` to a separate function within the functional component:

```
function CounterHooks() {
  const handleClick = () => {
  }
  return (
      <div>
        <h3 className="center">Welcome to the Counter of Life </h3>
        <button
          className="center-block"
          onClick={handleClick}> {count} </button>
      </div>
    );
}
```

Before the refactor, the count variable came from the class component's state object.

In functional components, and with hooks, that comes from invoking the `useState` function or hook.

`useState` is called with one argument, the initial state value e.g. useState(0) where 0 represents the initial state value to be kept track of.

Invoking this function returns an array with two values.

```
//🦄 returns an array with 2 values.
```

```
useState(0)
```

The first value being the current `state` value being tracked, and second, a function to update the `state` value.

Think of this as some `state` and `setState` replica - however, they aren't quite the same.

With this new knowledge, here's `useState` in action.

```
function CounterHooks() {
  // 🦄
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1)
  }
  return (
      <div>
        <h3 className="center">Welcome to the Counter of Life </h3>
        <button
          className="center-block"
          onClick={handleClick}> {count} </button>
      </div>
    );
}
```

There are a few things to note here, apart from the obvious simplicity of the code!

One, since invoking `useState` returns an array of values, the values could be easily destructed into separate values as shown below:

```
const [count, setCount] = useState(0);
```

Also, note how the `handleClick` function in the refactored code doesn't need any reference to `prevState` or anything like that.

It just calls `setCount` with the new value, `count + 1`.

Simple as it sounds, you've built your very first component using hooks. I know it's a contrived example, but that's a good start!

NB: it's also possible to pass a function to the state updater function. This is usually recommended as with `setState` when a state update depends on a previous value of state e.g. `setCount(prevCount => prevCount + 1)`

Multiple useState calls.

With class components, we all got used to setting state values in an object whether they contained a single property or more.

```
// single property
state = {
  count: 0
}
// multiple properties
state = {
 count: 0,
 time: '07:00'
}
```

With `useState` you may have noticed a subtle difference.

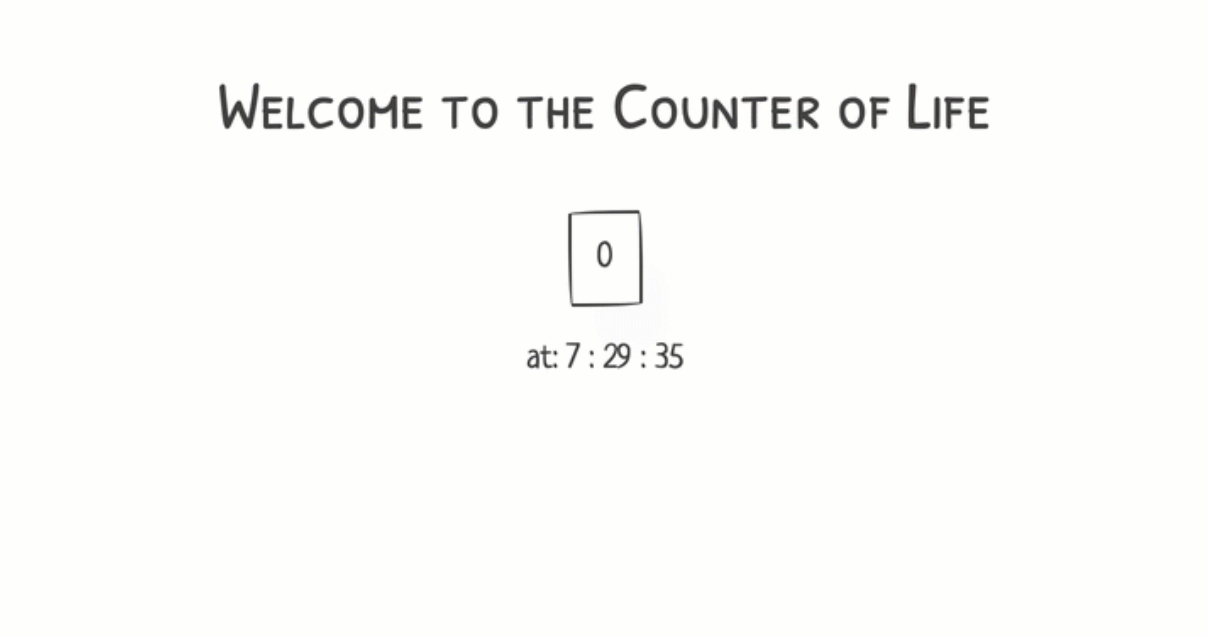
In the example above, we only called `useState` with the actual initial value. Not an object to hold the value.

```
useState(0)
```

So, what if we wanted to keep track of another state value?

Can multiple `useState` calls be used?

Consider the component below. It's the same counter application with a twist. This time the counter keeps track of the time of click.




```
function CounterHooks() {
  const [count, setCount] = useState(0);
  const [time, setTime] = useState(new Date())
  const handleClick = () => {
    setCount(count + 1);
    setTime(new Date())
  }
  return (
      <div>
        <h3>Welcome to the Counter of Life </h3>
        <button onClick={handleClick}>{count}</button>
        <p>
          at: {`${time.getHours()}
            ...
        </p>
       </div>
    );
```

```
}
```

As you can see, the hooks usage is quite the same, except for having a new `useState` call.

```
const [time, setTime] = useState(new Date())
```

Now, the `time`state variable is used in the rendered markup to display the hour, minute and second of the click.

```
<p>
    at: {`${time.getHours()} : ${time.getMinutes()} :$
{time.getSeconds()}`}
  </p>
```

Great!

Object as Initial Values

Is it possible to use an object with `useState` as opposed to multiple `useState` calls?

Absolutely!

If you choose to do this, you should note that unlike `setState` calls, the values passed into `useState` replaces the state value.

`setState` merges object properties but `useState` replaces the entire value.

```
// 🦕 merge (setState) vs replace (useState)
// assume initial state is {name: "Ohans"}
setState({age: "unknown"})
// new state object will be
// {name: "Ohans", age: "unknown"}
useState({age: "unknown"})
// new state object will be
```

```
// {age: "unknown"} - initial object is replaced
```

The Effect Hook.

With class components you've likely performed side effects such as logging, fetching data or managing subscriptions.

These side effects may be called "effects" for short, and the effect hook, `useEffect` was created for this purpose.

How's it used?

Well, the `useEffect` hook is called by passing it a function within which you can perform your side effects.

Below's a quick example:

```
useEffect(() => {
  // 🦕 you can perform side effects here
  console.log("useEffect first timer here.")
})
```

To `useEffect` I've passed an anonymous function with some side effect called within it.

The next logical question is, when is the `useEffect` function invoked?

Well, remember that in class components you had lifecycle methods such as `componentDidMount` and `componentDidUpdate`.

Since functional components don't have these lifecycle methods, `useEffect` *kinda* takes their place.

Thus, in the example above, the function within `useEffect` also known as the effect function, will be invoked when the functional component mounts (`componentDidMount`) and when the component updates `componentDidUpdate`).

Here's that in action.

By adding the `useEffect` call above to the counter app, we indeed get the log from the `useEffect` function.

```
function CounterHooks() {
  const [count, setCount] = useState(0);
  const [time, setTime] = useState(new Date())
  // 🦕 look here.
  useEffect(() => {
    console.log("useEffect first timer here.")
  }, [count])

  const handleClick = () => {
    setCount(count + 1);
    setTime(new Date())
  }
  return (
     ...
    );
}
```

By default, the `useEffect` function will be called after every render.

NB: The `useEffect` hook isn't entirely the same as `componentDidMount` + `componentDidUpdate`. It can be viewed as such, but the implementation differs with some subtle differences.

Passing Array Dependencies.

It's interesting that the effect function is invoked every time there's an update. That's great, but it's not always the desired functionality.

What if you only want to run the effect function only when the component mounts?

That's a common use case and `useEffect` takes a second parameter, an array of dependencies to handle this.

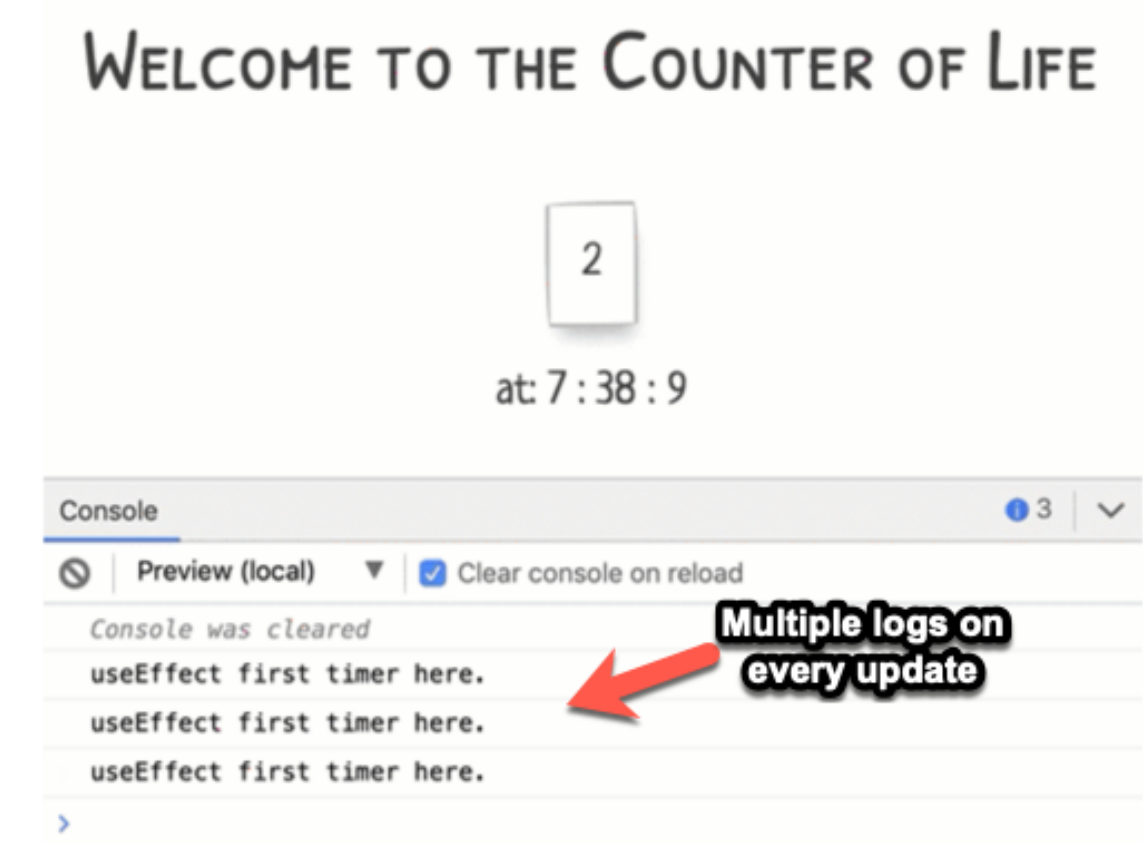
If you pass in an empty array, the effect function is run only on mount—subsequent re-renders don't trigger the effect function.

```
useEffect(() => {
    console.log("useEffect first timer here.")
}, [])
```

If you pass any values into this array, then the effect function will be run on mount, and anytime the values passed are updated. i.e if any of the values are changed, the effected call will re-run.

```
useEffect(() => {
    console.log("useEffect first timer here.")
}, [count])
```

The effect function will be run on mount, and whenever the count function changes.



What about subscriptions?

It's common to subscribe and unsubscribe from certain effects in certain apps.

Consider the following:

```
useEffect(() => {
  const clicked = () => console.log('window clicked');
  window.addEventListener('click', clicked);
}, [])
```

In the effect above, upon mounting, a click event listener is attached to the window.

How do we unsubscribe from this listener when the component is unmounted?

Well, `useEffect` allows for this.

If you return a function within your effect function, it will be invoked when the component unmounts. This is the perfect place to cancel subscriptions as shown below:

```
useEffect(() => {
    const clicked = () => console.log('window clicked');
    window.addEventListener('click', clicked);

    return () => {
      window.removeEventListener('click', clicked)
    }
 }, [])
```

There's a lot more you can do with the useEffect hook such as making API calls.

Build Your own Hooks

From the start of the hooks section we've taken (and used) candies from the candy box React provides.

However, React also provides a way for you to make your own unique candies – called custom hooks.

So, how does that work?

A custom hook is just a regular function. However, its name must begin with the word, **use** and if needed, it may call any of the React hooks within itself.

Below's an example:

```
// 🐢 custom hook - name starts with "use" (useMedium)
useMedium(() => {
  const URI = "https://some-path-api";
  // 🦄 custom hook can use any of the default
  // React hooks - this is NOT compulsory.
```

```
  useEffect(() => {
    fetch(URI)
  },[])
})
```

The Rules of Hooks

There are two rules to adhere to while using hooks.

- Only Call Hooks at the Top Level i.e. not within conditionals, loops or nested functions.
- Only Call Hooks from React Functions i.e. Functional Components and Custom Hooks.

This ESLint plugin is great to ensure you adhere to these rules within your projects.

Advanced Hooks

We have only considered two out of 10 of the hooks React provides!

What's interesting is that the knowledge of how `useState` and `useEffect` works helps you quickly learn the other hooks.

Curious to learn about those, I have created a hooks cheatsheet with live editable examples.

Why this is important is that you can immediately begin to tinker with real examples that'll reinforce your knowledge of how hooks work. All of them!

Remember that learning is reinforced when you actual solve problems and build stuff.

What's more interesting as well is, after you get through the live examples for each of the hooks, there's an extra section for other generic examples that don't exactly fit one hook or require a separate case study.

In the example section you'll find examples such as fetching data from a remote server using hooks and more.

Go, check it out.

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

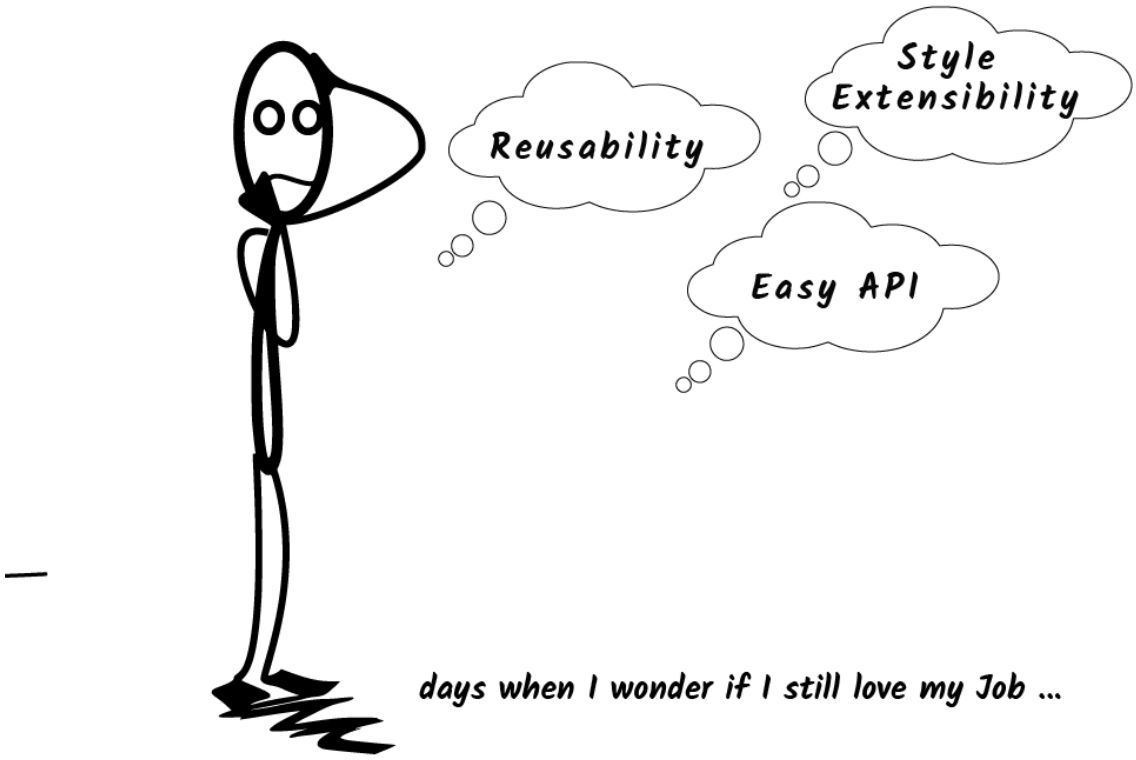
If you're completely new to these advanced patterns, don't worry, I'll explain them in detail. If you're familiar with how these patterns work from previous experiences with class components, I'll show you how to use these patterns with hooks.

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.



Yes, John can build components with his current React skills, however, with building highly reusable components comes specific problems.

There's a million different ways the components can be consumed, and you want to give consumers of the component as much flexibility as possible.

They must be able to extend the functionality and styles of the component as they deem fit.

The advanced patterns we'll consider here are tested and tried methods for building very reusable components that don't cripple flexibility.

I didn't create these advanced patterns. Truth be told, most of the advanced React patterns were made popular by one guy, Kent Dodds – an amazing Javascript engineer.

The community has received these patterns extremely well, and I'm here to help you understand how they work!

Compound Components Pattern

The first pattern we will consider is called the Compound Components pattern. I know it sounds fancy, so I'll explain what it really means.

The keyword in the pattern name is the word *Compound*.

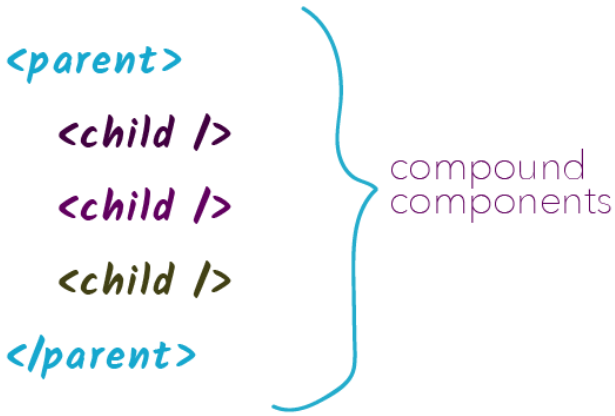
Literarily, the word *compound* refers to something that is composed of two or more separate elements.

With respect to React components, this could mean a component that is composed of two or more separate components.

It doesn't end there.

Any React component can be composed of 2 or more separate components. So, that's really not a brilliant way to describe compound components.

With compound components, there's more. The separate components within which the main component is composed cannot really be used without the parent.



The main component is usually called the parent, and the separate composed components, children.

The classic example is to consider the `html` select element.

```
<select>
  <option value="value0">key0</option>
  <option value="value1">key1</option>
  <option value="value2">key2</option>
</select>
```

With `select` being the parent, and the many `option` elements, children.

This works like a compound component. For one, it really makes no sense to use the `<option>key0</option>` element without a `select` parent tag. The overall behaviour of a `select` element also relies on having these composed `option` elements as well.

They are so connected to one another.

Also, the state of the entire component is managed by `select` with all child elements dependent on that state.

Do you get a sense for what compound components are now?

It is also worth mentioning that compound components are just one of many ways to express the API for your components.

For example, while it doesn't look as good, the `select` element could have been designed to work like this:

```
<select options="key:value;anotherKey:anotherValue"></select>
```

This is definitely not the best way to express this API as It makes passing attributes to the child components almost impossible.

With that in mind, let's take a look at an example that'll help you understand and build your own compound components.

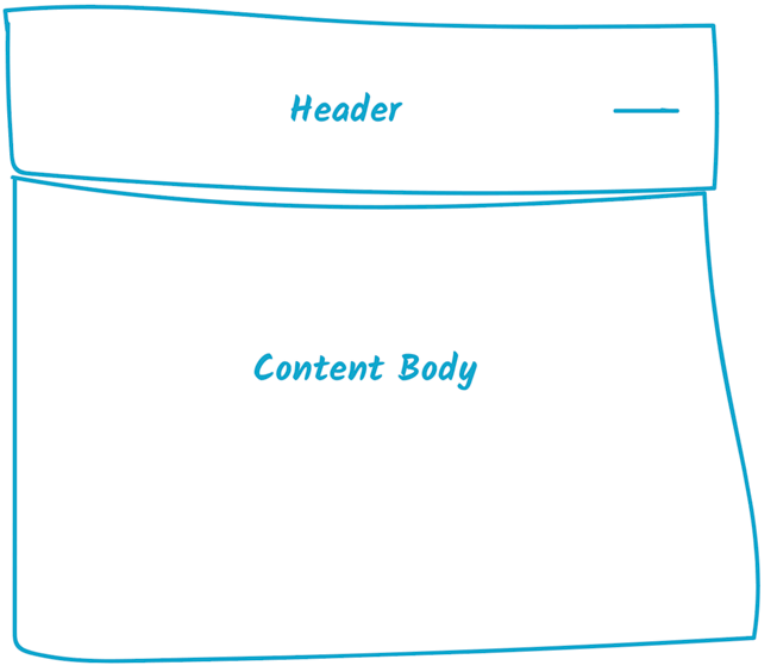
Example: Building an Expandable Component.

We'll be building an `Expandable` component. Did you just ask what that means?

Well, consider an `Expandable` component to be a miniature accordion element. It has a clickable header, which toggles the display of an associated body of content.

In the unexpanded state the component would look like this:

And this, when expanded:



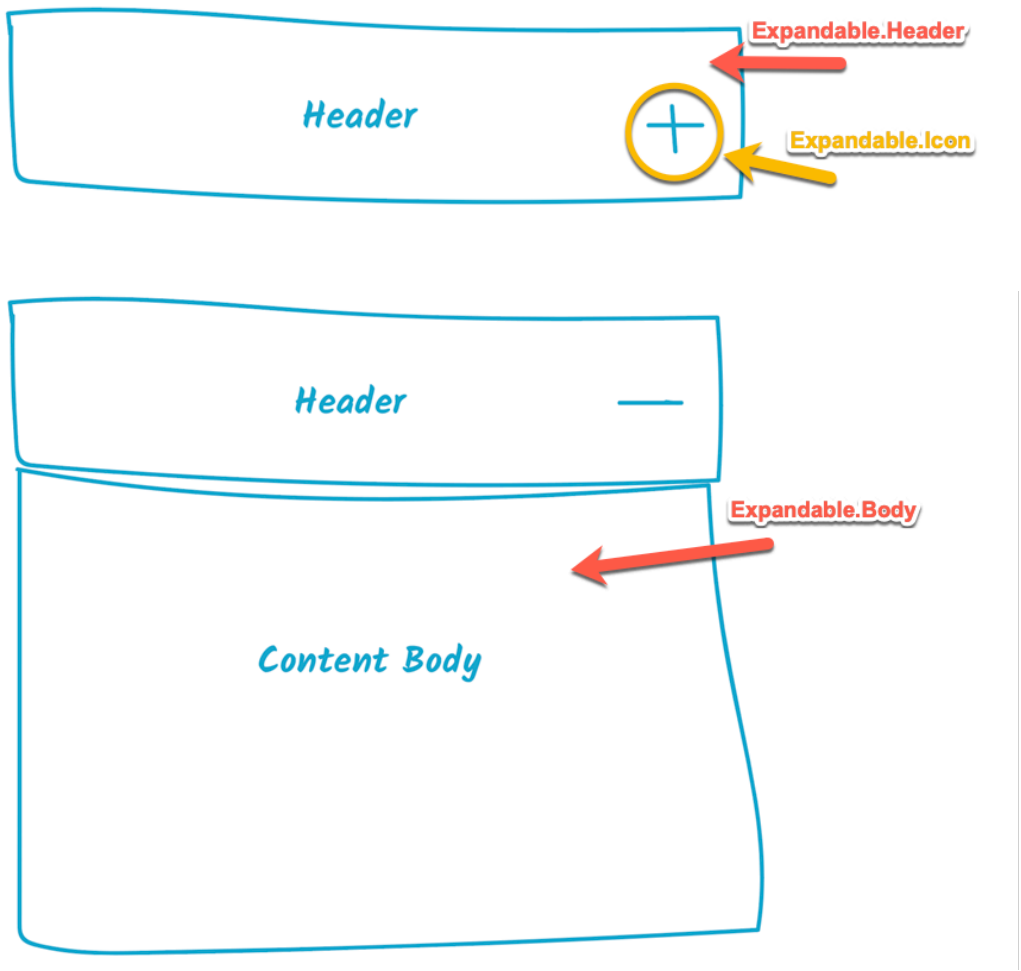
You get the idea now ?

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>
```



In the code block above you'd have noticed I have expressions like this: `Expandable.Header`

You could as well do this:

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

It doesn't matter. 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.

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

Building the Expandable Component

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

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

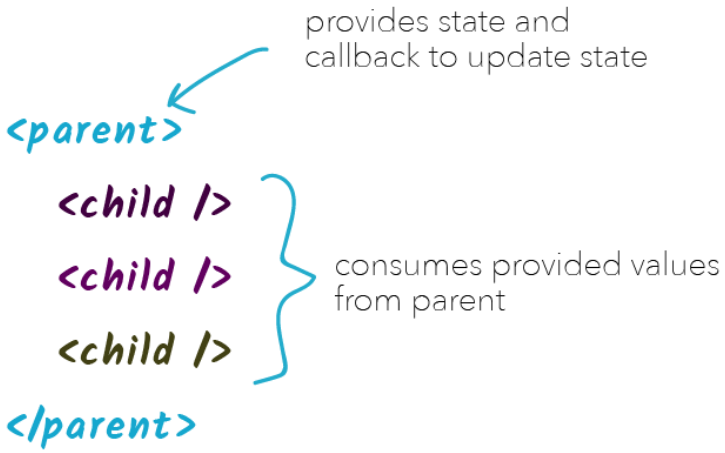
The `Expandable` component needs to communicate the state to every child component regardless of their position in the nested component tree.

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

How best may we go about this?

If you said `context`, you're correct!

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.



If that sounds alright to you, here's the starting point for the `Expandable` component.

```
// 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.

A context object is created and the `Provider` component 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`.

```
// Expandable.js
import React, { useState } from 'react'
...
const Expandable = ({children}) => {
  // look here 👇
  const [expanded, setExpanded] = useState(false)
  return <Provider>{children}</Provider>
}
```

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

```
// Expandable.js
...
const Expandable = ({children}) => {
  const [expanded, setExpanded] = useState(false)
  // look here 👇
  const toggle = setExpanded(prevExpanded => !prevExpanded)

  return <Provider>{children}</Provider>
}
```

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 thing I've done above. The function passed to `setExpanded` receives the previous value of `expanded` and returns the opposite of that, `!expanded`

`toggle` acts as a callback function and It'll eventually be invoked by `Expandable.Header`. Let's prevent any future performance issue by memoizing the callback.

```
...
import { useCallback } from 'react';


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

Not sure how `useCallback` works? You probably skipped the previous advanced hooks section that pointed to the cheatsheet. Have a look.

Once we have both `expanded` and `toggle` created, we can expose these via the `Provider`'s value prop.

```
...
const Expandable = ({children}) => {
  const [expanded, setExpanded] = useState(false)
```

```
  const toggle = useCallback(
    () => setExpanded(prevExpanded => !prevExpanded),
    []
  )
   // look here 👇
  const value = { expanded, toggle }
   // and here 👇
  return <Provider value={value}>{children}</Provider>
}
```

This works, but the `value` reference will be different on every re-render causing the `Provider` to re-render its children.

Let's memoize the `value`.

```
...
const Expandable = ({children}) => {
  ...
 // look here 👇
  const value = useMemo(
    () => ({ expanded, toggle }),
    [expanded, toggle]
  )
  return <Provider value={value}>{children}</Provider>
}
```

`useMemo` takes a callback that returns the object value `{ expanded, toggle }` and we pass an array dependency `[expanded, toggle]` so that the memoized value remains the same unless those change.

We've done a great job so far!

Now, there's just one other thing to do in the `Expandable` parent component.

Handling State Change Callbacks

Let's borrow a concept from years of experience with React's class components. If you remember, it's possible to do this with class components:

```
this.setState({
  name: "value"
}, () => {
  this.props.onStateChange(this.state.name)
})
```

If you don't have experience with class components, this is how you trigger a **callback after a state change** in class components.

Usually, the callback e.g. `this.props.onStateChange` is always invoked with the current value of the updated state as shown below:

```
this.props.onStateChange(this.state.name)
```

Why is this important?

This is good practice when creating reusable components, because this way the consumer of your component can attach any custom logic to be run after a state update.

For example:

```
const doSomethingPersonal = ({expanded}) => {
  // do something really important after being expanded
}
<Expandable onExpanded={doSomethingPersonal}>
 ...
</Expandable>
```

In this example, we assume that after the `Expandable` component's `expanded` state property is toggled, the `onExpanded` prop will be invoked – hence calling the user's callback, `doSomethingPersonal`.

We will add this functionality to the `Expanded` component.

With class components this is pretty straightforward. With functional components, we need to do a little more work – not so much though :)

Whenever you want to perform a side effect within a functional component, for most cases, always reach out for `useEffect`.

So, the easiest solution might look like this:

```
useEffect(() => {
  props.onExpanded(expanded)
}, [expanded])
```

The problem however with this is that the `useEffect` effect function is called at least once – when the component is initially mounted.

So, even though there's a dependency array, `[expanded]`, the callback will also be invoked when the component mounts!

```
useEffect(() => {
  // this function will always be invoked on mount
})
```

The functionality we seek requires that the callback passed by the user isn't invoked on mount.

How can we enforce this?

First, consider the naive solution below:

```
//faulty solution
...
let componentJustMounted = true
useEffect(
    () => {
      if(!componentJustMounted) {
        props.onExpand(expanded)
```

```
        componentJustMounted = false
      }
    },
    [expanded]
  )
...
```

What's wrong with the code above?

Loosely speaking, the thinking behind the code is correct. You keep track of a certain variable `componentJustMounted` and set it to `true`, and only call the user callback `onExpand` when `componentJustMounted` is false.

Finally, the `componentJustMounted` value is only set to `false` after the user callback has been invoked at least once.

Looks good.

However, the problem with this is that whenever the function component re-renders owing to a state or prop change, the `componentJustMounted` value will always be reset to `true`. Thus, the user callback `onExpand` will never be invoked as it is only invoked when `componentJustMounted` is falsey.

```
...
if (!componentJustMounted) {
    onExpand(expanded)
}
...
```

Well, the solution to this is simple. We can use the `useRef` hook to ensure that a value stays the same all through lifetime of the component.

Here's how it works:

```
//correct implementation
  const componentJustMounted = useRef(true)
  useEffect(
```

```
    () => {
      if (!componentJustMounted.current) {
        onExpand(expanded)
      }
     componentJustMounted.current = false
    },
    [expanded]
  )
```

`useRef` returns a `ref` object, and the value stored in the object may be retrieved from the current property, `ref.current`

The signature for `useRef` looks like this: `useRef(initialValue)`.

Hence, stored initially in `componentJustMounted.current` is a ref object with the `current` property set to `true`.

```
const componentJustMounted = useRef(true)
```

After invoking the user callback, we then update this value to `false`.

```
componentJustMounted.current = false
```

Now, whenever there's a state or prop change the value in the ref object isn't tampered with. It remains the same.

With the current implementation, whenever the `expanded` state value is toggled, the user callback function `onExpanded` will be invoked with the current value of `expanded`.

Here's what the final implementation of the `Expandable` component now looks like:

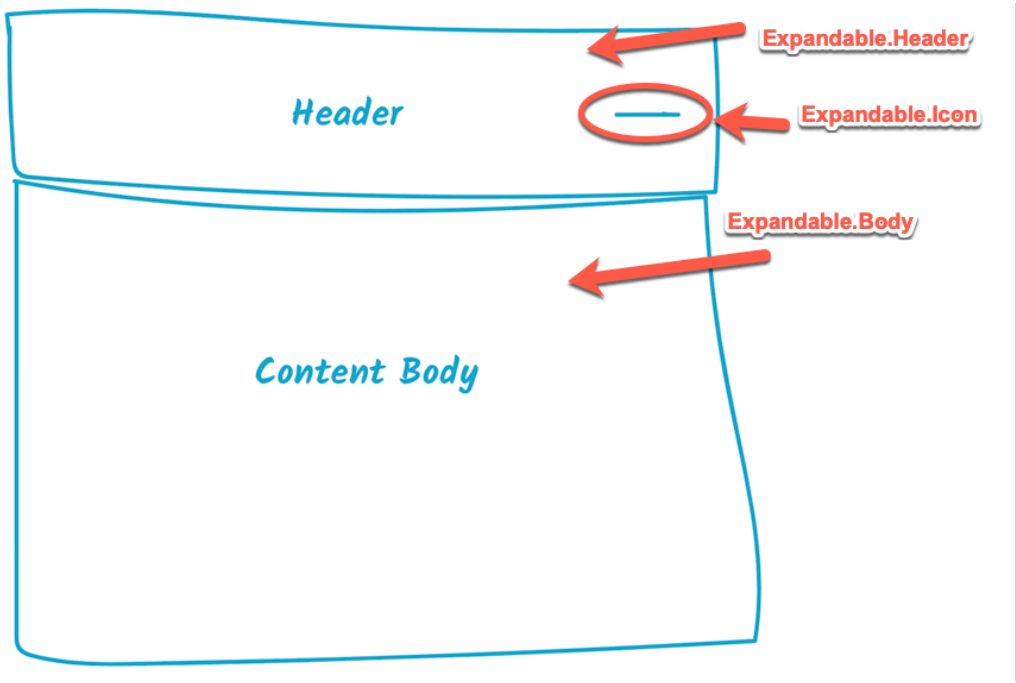
```
// Expandable.js
const Expandable = ({ children, onExpand }) => {
  const [expanded, setExpanded] = useState(false)
  const toggle = useCallback(
```

```
    () => setExpanded(prevExpanded => !prevExpanded),
    []
  )
  const componentJustMounted = useRef(true)
  useEffect(
    () => {
      if (!componentJustMounted) {
        onExpand(expanded)
      }
     componentJustMounted.current = false
    },
    [expanded]
  )
  const value = useMemo(
   () => ({ expanded, toggle }),
   [expanded, toggle]
  )
  return (
    <Provider value={value}>
        {children}
    </Provider>
  )
}
```

If you've followed along, that's great. We've sorted out the most complex component in the bunch. Now, let's build the child components.

Building the Compound Child Components

There are three child components for the `Expandable` component.



These child components need to consume values from the context object created in `Expandable.js`.

To make this possible, we'll do a little refactoring as shown below:

```
export const ExpandableContext = createContext()
```

We export the context object, `ExpandableContext` from `Expandable.js`.

Now, we may use the `useContext` hook to consume the values from the `Provider`.

Below's the `Header` child component fully implemented.

```
//Header.js
import React, { useContext } from 'react'
import { ExpandableContext } from './Expandable'

const Header = ({children}) => {
  const { toggle } = useContext(ExpandableContext)
  return <div onClick={toggle}>{children}</div>
}
```

```
export default Header
```

Simple, huh?

It renders a `div` whose `onClick` callback is the `toggle` function for toggling the `expanded` state within the `Expandable` parent component.

Here's the implementation for the `Body` child component:

```
// Body.js
import { useContext } from 'react'
import { ExpandableContext } from './Expandable'

const Body = ({ children }) => {
  const { expanded } = useContext(ExpandableContext)
  return expanded ? children : null
}
export default Body
```

Pretty simple as well.

The `expanded` value is retrieved from the context object and used within the rendered markup. It reads like this: If expanded, render `children` else render nothing.

The `Icon` component is just as simple.

```
// Icon.js
import { useContext } from 'react'
import { ExpandableContext } from './Expandable'

const Icon = () => {
  const { expanded } = useContext(ExpandableContext)
  return expanded ? '–' : '+'
}
```

```
export default Icon
```

It renders either + or – depending on the value of `expanded` retrieved from the context object.

With all child components built, we can set the child components as `Expandable` properties. See below:

```
import Header from './Header'
import Icon from './Icon'
import Body from './Body'

const Expandable = ({ children, onExpand }) => {
    ...
}
// Remember this is just a personal reference. It's not mandatory
Expandable.Header = Header
Expandable.Body = Body
Expandable.Icon = Icon
```

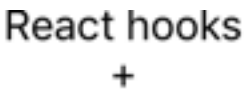
Now we can go ahead to use the `Expandable` component as designed:

```
<Expandable>
    <Expandable.Header>React hooks</Expandable.Header>
    <Expandable.Icon />
    <Expandable.Body>Hooks are awesome</Expandable.Body>
</Expandable>
```

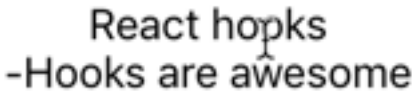
Does it work?

You bet!

Here's what's rendered when not expanded:



And when expanded:



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" />
```

Now, our goal isn't just styling the component, but to make it as reusable as possible. This means letting whoever consumes the component style the component whichever they want i.e using inline styles via the `style` prop, or by passing some `className` prop.

Let's begin with the `Header` child component:

```
// before
const Header = ({children}) => {
  const { toggle } = useContext(ExpandableContext)
  return <div onClick={toggle}>{children}</div>
}
```

First, let's change the rendered markup to a `button`. It's a more accessible and semantic alternative to the `div` used earlier.

```
const Header = ({children}) => {
  const { toggle } = useContext(ExpandableContext)
  // look here 👇
  return <button onClick={toggle}>{children}</button>
}
```

We will now write some default styles for the `Header` component in a `Header.css` file.

```
// Header.css
.Expandable-trigger {
    background: none;
    color: hsl(0, 0%, 13%);
    display: block;
    font-size: 1rem;
    font-weight: normal;
    margin: 0;
    padding: 1em 1.5em;
    position: relative;
```

```
    text-align: left;
    width: 100%;
    outline: none;
    text-align: center;
  }

  .Expandable-trigger:focus,
  .Expandable-trigger:hover {
    background: hsl(216, 94%, 94%);
  }
```

I'm sure you can figure out the simple CSS above. If not, don't stress it. What's important is to note the default `className`used here, `.Expandable-trigger`

To apply these styles, we need to import the `CSS` file and apply the appropriate `className` prop to the rendered `button`.

```
...
import './Header.css'
const Header = () => {
  const { toggle } = useContext(ExpandableContext)
  return <button onClick={toggle}
            className="Expandable-trigger">
    {children}
</button>
}
```

This works great, however the `className` is set to the default string `Expandable-trigger`.

This will apply the styling we've written in the `CSS` file, but it doesn't take into the account any `className` prop passed in by the user.

It's important to accommodate passing this `className` prop as a user might like to change the default style you've set in your `CSS`.

Here's one way to do this:

```
// Header.js
import './Header.css'
const Header = ({ children, className}) => {
  // look here 👇
  const combinedClassName = `Expandable-trigger ${className}`
  return (
    <button onClick={toggle} className={combinedClassName}>
      {children}
    </button>
  )
}
```

Now, whatever `className` is passed to the `Header` component will be combined with the `default Expandable-trigger` before been passed on to the rendered `button` element.

Let's consider how good the current solution is.

First, if the `className` prop is `null` or `undefined`, the `combinedClassName` variable will hold the value `"Expandable-trigger null"` or `"Expandable-trigger undefined".`

To prevent this, be sure to pass a default `className` by using the ES6 default parameters syntax as shown below:

```
// note how className defaults to an empty string
const Header = ({ children, className = '' }) => {
  ...
}
```

Having provided a default value, if the user still doesn't enter a `className`, the `combinedClassName` value will be equal to `"Expandable-trigger "`.

Note the empty string appended to the `Expandable-trigger`. This is owing to how template literals work.

My preferred solution is to do this:

```
const combinedClassName = ['Expandable-trigger', className].join('')
```

This solution handles the previously discussed edge cases. If you also want to explicit about removing `null`, `undefined` or any other falsey values, you can do the following:

```
const combinedClassName = ['Expandable-trigger',
className].filter(Boolean).join('')
```

I'll stick with the simpler alternative, and providing a default for `className` via default parameters.

With that being said, here's the final implementation for `Header`:

```
// after
...
import './Header.css'
const Header = ({ children, className = ''}) => {
  const { toggle } = useContext(ExpandableContext)
  const combinedClassName = ['Expandable-trigger', className].join('')
  return (
    <button onClick={toggle} className={combinedClassName}>
      {children}
    </button>
  )
}
```

So far, so good.

Incase you were wondering, `combinedClassName` returns a string. Since strings are compared by value, there's no need to memoize this value with `useMemo`.

So far, we've graciously handled the `className` prop. How about the option to override default styles by passing a `style` prop?

Well, let's fix that.

Instead of explicitly destructuring the `style` prop, we can pass on any other prop passed by the user to the `button` component.

```
// rest paramter ...otherProps 👇
const Header = ({ children, className = '', ...otherProps }) => {
return (
    // spread syntax {...otherProps} 👇
    <button {...otherProps}>
      {children}
    </button>
  )
}
```

Note the use of the rest parameter and spread syntax.

With this done, the `Header` component receives our default styles, yet allows for change via the `className` or `style` props.

```
// override style via className
<Expandable.Header className="my-class">
    React hooks
</Expandable.Header>

// override style via style prop
<Expandable.Header style={{color: "red"}}>
    React hooks
</Expandable.Header>
```

Now, I'll go ahead and do the same for the other child components, `Body` and `Icon`.

```
// before
const Body = ({ children }) => {
  const { expanded } = useContext(ExpandableContext)
  return expanded ? children : null
}

// after
import './Body.css'
const Body = ({ children, className = '', ...otherProps }) => {
  const { expanded } = useContext(ExpandableContext)
  const combinedClassNames = ['Expandable-panel', className].join('')

  return expanded ? (
    <div className={combinedClassNames} {...otherProps}>
      {children}
    </div>
  ) : null
}

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

Do the same for `Icon` component:

```
// before
const Icon = () => {
  const { expanded } = useContext(ExpandableContext)
```

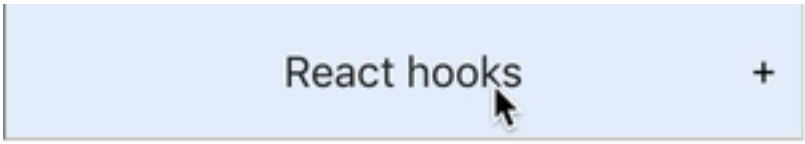
```
  return expanded ? '–' : '+'
}

// after
...
import './Icon.css'
const Icon = ({ className = '', ...otherProps }) => {
  ...
  const combinedClassNames = ['Expandable-icon', className].join('')

  return (
    <span className={combinedClassNames} {...otherProps}>
      {expanded ? '–' : '+'}
    </span>
  )
}

// Icon.css
.Expandable-icon {
    position: absolute;
    top: 16px;
    right: 10px;
}
```

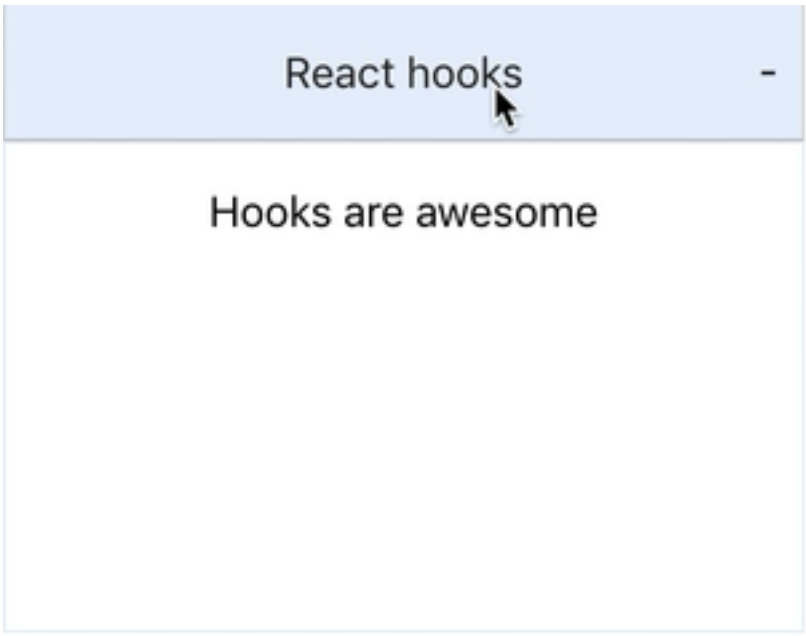
And finally, some styles for the parent component, `Expandable`.

```
import './Expandable.css'
const Expandable = ({ children, onExpand, className =
'', ...otherProps }) => {
  ...
  const combinedClassNames = ['Expandable', className].join('')
  return (
```

```
    <Provider value={value}>
      <div className={combinedClassNames} {...otherProps}>
        {children}
      </div>
    </Provider>
  )
}

// Expandable.css
.Expandable {
     position: relative;
     width: 350px;
}
```

Now we've got a beautiful reusable component!

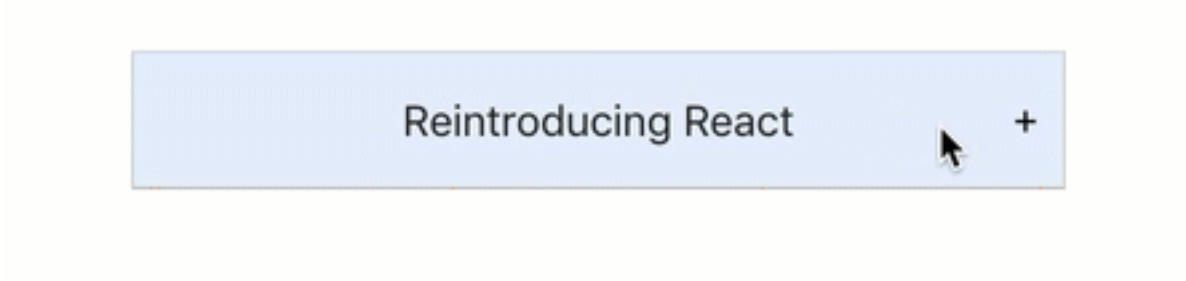


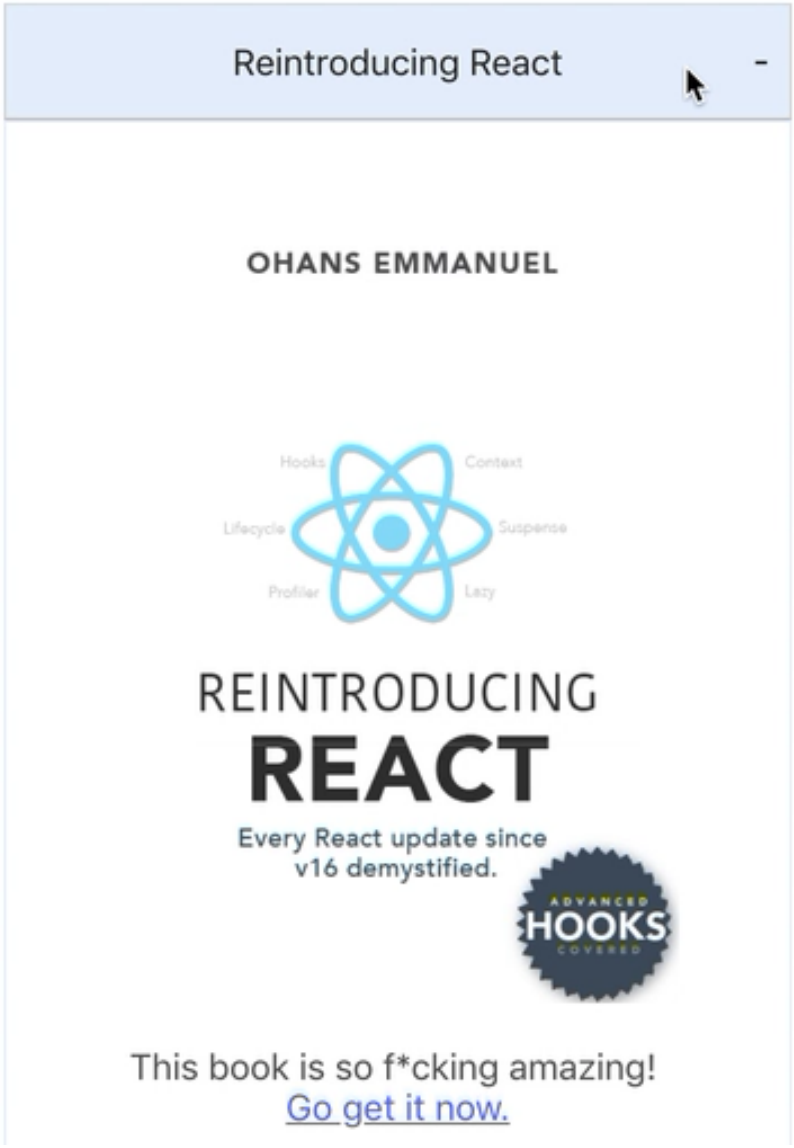


We've not just made it beautiful, but it's customisable as well.

How customisable is the component we've built?

See what I've done below with the same component!





And this didn't take a lot of code.

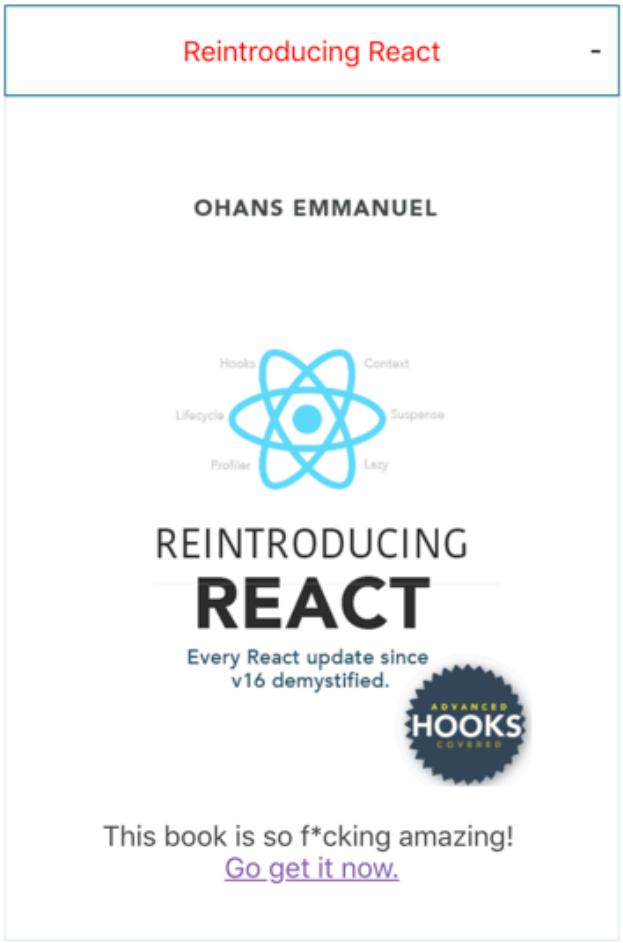
```
<Expandable>
    <Expandable.Header>Reintroducing React</Expandable.Header>
    <Expandable.Icon />
    <Expandable.Body>
        <img
            src='https://i.imgur.com/qpj4Y7N.png'
            style={{ width: '250px' }}
            alt='reintroducing react book cover'
```

```
        />
        <p style={{ opacity: 0.7 }}>
          This book is so f*cking amazing! <br />
        <a
          href='https://leanpub.com/reintroducing-react'
          target='_blank'
          rel='noopener noreferrer'
          >
            Go get it now.
        </a>
       </p>
     </Expandable.Body>
</Expandable>
```

You can go one step further to test if overriding styles via the `style` prop works as well.

```
<Expandable>
   <Expandable.Header
      // look here 👇
      style={{ color: 'red', border: '1px solid teal' }}>
        Reintroducing React
    </Expandable.Header>
        ...
</Expandable>
```

And below's the result of that:



Yay! it works as expected.

Control Props

The compound component we built in the last section works great, however, it's quite limited in its extensibility.

Let me show you what I mean.

Assume another developer, which we'll refer to as "user", decided to use the component we've built, but in a much different way.

This user has found some very good information they think is worth sharing. So within their app, they store this information in an array:

```
// user's app
const information = [
  {
    header: 'Why everyone should live forrever',
    note: 'This is highly sensitive information ... !!!!'
  },
  {
    header: 'The internet disappears',
    note:
      'I just uncovered the biggest threat...'
  },
  {
    header: 'The truth about Elon musk and Mars!',
    note: 'Nobody tells you this...'
  }
]
```

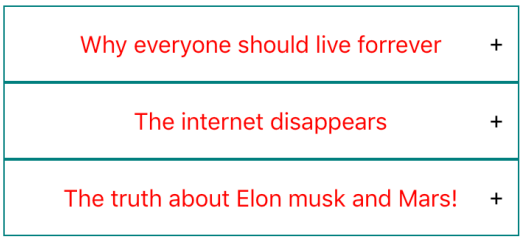
In their `App`, they loop over the list of information and render our `Expandable` component with its children as seen below:

```
// user's app
function App () {
  return (
    <div className='App'>
      {information.map(({ header, note }, index) => (
        <Expandable key={index}>
          <Expandable.Header
            style={{ color: 'red', border: '1px solid teal' }}>
            {header}
          </Expandable.Header>
```

```
            <Expandable.Icon />
            <Expandable.Body>{note}</Expandable.Body>
          </Expandable>
        ))}
      </div>
    )
}
```

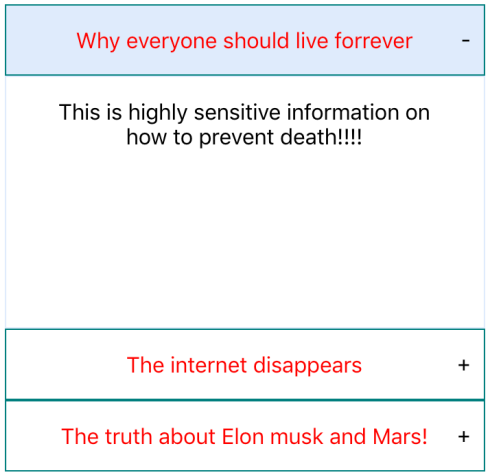
Nothing out of the ordinary here.

Below's the result the user gets when no header has been clicked.



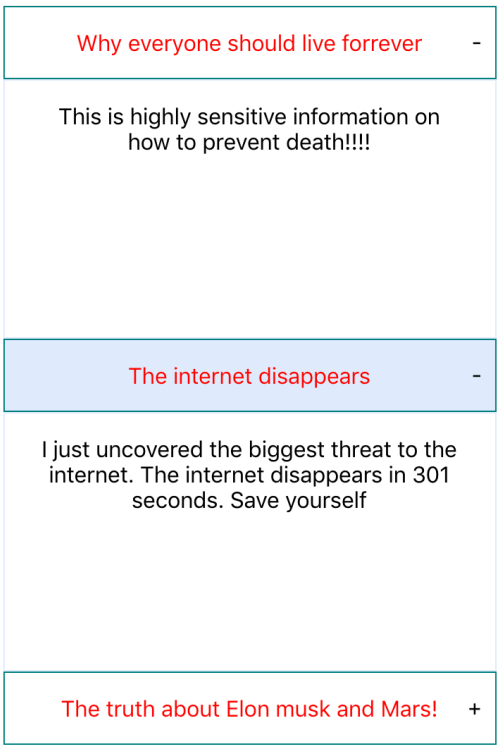
Looks good!

When the user clicks any of the headers, the content expands as designed. In the graphic below, you can see the result of clicking just one of the headers.



What happens when another header is clicked?

Well, the header expands as well. This is exactly how the internals of our `Expandable` component works.



As you can see from the graphic above, both clicked headers expand their content. Sadly, this is not the behaviour the user seeks.

What the user really wants is a behaviour similar to an accordion — *an accordion only expand one container at a time*. Never expand more than one. If the user clicks another header, open that and but close the rest.

So, how do extend the `Expandable` component to handle this use case?

Well, this is where the control props pattern comes into play. Note that I'm using the compound component we built as an example, in actuality, you can implement this functionality in any component you wish.

How Control Props Work

Consider a simple input field.

```
<input />
```

In React, a controlled input element will be defined like this:

```
<input value={someValue} onChange={someHandler} />
```

Now, that's different?

A controlled input component has its `value` and `onChange` handler managed **externally**.

This is exactly what we aim for when implementing the control props pattern. We want the state managed internally to be manageable from the outside via props!

The implementation is quite simple, but I need you to understand how it works before we delve into writing some code.

The default usage of the `Expandable` is as shown below:

```
<Expandable>
   <Expandable.Header>
       {someHeader}
    </Expandable.Header>
    <Expandable.Icon />
```

```
    <Expandable.Body>{someContent}</Expandable.Body>
 </Expandable>
```

Within the `Expandable` component, the `expanded` value is internally managed and communicated to the children.

If we wanted to make`Expandable` a controlled component, then we need to accept a prop which defines the `expanded` value externally.

For example:

```
<Expandable shouldExpand={someUserDefinedBooleanValue}>
   <Expandable.Header>
       {someHeader}
    </Expandable.Header>
    <Expandable.Icon />
    <Expandable.Body>{someContent}</Expandable.Body>
 </Expandable>
```

Internally, if the `shouldExpand` prop is passed, we will give up control of the state update to the user i.e let the expand value return whatever the user defines in the `shouldExpand` prop.

Now, let's move on to the actual technical implementation.

First, we will check if the control prop `shouldExpand` exists. If it does, then the component is a controlled component i.e `expanded` managed externally.

```
// Expandable.js
const Expandable = ({
  shouldExpand, // 👈 take in the control prop
  ...otherProps
}) => {
  // see check 👇
  const isExpandControlled = shouldExpand !== undefined
  ...
```

```
}
```

Not everyone would pass in the control prop. It's possible that a lot of users just go for the default use case of letting us handle the state internally. This is why we check to see if the control prop is defined.

Now, the `isExpandControlled` variable is going to define how we handle a couple other things internally.

It all begins with the value the `Expandable` component communicates to the child elements.

```
// before
 const value = useMemo(() => ({ expanded, toggle}), [
    expanded, toggle
  ])
```

Before now, we passed along the `expanded` and `toggle` values from the internals of the `Expandable` component.

Now, we can't do this.

Remember, if the component is controlled, we want `expanded` to return the `shouldExpand` prop passed in by the user.

We can extract this behaviour into a `getState` variable defined below:

```
// Expanded.js
...
const getState = isExpandControlled ? shouldExpand : expanded
...
```

Where `shouldExpand` is the user's control prop, and `expanded` our internal state.

Also, remember that a controlled input element is passed the value prop and **onChange** prop. A controlled component receives both the state value and the function handler as well.

Before now, we had an internal `toggle` function.

```
// before. See toggle below 👇
 const toggle = useCallback(
    () => setExpanded(prevExpanded => !prevExpanded),
    []
  )
```

Now, we can introduce a separate variable `getToggle` defined below:

```
// Expandable.js
 ...
  const getToggle = isExpandControlled ? onExpand : toggle
...
```

If the component is controlled, we return the `onExpand` prop – a user defined callback. If not, we return our internal implementation, `toggle`.

With these done, here's the value communicated to the child elements:

```
...
const value = useMemo(() => ({ expanded: getState, toggle: getToggle }),
[
    getState,
    getToggle
  ])
```

Note how `getState` and `getToggle` are both used to get the desired `expanded` state and `toggle` function depending on whether the component is controlled or not.

We're pretty much done implementing the control prop pattern here.

There's one more cleanup to be performed.

When the component wasn't controlled, the `onExpand` prop was a user defined function invoked after every state change. That's not going to be relevant if the

component is controlled. Why? The user handles the toggle function themself when the component is controlled. They can choose to do anything in the callback with the already controlled state value too.

Owing to the explanation above, we need to make sure the `useEffect` call isn't run when the component is controlled.

```
// Expandable.js
...
useEffect(
    () => {
        // run only when component is controlled.
        // see !isExpandControlled  👇
      if (!componentJustMounted && !isExpandControlled) {
        onExpand(expanded)
        componentJustMounted.current = false
      }
    },
    [expanded, onExpand, isExpandControlled]
  )
...
```

And that is it!

We've implemented the control props pattern.

Huh, but I have not shown you how it solves the user's problem. That's next.

Using the Controlled Component

We reached out to the user and let them know we have implemented a pattern to cater for their specific use case.

Out of excitement they get on to implement the change.

Here's how.

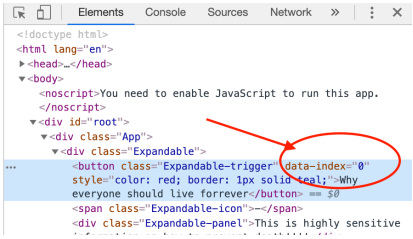
Within the user app, they send a `data-index` prop to every `Header` element.

```
// before
<Expandable.Header />
// now
<Expandable.Header data-index={index} />
```

Where index is retrieved from the iteration index.

```
function App () {
  ...
                // see index below 👇
 {information.map(({ header, note }, index) => (
   ...
 )}
}
```

With this, the returned element from the header gets a `data-index` property.



How does this help their cause?

Well, here's the full usage of the controlled component.

```
// user's app
function App () {
  const [activeIndex, setActiveIndex] = useState(null)
  const onExpand = evt => setActiveIndex(evt.target.dataset.index)
```

```
  return (
    <div className='App'>
      {information.map(({ header, note }, index) => (
        <Expandable
          shouldExpand={index === +activeIndex}
          onExpand={onExpand}
          key={index}
        >
          <Expandable.Header
            style={{ color: 'red', border: '1px solid teal' }}
            data-index={index}
          >
            {header}
          </Expandable.Header>
          <Expandable.Icon />
          <Expandable.Body>{note}</Expandable.Body>
        </Expandable>
      ))}
    </div>
  )
}
```

Note the `shouldExpand` controlled prop being the result of `index === +activeIndex`.

`activeIndex` is a state variable within the user app. Whenever any header is clicked, the `activeIndex` is set to the `data-index` of the element.

This way the user knows which header was just clicked i.e should be active at that given time.

This explains why they've defined the control prop `shouldExpand` as shown below:

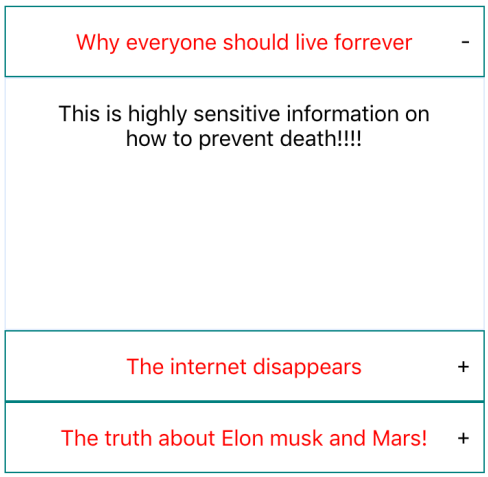
```
shouldExpand={index === +activeIndex} // 👈 the "+" converts the
activeIndex to a number
```

Also note the user's `onExpand` callback.

```
const onExpand = evt => setActiveIndex(evt.target.dataset.index)
```

This is what's invoked on every click NOT our custom `toggle` function. When the user clicks an header, the `activeIndex` state variable is set to the `data-index` from the click target.

With this the user satisfactorily implements their feature. No two contents are expanded at the same time.



Be sure to check out the full the source code in the `control-props` directory.

Great job so far!

With our controlled props API, we've made it relatively for the user to control the state of the `Expanded` component. How convenient!

Building a Custom Hook

Before we go into the other advanced patterns, it's important to understand the default way of sharing functionality with hooks – building a custom hook.

Only by building on this foundation can we take advantage of the other advanced patterns to be discussed.

So far we've built a compound component that works great! Let's say you were the author of some open-source library, and you wanted to expose the "expand" functionality via a custom hook, how would you do this?

First, let's agree on the name of your open-source (OS) library. Uh, we'll leave it as `Expandable` – same as before.

Now, instead of having the logic for managing the `expanded` state in a `Expandle` component that returns a `Provider`, we can get rid of most of that.

Now, we're just going to export 2 custom hooks.

```
// Expandable.js
import useExpanded from '../useExpanded'
import useEffectAfterMount from '../useEffectAfterMount'

export { useExpanded as default, useEffectAfterMount }
```

The `useExpanded` custom hook will now handle the logic for the `expanded` state variable, and `useEffectAfterMount` the logic for invoking a callback only after mount.

So, shall we write these custom hooks?

Note that these custom hooks will pretty much use the same logic as before I.e within the `Expandable` compound component we had written earlier. The difference here will be wrapping these in a custom hook.

Here's the `useExpanded` custom hook:

```
// useExpanded.js
```

```
import { useCallback, useMemo, useState } from 'react'


export default function useExpanded () {
  const [expanded, setExpanded] = useState(false)
  const toggle = useCallback(
    () => setExpanded(prevExpanded => !prevExpanded),
    []
  )


  const value = useMemo(() => ({ expanded, toggle }), [expanded,
toggle])


  return value
}
```

I'm not going to spend time discussing the internal logic of this custom hook. I did so in the section on compound components.

What's important is that we've wrapped this functionality in a `useExpanded` function (aka custom hook) which returns the `value` we previously had in a context `Provider`.

We'll do similar with the `useEffectAfterMount` custom hook as shown below:

```
// useEffectAfterMount.js
import { useRef, useEffect } from 'react'


export default function useEffectAfterMount (cb, deps) {
  const componentJustMounted = useRef(true)
  useEffect(() => {
    if (!componentJustMounted) {
      cb()
      componentJustMounted.current = false
    }
```

```
  }, deps)
}
```

The only difference here is that `useEffectAfterMount` doesn't return any value. Rather it invokes the `useEffect` hook. To make this as generic as possible, the custom hook takes in two arguments, the callback to be invoked after mount, and the array dependencies for which the `useEffect` function relies upon.

Also, note that line 7 reads, `return cb()`. This is to make sure to handle unsubscriptions by returning whatever is returned by the callback – if any e.g a function to handle unsubscriptions.

Great!

Now that you've built these custom hooks, how would a typical consumer of your OS library use these hooks?

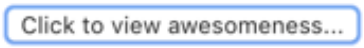
Here's one simple example:

```
// User's app
import useExpanded from './components/Expandable'

function App () {
  const { expanded, toggle } = useExpanded()
  return (
    <div style={{ marginTop: '3rem' }}>
      <button onClick={toggle}>Click to view awesomeness...</button>
      {expanded ? <p>{'😎'.repeat(50)}</p> : null}
    </div>
  )
}
```

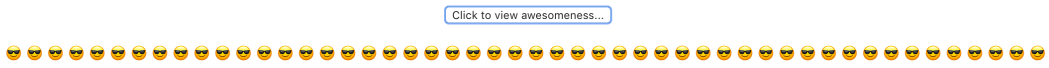
The consumer imports your `useExpanded` custom hook and invokes the function to retrieve `expanded` and `toggle`.

With these values, they can render whatever they want. We provide the logic, and let the user render whatever UI they deem fit. Here's the result of what this user chose to do:

By default they render a button:



And this button invokes the exposed `toggle` function from the custom hook. Based on the toggled `expanded` state, they render the following:



Well, this is the user's version of an expandable container.

Do you realise what we've done here?

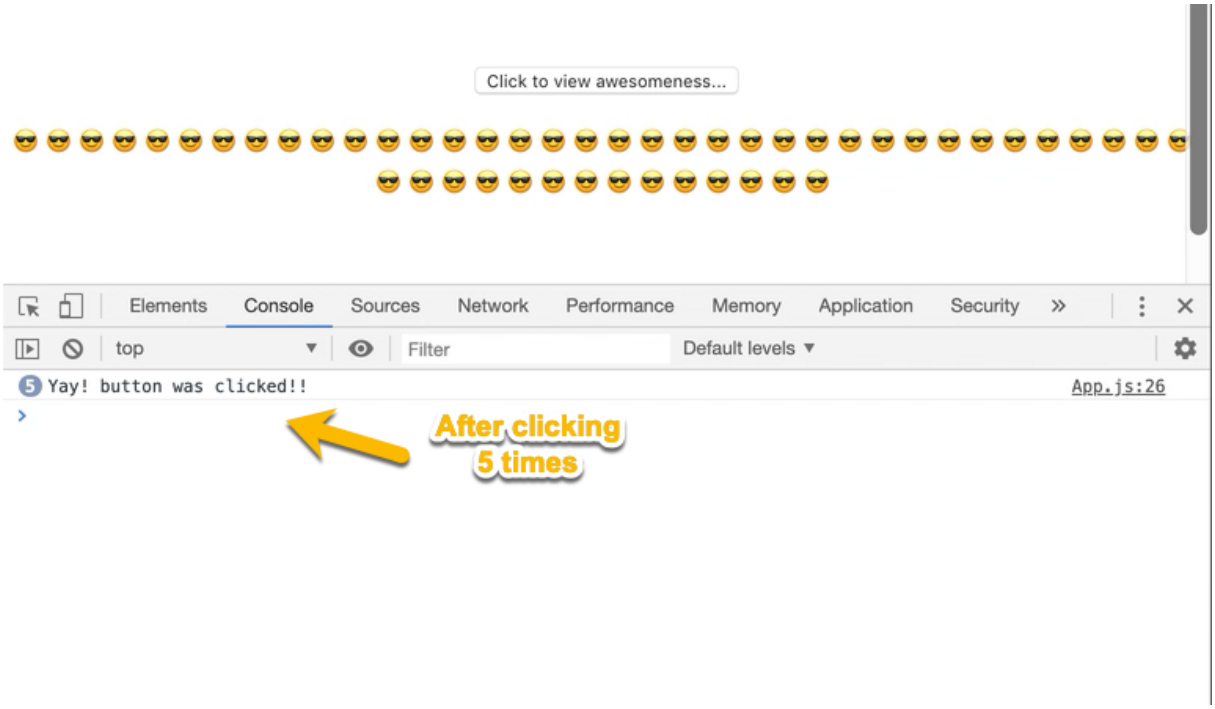
Like the render props API, we've given control over to the user to render whatever UI they want., while we handle whatever logic is required for them to do this.

To use the the `useEffectAfterMount`hook, they just do this:

```
// User's app
function App () {
  const { expanded, toggle } = useExpanded()
  // look here 👇
  useEffectAfterMount(
    () => {
      // user can perform any side effect here 👇
      console.log('Yay! button was clicked!!')
    },
```

```
    [expanded]
  )
  return (
    <div style={{ marginTop: '3rem' }}>
      <button onClick={toggle}>Click to view awesomeness...</button>
      {expanded ? <p>{'😎'.repeat(50)}</p> : null}
    </div>
  )
}
```

Now, whenever the button is clicked, they'll indeed get the logs!



Providing Default UI Elements

If you were building such an OS library, as opposed to just exporting these custom hooks, it may be nice to also export some default UI elements.

For convenience, if the user is reaching out for a `Expandable` library, they need the custom logic you've extracted into a reusable hook, yes, but they also need to render some UI elements.

You could also export some default `Header`, `Body` and `Icon` elements built to have their styling configurable.

This sounds like we can reuse the components we built earlier, however we need to do some refactoring.

The child elements for the compound component required access to some context object. We may get rid of that and solely base the contract on props.

```
// Body.js (before)
const Body = ({ children, className = '', ...otherProps }) => {
  ...
  // look here 👇
  const { expanded } = useContext(ExpandableContext)
  return ...
}
// now (takes in expanded as props)
const Body = ({
    children, className = '', expanded, ...otherProps }) => {
    ...
  return ...
}
```

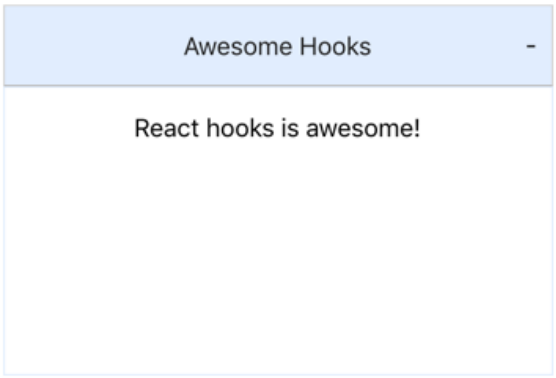
The same may be done for the `Icon` and `Header` components.

By providing these default UI elements, you allow the consumer of your library the option of not having to bother about UIs as much.

```
import Header from './components/Header'
import Icon from './components/Icon'
import Body from './components/Body'
```

```
// user's app
function App () {
  const { expanded, toggle } = useExpanded()
  return (
    <div className='Expandable'>
      <Header toggle={toggle}>Awesome Hooks </Header>
      <Icon expanded={expanded} />
      <Body expanded={expanded}>React hooks is awesome!</Body>
    </div>
  )
}
```

Which results in this:



It's important to note that you don't always have to do this. You can always share custom functionality by just writing a custom hook.

We've done a brilliant job building this imaginary OS library, huh? In the following sections we'll incorporate even more advanced patterns, and users of the library will find them incredibly useful.

Props Collection

Regardless of the component you build, and even though it's going to be used by different people, some things will remain the same across all users of the component.

How can we know this?

Well, consider the the `Expandable` component we've worked so hard at.

First, regardless of the UI solution the user decides to go for e.g. a custom UI, they'll always need an `onClick` handler for the "toggle element".

```
const { toggle, expanded } = useExpanded()

// user 1.

<button onClick={toggle}>Click to view awesomeness...</button>

// user 2.

<div onClick={toggle}> Burn the place down 🔥</div>

// user 3.

<img onClick={toggle} alt="first image in carousel"/>
```

You get the point.

Every user, regardless of the element they choose to render, still needs to handle the `onClick` callback.

Also, if these users care about accessibility at all, they also will do this:

```
const { toggle, expanded } = useExpanded()

// user 1.

<button onClick={toggle} aria-expanded={expanded}>Click to view
awesomeness...</button>
```

```
// user 2.
<div onClick={toggle} aria-expanded={expanded}> Burn the place down 🔥</
div>
// user 3.
<img onClick={toggle} aria-expanded={expanded} alt="first image in
carousel"/>
```

Now, that's two props.

Depending on your use case there could be more "common props" associated with your reusable hook or component.

So, what's a props collection?

> *A props collection is basically a collection of "common props" associated with a custom hook or component.*

Is there a way we can prevent the users from writing these props every single time? Can we expose some API from within your reusable custom hook or component?

This is the question the props collection pattern answers with a resounding yes!

Take a look at the props collection for the `useExpandable` 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 `useExpandable` 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.

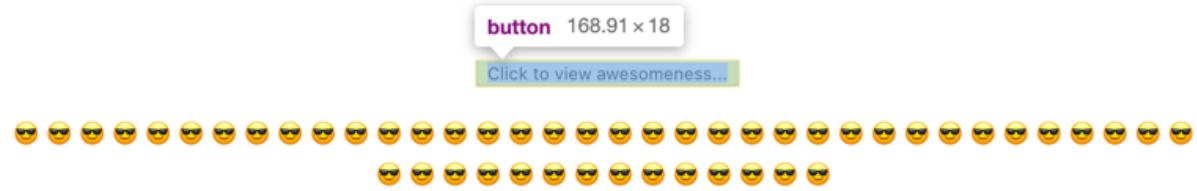
```
export default function useExpanded () {
  ...
  // look here - togglerProps has been included 👇
  const value = useMemo(() => ({ expanded, toggle, togglerProps }), [
    expanded,
    toggle,
    togglerProps
  ])

  return value
}
```

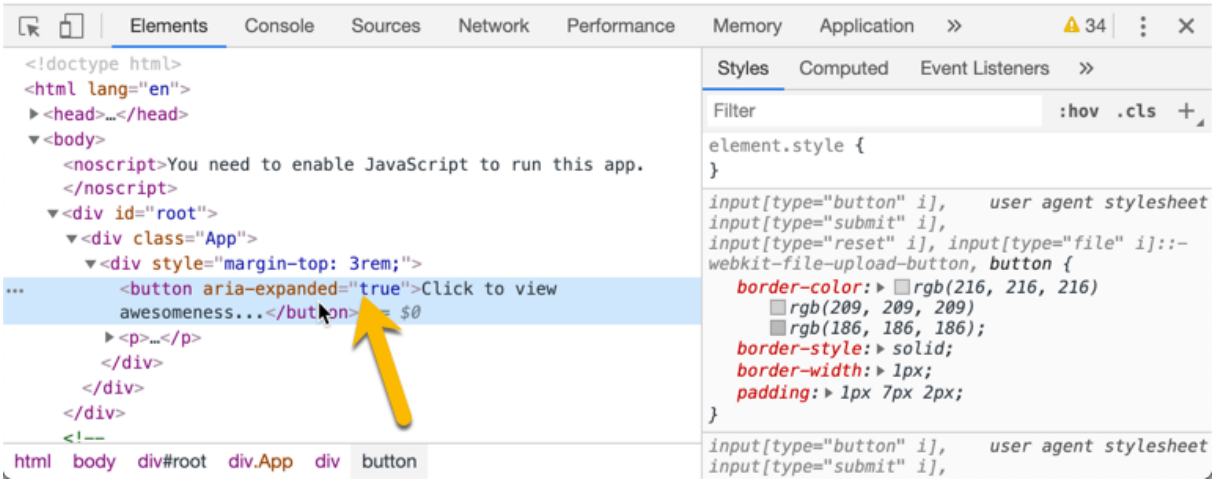
With `togglerProps` now exposed, it may be consumed by any user as shown below:

```
// user's app
function App () {
  const { expanded, togglerProps } = useExpanded()
```

```
  ...
  return (
    <div style={{ marginTop: '3rem' }}>
       // look here 👇
      <button {...togglerProps}>Click to view awesomeness...</button>
      {expanded ? <p>{'😎'.repeat(50)}</p> : null}
    </div>
  )
}
```

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





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

Prop Getters

In JavaScript, all functions are objects. However, functions are built to more customizable and reusable.

Consider the following:

```
const obj = {name: "React hooks"}
```

If you wanted to created the same object but allow for some level of customizability, you could do this:

```
const objCreator = ({name}) => ({name})
```

Now, this `objCreator` function can then be called multiple times to create different object as shown below:

```
const obj = objCreator("React hooks")
const obj = objCreator("React hooks mastery")
const obj = objCreator("React hooks advanced patterns")
```

Why have I chosen to explain this?

Well, this is the difference between the props collection pattern and prop getters pattern.

While props collection relies on providing an object, prop getters expose functions. Functions that can be invoked to create a collection of props.

Owing to the customisability of functions, using a prop getter allows for some more interesting use cases.

First, let's start by refactoring the previous solution to use a props getter, then we'll consider some interesting use cases.

```
// before
  const togglerProps = useMemo(
    () => ({
      onClick: toggle,
      'aria-expanded': expanded
    }),
    [toggle, expanded]
  )
```

```
// now
  const getTogglerProps = useCallback(
    () => ({
      onClick: toggle,
      'aria-expanded': expanded
    }),
    [toggle, expanded]
  )
```

Instead of `togglerProps`, we now create a memoized function `getTogglerProps` that returns the same props collection.

We'll expose this via the returned `value` variable within the `useExpanded` hook as shown below:

```
export default function useExpanded () {
  ...
  const getTogglerProps = useCallback(
    () => ({
      onClick: toggle,
      'aria-expanded': expanded
    }),
    [toggle, expanded]
  )
  // look here 👇
  const value = useMemo(() => ({ expanded, toggle, getTogglerProps }), [
    expanded,
    toggle,
    getTogglerProps
  ])
  return value
```

```
}
```

Now, we need to update how the props collection is consumed.

```
// before
<button {...togglerProps}>Click to view awesomeness...</button>
// now
<button {...getTogglerProps()}>Click to view awesomeness...</button>
```

And that's it! The user's app works just like before.

I know what you're thinking. If we're not going to pass any arguments to the `getTogglerProps` function, why bother with the refactor?

That's a great question, but we'll pass in some arguments real soon.

Additional User Props

With the prop getter, `getTogglerProps` we can cater for other additional props the user may be interested in passing down to the toggle element.

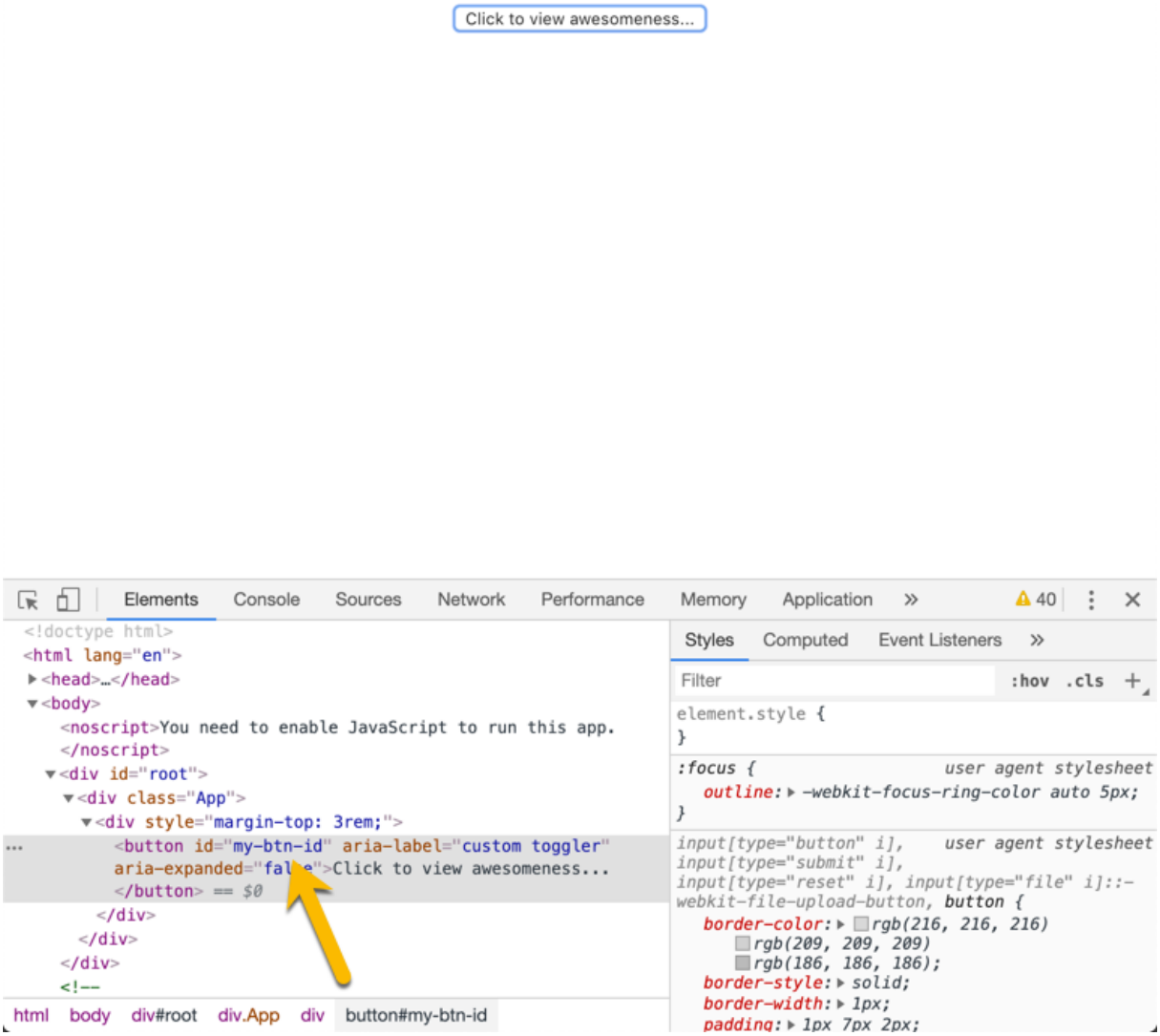
For example, we currently have this usage by the user:

```
<button {...getTogglerProps()}>Click to view awesomeness...</button>
```

If the user needs some more props we haven't catered for in the prop collection, they could do this:

```
<button id='my-btn-id' aria-label='custom
toggler' {...getTogglerProps()}>
        Click to view awesomeness...
      </button>
```

And this works just fine i.e passing `id` and `aria-label`.



However, since these are still part of the "toggler props", though not common enough to make them a part of the default props collection, we could allow for the following usage:

```
<button
        {...getTogglerProps({
          id: 'my-btn-id',
          'aria-label': 'custom toggler'
        })}
      >
```

```
      Click to view awesomeness...
</button>
```

It's arguably more expressive. So, let's extend the `getTogglerProps` function to handle this use case.

```
...
const getTogglerProps = useCallback(
    ({ ...customProps }) => ({
      onClick: toggle,
      'aria-expanded': expanded,
      ...customProps
    }),
    [toggle, expanded]
  )
```

Since the user will invoke `getTogglerProps` with an object of custom props, we could use the `rest` parameter to hold a reference to all the custom props.

```
({ ...customProps }) => ({

})
```

Then we could `spread` over all custom props into the returned props collection.

```
({ ...customProps }) => ({
      onClick: toggle,
      'aria-expanded': expanded,
      // look here 👇
      ...customProps
})
```

With this extension to `getTogglerProps` we've handled the case for custom props not catered for in the default props collection we expose.

How interesting!

In the next section, we'll consider an even more interesting case.

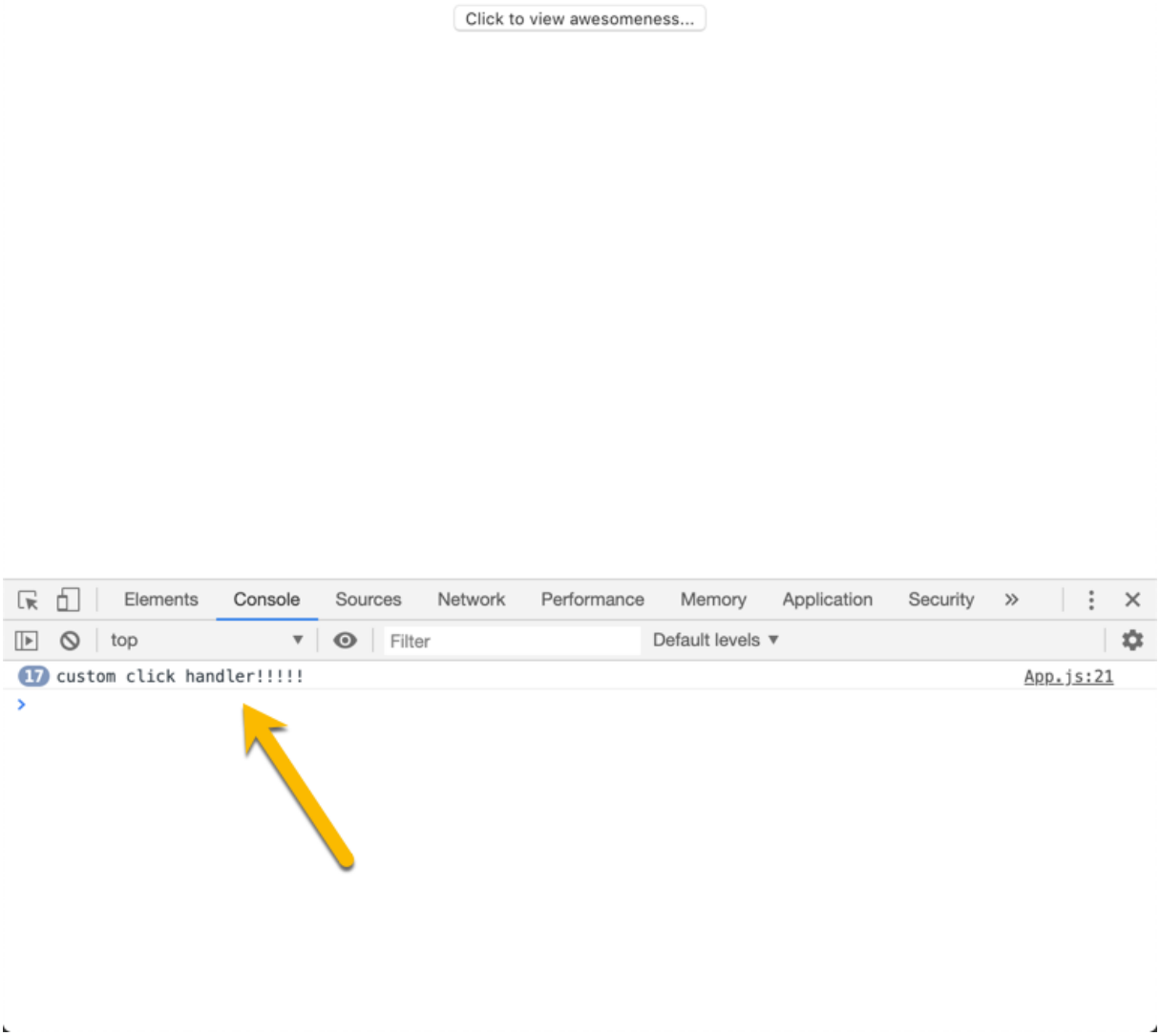
How do you handle props with the same property name?

We allowed for passing custom props into the `getToggler` props getter. Now, what if the user wanted to pass in a custom `onClick` prop?

Here's what I mean:

```
// user's app
...
const customClickHandler = () => {
    console.log('custom click handler!!!!!')
}
...
<button
    {...getTogglerProps({
      id: 'my-btn-id',
      'aria-label': 'custom toggler'
       // look here 👇
      onClick: customClickHandler
     })}
   >
    Click to view awesomeness...
</button>
```

If the user does this, it completely overrides the `onClick` handler within the `useExpanded` custom hook and breaks the functionality of the app.



The button no longer expands the element, but the user's custom click handler is invoked whenever you click the button.

Why's this the case?

This is because we have an `onClick` property in the returned object, but this is overridden by the user's.

```
// useExpandable.js

...
  const getTogglerProps = useCallback(
```

```
    ({ ...customProps }) => ({
      onClick: toggle, // this is overriden
      'aria-expanded': expanded,
      // customProps overrides the onClick above
      ...customProps
    }),
    [toggle, expanded]
  )
```

If you move the position of the internal `onClick` handler, you can override the user's.

```
 const getTogglerProps = useCallback(
    ({ ...customProps }) => ({
      'aria-expanded': expanded,
      ...customProps,
      onClick: toggle // this override's the user's
    }),
    [toggle, expanded]
  )
```

This works. However, we want a component that's as flexible as possible. The goal is to give the user the ability to invoke their own custom `onClick` handler.

How can we achieve this?

Well, we could have the internal `onClick` invoke the user's click handler as well.

Here's how:

```
// useExpanded.js
...
const getTogglerProps = useCallback(
    ({ onClick, ...props } = {}) => ({
      'aria-expanded': expanded,
```

```
      onClick: callFunctionsInSequence(toggle, onClick),
      ...props
    }),
    [toggle, expanded]
  )
...
```

Now instead of returning an object with the `onClick` set to a single function i.e our internal `toggle` function or the user's `onClick`, we could set the `onClick` property to a function, `callFunctionsInSequence` that invokes both functions - our `toggle` function and the user's `onClick`!

```
...
onClick: callFunctionsInSequence(toggle, onClick)
```

I've called this function `callFunctionsInSequence` so it's easy to understand what it does.

Also, note how we destructure the user's `onClick` handler and use the rest parameter to handle the other props passed in. So you don't get an error if no object is passed in by the user, we set a default parameter value of `{}`.

```
...
// note destructuing and default parameter value of {}
({ onClick, ...props } = {}) => ({


})
```

Now back to the elephant in the room. How do we write a function that takes one or two functions and invokes all of them without neglecting the passed arguments?

This function must not break if any of the function arguments is `undefined` too.

Here's one solution to that:

```
const callFunctionsInSequence = (...fns) => (...args) =>
```

```
  fns.forEach(fn => fn && fn(...args))
```

When we do this:

```
...
onClick: callFunctionsInSequence(toggle, onClick) // this invokes
callFunctionsInSequence
```

The return value saved in the `onClick` object property is the returned function from invoking `callFunctionsInSequence`:

```
(...args) => fns.forEach(fn && fn(...args))
```

If you remember from basic React, we always attach click handlers like this:

```
...
render() {
  return <button onClick={this.handleClick} />
}
```

We attach the click handler `handleClick`, but in the `handleClick` declaration, we expect an `evt` argument to be passed in at invocation time.

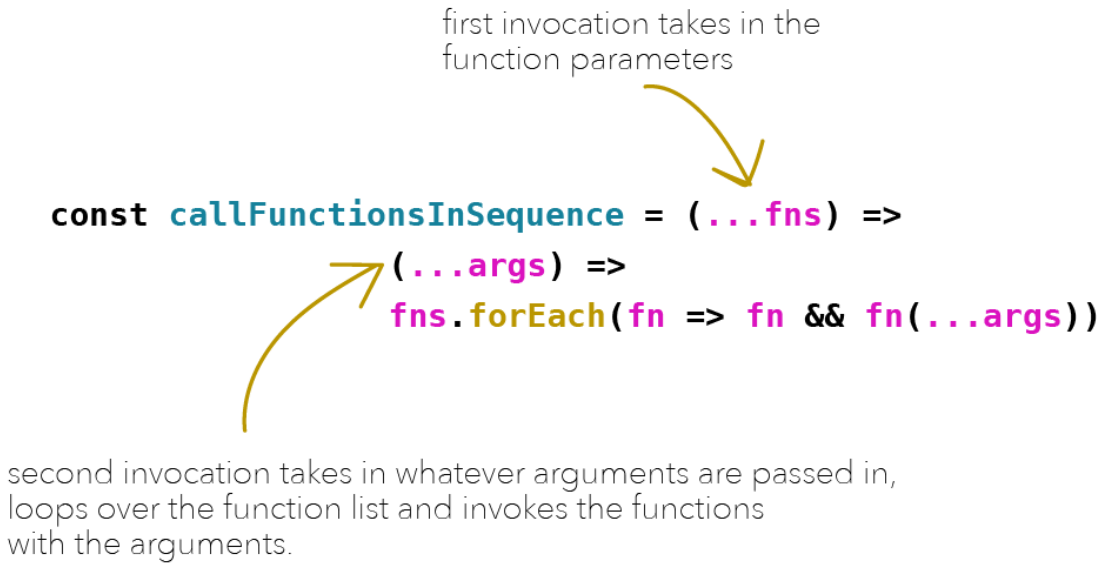
```
// see event object (evt)
handleClick = evt => {
    console.log(evt.target.value)
}
...
```

Now back to the returned function from `callFunctionsInSequence`:

```
(...args) => fns.forEach(fn && fn(...args))
```

The function receives whatever arguments are passed into the function, `(...args)`, and invokes all function parameters with these arguments if the function isn't falsey. `fn && fn(...args)`.

In this case, the argument received by the function will be the event object and it'll be passed down to both `toggle` and `onClick` as arguments.

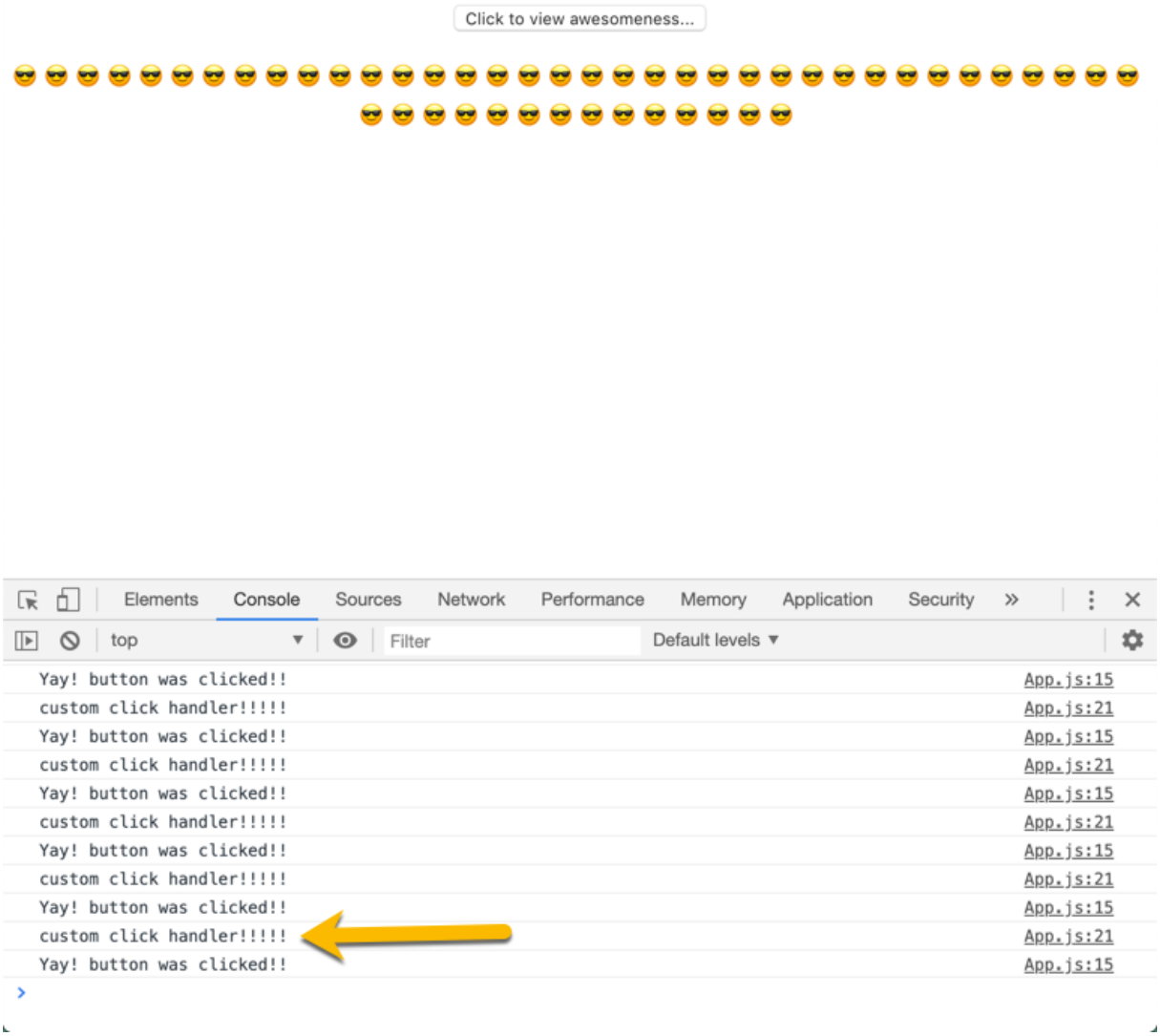


The full implementation of the custom hook now looks like this:

```
// useExpanded.js
const callFunctionsInSequence = (...fns) => (...args) =>
  fns.forEach(fn => fn && fn(...args))

export default function useExpanded () {
  const [expanded, setExpanded] = useState(false)
  const toggle = useCallback(
    () => setExpanded(prevExpanded => !prevExpanded),
    []
  )
  const getTogglerProps = useCallback(
    ({ onClick, ...props } = {}) => ({
      'aria-expanded': expanded,
```

```
      onClick: callFunctionsInSequence(toggle, onClick),
      ...props
    }),
    [toggle, expanded]
  )

  const value = useMemo(() => ({ expanded, toggle, getTogglerProps }), [
    expanded,
    toggle,
    getTogglerProps
  ])

  return value
}
```

With this addition, our internal click handler, `toggle` and the user's custom handler, `onClick` are both invoked when the toggle button is clicked!



How amazing. We wouldn't be able to cater for this case with just a prop collection object. Thanks to prop getters, we have a lot more power to provide very reusable components.

State Initializers

To Initialize means to set the value of something. Going by that definition, the state initializer pattern exists to make it easy for the consumer of your custom hook to set the "value of state".

Note that the state initializer pattern doesn't give full control over setting "value of state" every single time. It mostly allows for setting initial state and resetting state.

This isn't the same as full control over setting the state value, but it offers some benefits as you'll see soon.

In our implementation of the custom hook, we've done the following:

```
export default function useExpanded () {
    // look here 👇
    const [expanded, setExpanded] = useState(false)
    ...
}
```

We've assumed that the initial state passed to the `useState` call will be "`false`" every single time.

That may not be the case.

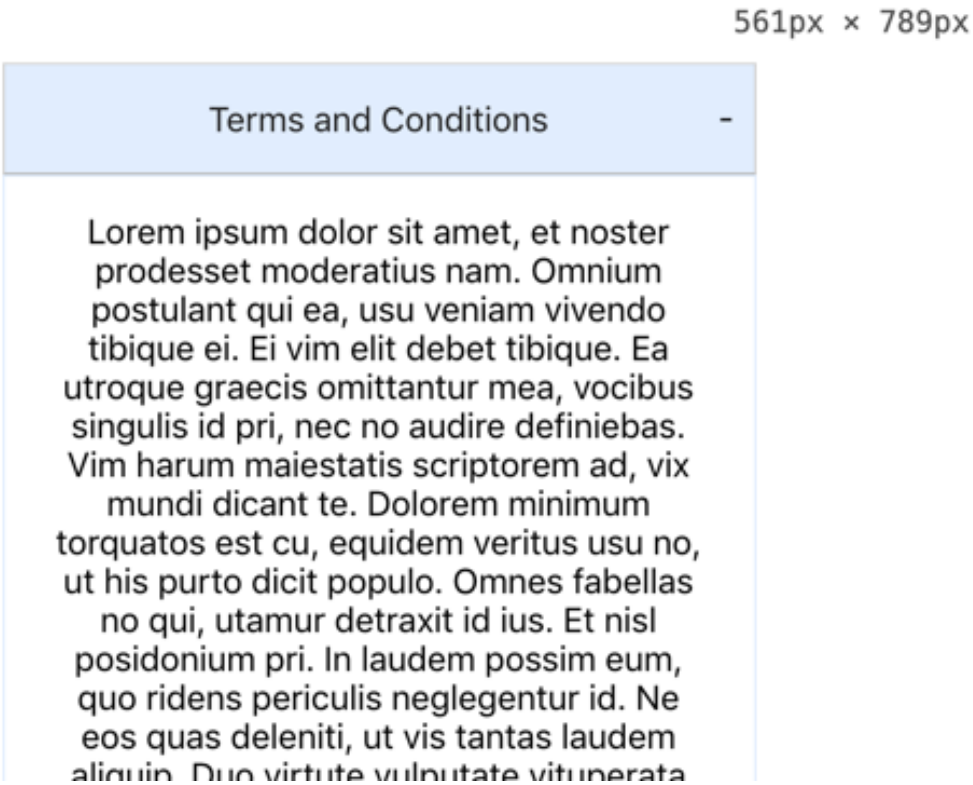
Let's have this value controllable from an argument the user can pass in. Let's call this parameter `initialExpanded` as seen below:

```
export default function useExpanded (initialExpanded = false) {
  const [expanded, setExpanded] = useState(initialExpanded)
  ...
  return value
}
```

Now the user can pass in `initialExpanded` into the `useExpanded` custom hook call to decide the initial state of the expanded component.

Here's an example where a user may want the terms and condition content expanded by default:

```
// user's app
function App () {
  // look here  - "true" passed into hook call 👇
  const { expanded, toggle } = useExpanded(true)
  return (
    ...
  )
}
```



With our previous setup this was not an easy feat for the user because we had internally hardcoded the initial state as `false` .

As you can see, this is a simple but helpful pattern.

While we've made it easier for the user to dictate the initial state within the custom hook, they should also be able to reset the state to the initial state at any point in

time i.e a `reset` callback they can invoke to reset the state to the initial default state they provided.

This is useful in many different use cases.

Let's consider a really trivial example.

Assume the terms and condition content in the user app was so long that they changed the default expanded state to `false` i.e the expandable content isn't open by default.

Since the content was long, they decided to provide a button towards the end of the write-up so a reader could click to revert the expandable content to the initial closed state (which may mean close the expandable content and perform some cleanups).

We could provide the user a `reset` callback for this, right?

Even though this particular example isn't the most realistic one for the `reset` functionality, in larger applications you can solve the problem using the same method discussed here.

So, here comes the solution.

All the `reset` function should do is set the "expanded" state back to the default provided i.e `initialExpanded`

Here's a simple implementation:

```
export default function useExpanded (initialExpanded = false) {
  const [expanded, setExpanded] = useState(initialExpanded)
  const reset = useCallback(
    () => {
      // look here 👇
      setExpanded(initialExpanded)
    },
    [initialExpanded]
  )
  ...
```

```
}
```

The code's simple enough. All the `reset` function does is call `setExpanded` with the `initialExpanded` value to reset the state back to the initial state supplied by the user.

Easy enough. However, there's still we need to do.

Remember, for this consumer of our custom hook, they want to close the terms and condition body and also perform some cleanup/side effect.

How will this user perform the cleanup after a reset? We need to cater for this use case as well.

Let's get some ideas from the implemented solution for the user to run custom code after every change to the internal `expanded` state.

```
// we made this possible 👇
useEffectAfterMount(
  () => {
    // user can perform any side effect here 👇
    console.log('Yay! button was clicked!!')
  },
  [expanded]
)
```

Well, now we need to make the same possible after a reset is made. Before I explain the solution to that, have a look at the usage of `useEffectAfterMount` in the code block above.

`useEffectAfterMount` accepts a callback to be invoked and an array dependency that determines when the effect function is called except when the component just mounts.

Now, for the regular state update, the user just had to pass in the array dependency `[expanded]` to get the effect function to run after every expanded state update.

For a reset what do we do?

Ultimately, here's what we want the user to do.

```
// user's app
const { resetDep } = useExpanded(false)
useEffectAfterMount(
  () => {
    console.log('reset cleanup in progress!!!!')
  },
  [resetDep]
 )
...
```

We need to provide a reset dependency the user can pass into the `useEffectAfterMount` array dependency.

That's the end goal.

Can you think of a solution to this? What do we expose as a reset dependency?

Well, the first thing that comes to mind is a state value to be set whenever the user invokes the `reset` callback. The state value will keep track of how many times a reset's been made.

If we increment the counter variable every time a reset is made, we can expose this as a reset dependency as it only changes when an actual reset is carried out.

Here's the implementation of that:

```
// useExpanded.js
...
const [resetDep, setResetDep] = useState(0)
const reset = useCallback(
  () => {
    // perform actual reset
    setExpanded(initialExpanded)
```

```
    // increase reset count - call this resetDep
    setResetDep(resetDep => resetDep + 1)
   },
   [initialExpanded]
 )
...
```

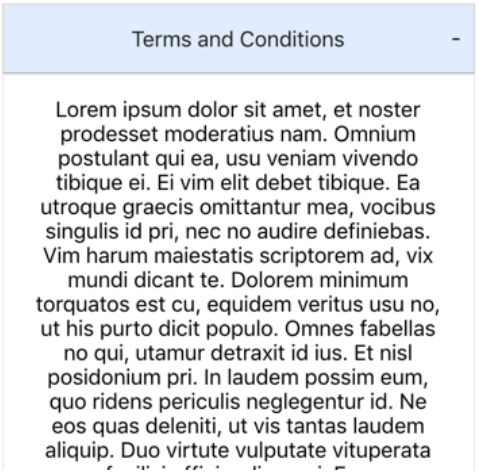
We can then expose `resetDep` alongside other values.

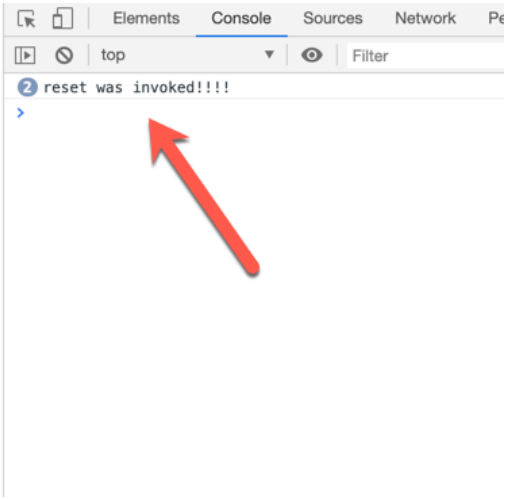
```
// useExpanded.js
...
const value = useMemo(
    () => ({
      expanded,
      toggle,
      getTogglerProps,
      reset,
      resetDep
    }),
    [expanded, toggle, getTogglerProps, reset, resetDep]
 )
 return value
...
```

This works just as expected!

```
// user's app
...
const { expanded, toggle, reset, resetDep } = useExpanded(false)

// useEffectAfterMount is now passed resetDep as an array dep
useEffectAfterMount(
   () => {
```

```
      // user can perform any reset cleanup/side effect
      console.log('reset was invoked!!!!')
    },
    [resetDep]
  )
...
```





This is a decent solution. It works fine, and is easy to reason about.

There's arguably one problem with the solution. Should we really be saving the reset count as a new state variable?

The `useExpanded` custom hook is mostly responsible for managing the `expanded` state. Introducing a new state variable feels like some inner conflict/pollution.

Before going on, this is just another case of personal preference. There's nothing technically wrong with the solution above.

I, on the other hand may prefer to provide the `resetDep` via a `ref` object. This way I'm sure to introduce only important variables as actual state variables.

The solution is similar, it's only an internal change within our custom hook, `useExpanded`. How the user consumes the `resetDep` remains the same.

So, here's the implementation with a `ref` object:

```
// useExpanded.js
```

```
 ...
const resetRef = useRef(0)
  const reset = useCallback(
    () => {
      // perform actual reset
      setExpanded(initialExpanded)
      // update reset count
      ++resetRef.current
    },
    [initialExpanded]
  )
...
// expose resetDep within "value"
...
resetDep: resetRef.current
```

And that's it! Same functionality, different approaches. Feel free to use whichever feels right to you. I pick the second though :)

As a summary, remember that with the state initialiser pattern you offer the user to ability to decide initial state within your custom hook, you allow for resetting and invoking a custom function after a reset is made as well.

Even If you had a more complex custom hook with multiple state values, you could still use this technique. Perhaps, receive the initial state passed in by the user as an object. Create a memoized `initialState` object from the user's and use this within your custom hook.

If you also choose to manage your internal state via `useReducer`, you can still apply this pattern.

Point is, regardless of your implementation, the goal remains the same. Provide the user with the ability to decide initial state, allow them to reset and invoke custom callbacks after a reset is made.

In the next section we'll take things a notch higher as we give even more control to the users of our components. Remember, that's what building highly reusable components is mostly about.

State Reducer

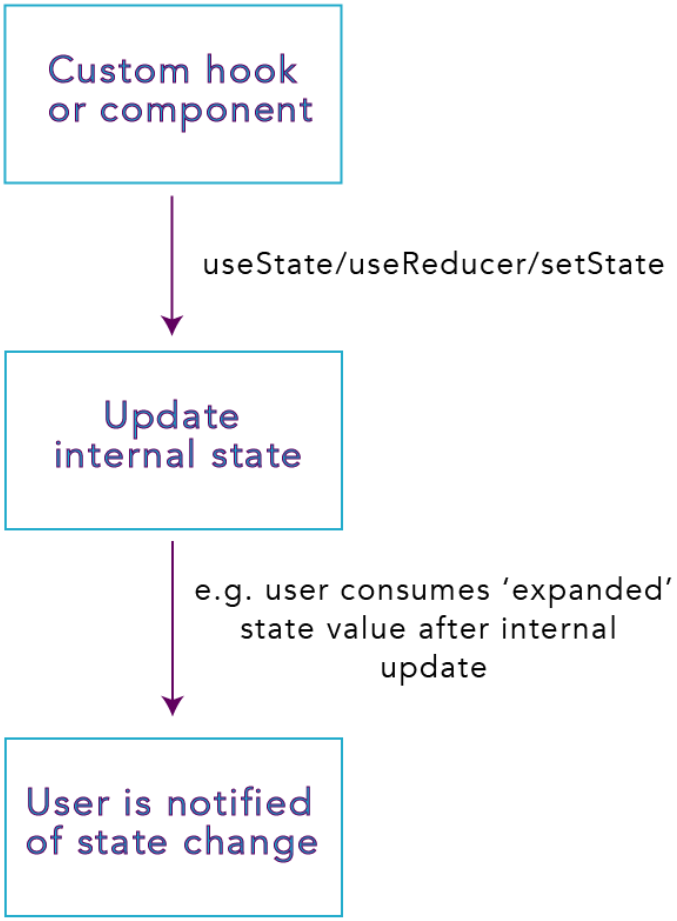
The state reducer pattern is the final and perhaps most advanced pattern we'll be discussing. Don't let that scare you. I'll take my time to explain how it really works without leaving behind why it matters as well.

When we write custom hooks or components, they most likely have some way to keep track of internal state. With the state reducer pattern, we give control to the user of our custom hook/component to control how state is updated internally.

The technical term for this is called "inversion of control". In lay man terms it means a system that allows the user to control how things work internally within your API.

Let's look at how this may work conceptually.

The typical flow for communicating updates to the user of your custom hook or component looks like this:

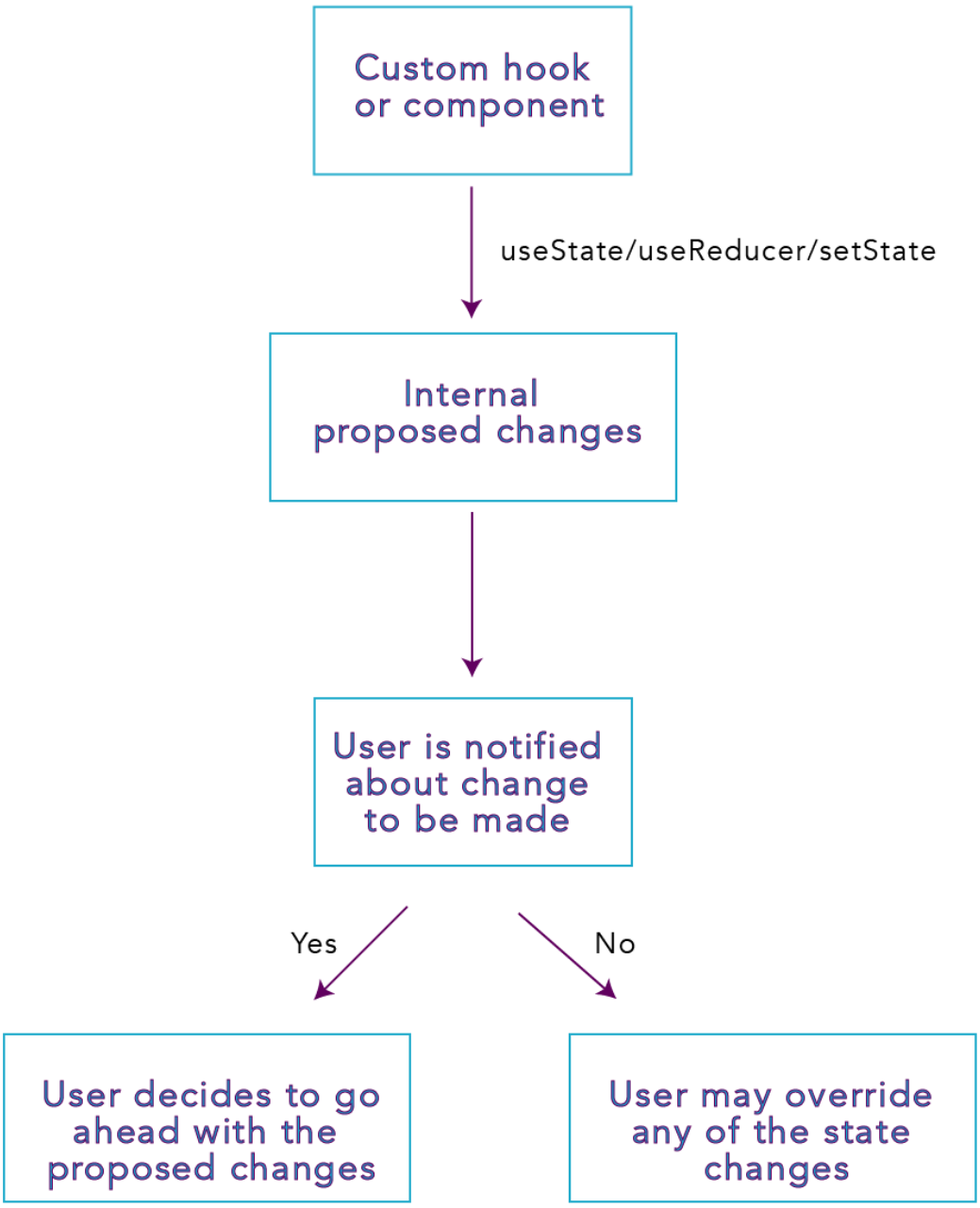


Within your custom hook you call `useState` or `useReducer` or `setState` to update state internally. Once the update **is already made**, the new state value is communicated to the user.

This is how all the patterns we've looked at so far has worked.

With the state reducer pattern, there's a significant change in how internal state updates are made.

Here's a graphical illustration:



With state reducers, what's important to note is that *before* we make any internal state update, we send our proposed changes over to the user. If the user is okay with the changes, we go ahead and update state with our proposed changes. If they need to override a state change, they can do that as well.

The internal update is controlled by the user. They have a say!

Refactoring to useReducer

The most intuitive way to make the state reducer pattern work is using the `useReducer` hook to manage internal state.

First, let's refactor our custom hook to use `useReducer` instead of `useState`.

To refactor our custom hook to use the `useReducer` hook for state updates, everything within the custom hook stays the same except how the `expanded` state is managed.

Here's what's to be changed.

First, we'll change the state from just a boolean value, `true` or `false` to an object like this: `{expanded: true || false}`.

```
// useExpanded.js
export default function useExpanded (initialExpanded = false) {
  const initialState = { expanded: initialExpanded }
  ...
}
```

Once we do that, we'll invoke `useReducer` to manage the state.

```
// useExpanded.js
// before
  ...
 const [expanded, setExpanded] = useState(initialExpanded)
```

```
// now

const [{ expanded }, setExpanded] = useReducer(internalReducer,
initialState)

...
```

Note how we need to destructure `expanded` from the state object. It's worth mentioning that `setExpanded` returned from the `useReducer` call now works as a dispatcher. I'll show the implication of this soon.

If you're new to `useReducer`, it is typically called with a reducer and an initial state value.

```
useReducer(internalReducer, initialState)
```

If you've Redux in the past then you're most likely familiar with the concept of reducers. If not, a reducer is just a function that receives `state` and `action` to return a *new state*.

`action` usually refers to an object, but `useReducer` isn't strict about this. If we stick to the Redux convention, a state update is always made by "dispatching" an action.

The "dispatcher" is the second value returned by the `useReducer` call. Also, In order for the reducer to identify each action being dispatched, a `type` property always exist on the action.

```
// for example

action = {
   type: "AN_ACTION_TYPE"
}
```

I don't want to turn this into a `redux` guide, but if you need a refresher, I wrote an illustrated guide that'll make the concept of reducers and actions all sink in.

With the `useReducer` call in place, we need to actually write the reducer used within the call i.e `internalReducer`

Here's the implementation:

```
// useExpanded.js
```

```
const internalReducer = (state, action) => {
  switch (action.type) {
    case useExpanded.types.toggleExpand:
      return {
        ...state,
        expanded: !state.expanded //toggle expand state property
      }
    case useExpanded.types.reset:
      return {
        ...state,
        expanded: action.payload // reset expanded with a payload
      }
    default:
      throw new Error(`Action type ${action.type} not handled`)
  }
}


export default function useExpanded (initialExpanded = false) {
const [{ expanded }, setExpanded] = useReducer(internalReducer,
initialState)
...
}
```

Remember I said actions typically have a `type` property. The reducer checks this type property and returns new state based on the type.

To stay consistent and prevent unwanted typos, the available types for the `useExpanded` custom hook are centralised within the same file.

```
// useExpanded.js
...
useExpanded.types = {
  toggleExpand: 'EXPAND',
```

```
  reset: 'RESET'
}
```

Which explains why they're used in the user as shown below:

```
const internalReducer = (state, action) => {
  switch (action.type) {
    case useExpanded.types.toggleExpand: //👈 look here
      return {
        ...state,
        expanded: !state.expanded
      }
 ...
}
```

Note that the returned value from the reducer represents the new state of the `expanded` state. For example, here's the returned value for type, `useExpanded.types.toggleExpand`.

```
...
return {
        ...state,
        expanded: !state.expanded
      }
...
```

For this to come full circle, consider how the `reset` function now works.

```
// before
setExpanded(initialExpanded)
// now
...
setExpanded({
  type: useExpanded.types.reset,
```

```
  payload: initialExpanded // pass a payload "initialExpanded"
})
```

`setExpanded` is now a "dispatcher" so it's called with an action object. An action object has a type and anything else. For redux users, you're familiar with the `payload` property which just keeps a reference to a value required by this action to make a state update.

Note how the reducer returns new state based on this dispatched reset action:

```
...
case useExpanded.types.reset:
      return {
        ...state,
        expanded: action.payload // reset expanded with the payload
      }
...
```

Where the payload passed in is the `initialExpanded` variable.

We've made decent progress!

Below's a quick summary of the changes made:

```
// useExpanded.js
export default function useExpanded (initialExpanded = false) {
   // keep initial state in a const variable
   // NB: state is now an object e.g {expanded: false}
  const initialState = { expanded: initialExpanded }


  // the useReducer call for state updates 👇
  const [{ expanded }, setExpanded] = useReducer(internalReducer,
initialState)
  // 👆 Note how we destructured state variable, expanded


  // setExpanded is now a dispatcher. Toggle fn refactored 👇
```

```
  const toggle = useCallback(
// Every setExpanded call should be passed an object with a type
    () => setExpanded({ type: useExpanded.types.toggleExpand }),
    []
  )
...
}
```

The available types for state updates are then centralised to prevent string typos.

```
// useExpanded.js
...
useExpanded.types = {
  toggleExpand: 'EXPAND',
  reset: 'RESET'
}
```

Here's how the reset is performed internally:

```
// before
setExpanded(initialExpanded)
// now
...
setExpanded({
  type: useExpanded.types.reset,
  payload: initialExpanded // pass a payload "initialExpanded"
})
...
```

The result of this is a complete refactor to `useReducer` – no additional functionality has been added.

Does anything confuse you?

Please have a look one more time. This will only prove difficult if you don't know how `useReducer` works. If that's the case, I suggest you quickly check that out.

Having completed this refactor, we need to implement the actual state reducer pattern.

Right now, the custom hook is invoked like this:

```
// user's app
...
const { expanded, toggle, reset, resetDep} = useExpanded(false)
```

Where the value passed to `useExpanded` is the initial expanded state.

Remember from the illustration above that we need to communicate our internal state changes to the user. The way we'll do this is by accepting a second parameter, the user's own reducer.

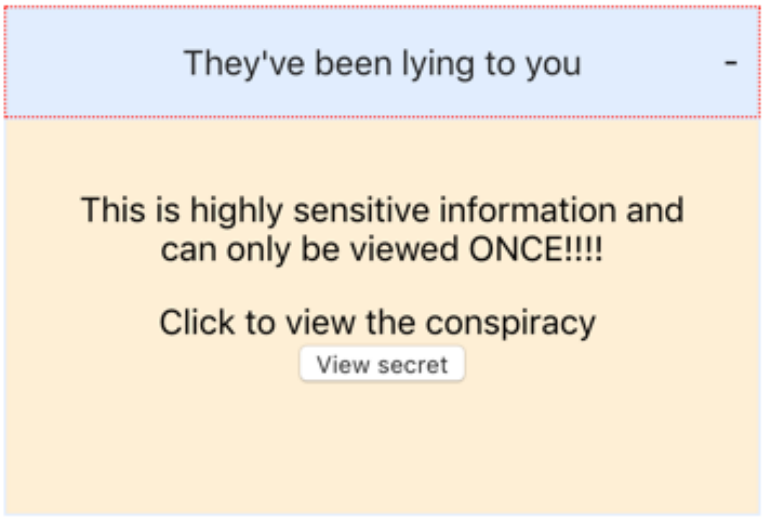
For example:

```
// user's app
function App () {
  const { expanded, toggle, reset, resetDep } = useExpanded(
    false,
    appReducer // 👈 the user's reducer
  )
...
}
```

Before we go ahead with the implementation, let's see how this particular user intends to use our custom hook.

The User's Application

Our user is an hacker who has found proof of a big conspiracy going on in Mars. To share this with the world, they want to use our `useExpanded` hook and default UI elements we provide to build the following expandable view:



That's the easy part.

What's more interesting is that the hacker only wants a reader to view this secret once. It's such a big conspiracy.

The hacker's goal is that whenever the user clicks the button to view the secret, the expandable content is `reset` to the default unexpanded state, and the reader can click all they want on the header but the content won't be expanded.

You may check the final working version of this user app by running the app in the `state-reducer` directory.

Do you see why the user needs the state reducer pattern?

By default, whenever the `Header` element is clicked, the hacker calls the `toggle` function.

...

```
// look here 👇
<Header toggle={toggle} style={{ border: '1px dotted red' }}>
   They've been lying to you
</Header>
...
```

Internally, the `toggle` function always toggles the `expanded` state property. The problem is, when the reader has viewed the secret, the hacker wants further clicks on the `Header` to **not** trigger a state update.

We need to cater for this use case.

In the mean time, here's how the hacker has handled viewing the secret document.

They attach a click handler to the `button` which points to the `reset` callback we provide.

```
...
 <section className='App'>
      ...
      <Body>
          <p>
            Click to view the conspiracy <br />
            <button onClick={reset}> View secret </button>
          </p>
        </Body>
      </div>
</section>
```

The `reset` callback resets the expanded state to the initial expanded state of `false` provided by the hacker.

```
...
const { expanded, toggle, reset, resetDep} = useExpanded(
 false)
```

The initial state provided by the hacker is `false` – this means the expandable content is closed after the click.

Good.

Within the `useEffectAfterMount` hook, they then perform a side effect, opening the secret in a new window tab based on the `resetDep` which changes when the user clicks the button.

```
...
useEffectAfterMount(
    () => {
      // open secret in new tab 👇
      window.open('https://leanpub.com/reintroducing-react', '_blank')
     // can do more e.g 👉 persist user details to database
    },
    [resetDep]
  )
...
```

So far the hacker is happy with the API we've provided. We just need to handle his last use case for preventing further state updates on clicking the Header element after a reader has viewed the secret once.

Implementing the State Reducer Pattern

Our solution to the hacker's feature request is to implement the state reducer pattern.

When the reader clicks the `Header` element, we'll communicate the changes we propose to be made internally to the hacker i.e we'll let the hacker know we want to update the expanded state internally.

The hacker will then check their own application state. If the reader has already viewed the secret resource, they'll communicate to our custom hook NOT to allow a state update.

There it is. The power of the state reducer pattern, giving control of internal state updates to the user.

Now, let's write the technical implementation.

First, we'll expect a second argument from the hacker, their own reducer.

```
// Note the second parameter to our hook, useExpanded 👇
function useExpanded (initialExpanded = false, userReducer) {
 ...
}
```

The second parameter to our `useExpanded` hook represents the user's reducer.

After the refactor to `useReducer`, we got the value of the `expanded` state by calling our internal reducer.

```
// before. Note the use of our "internalReducer"
 const [{ expanded }, setExpanded] = useReducer(internalReducer,
initialState)
...
```

The problem with this is that our internal reducer always returns our internal proposed new state. This isn't the behaviour we want. Before we decide what the `expanded` state will be, we need to communicate our proposed state change to the user i.e the hacker.

So, what do we do?

Instead of passing our `internalReducer` to the `useReducer` call, let's pass in another reducer we'll call `resolveChangesReducer`.

The sole purpose of `resolveChangesReducer` is to resolve changes between our internal implementation and what the user suggests.

Every reducer takes in state and action, right?

The implementation of `resolveChangesReducer` begins by receiving the state and action, and holding a reference to our internal change.

```
const resolveChangesReducer = (currentInternalState, action) => {
    // look here 👇
    const internalChanges = internalReducer(currentInternalState,
action)
    ...
  }
```

A reducer always returns new state. The value `internalChanges` holds the new state we propose internally by invoking our `internalReducer` with the current state and action.

We need to communicate our proposed changes to the user. The way we do this is by passing this `internalChanges` to the user's reducer i.e the second argument to our custom hook.

```
const resolveChangesReducer = (currentInternalState, action) => {
    const internalChanges = internalReducer(currentInternalState,
action)
    // look here 👇
    const userChanges = userReducer(currentInternalState, {
      ...action,
      internalChanges
    })
    ...
  }
```

Right now, we invoke the user's reducer with the internal state and an action object.

Note that the action object we send to the user is slightly modified. It contains the action being dispatched and our internal changes!

```
{
 ... action,
 internalChanges: internalChanges
```

```
}
```

Every reducer takes in state and action to return a new state. It is the responsibility of the user's reducer to return whatever they deem fit as the new state. They can either return our changes if they're happy with it OR override whatever changes they need to.

With a reference to the user changes held, the `resolveChangesReducer` now returns the proposed changes from the user NOT ours.

```
// useExpanded.js

...
const resolveChangesReducer = (currentInternalState, action) => {
    const internalChanges = internalReducer(currentInternalState,
action)
    const userChanges = userReducer(currentInternalState, {
      ...action,
      internalChanges
    })

   // look here 👇
    return userChanges
  }
...
```

The return value of a reducer depicts new state. The `resolveChangesReducer` reducer returns the user's changes as the new state!

That's all there is to do!

Have a look at the complete implementation of the state reducer. I've omitted whatever hasn't changed. You can look in the source yourself if you care about the entire source code (and you should!)

```
// necessary imports 👇
import { useCallback, useMemo, useRef, useReducer } from 'react'
```

```
...
// our internal reducer 👇
const internalReducer = (state, action) => {
  switch (action.type) {
    case useExpanded.types.toggleExpand:
      return {
        ...state,
        expanded: !state.expanded
      }
    case useExpanded.types.reset:
      return {
        ...state,
        expanded: action.payload
      }
    default:
      throw new Error(`Action type ${action.type} not handled`)
  }
}
// the custom hook 👇
export default function useExpanded (initialExpanded = false,
userReducer) {
  const initialState = { expanded: initialExpanded }
  const resolveChangesReducer = (currentInternalState, action) => {
    const internalChanges = internalReducer(
        currentInternalState,
        action
    )
    const userChanges = userReducer(currentInternalState, {
      ...action,
      internalChanges
    })
```

```
    return userChanges
  }
  // the useReducer call 👇
  const [{ expanded }, setExpanded] = useReducer(
    resolveChangesReducer,
    initialState
  )
  ...
  // 👇 value returned by the custom hook
  const value = useMemo(
    () => ({
      expanded,
      toggle,
      getTogglerProps,
      reset,
      resetDep: resetRef.current
    }),
    [expanded, toggle, getTogglerProps, reset]
  )

  return value
}
// 👇 the available action types
useExpanded.types = {
  toggleExpand: 'EXPAND',
  reset: 'RESET'
}
```

Not so hard to understand, huh?

With the pattern now implemented internally, here's how the hacker took advantage of that to implement their feature.

```
// the user's app 👇
function App () {
  // ref holds boolean value to decide if user has viewed secret or not
  const hasViewedSecret = useRef(false) // 👈 initial value is false
  // hacker calls our custom hook 👇
  const { expanded, toggle, reset, resetDep = 0 } = useExpanded(
    false,
    appReducer // 👈 hacker passes in their reducer
  )

  // The user's reducer
  // Remember that action is populated with our internalChanges
  function appReducer (currentInternalState, action) {
    // dont updated "expanded" if reader has viewed secret
    // i.e hasViewedSecret.current === true
    if (hasViewedSecret.current) {
     // object returned represents new state proposed by hacker
      return {
        ...action.internalChanges,
        // override internal update
        expanded: false
      }
    }

    // else, hacker is okay with our internal changes
    return action.internalChanges
  }

  useEffectAfterMount(
```

```
    () => {
      // open secret in new tab 👇
      window.open('https://leanpub.com/reintroducing-react', '_blank')
      //after viewing secret, hacker sets the ref value to true.
      hasViewedSecret.current = true
    },
    [resetDep]
  )
 ...
}
```

That's a well commented code snippet, you agree?

I hope you find it good enough to understand how a user may take advantage of the state reducer pattern you implement in your custom hooks / component.

With this implementation, the hacker's request is fulfilled! Here's a GIF that shows the app in action.

There are many different ways the user could choose to dictate how internal state is updated.

For example the hacker could be more explicit about interfering state changes only when the action type refers to a toggle expand.

Here's what I mean:

```
// user's app
...
function appReducer (currentInternalState, action) {
    if (
      hasViewedSecret.current &&
      // look here 👇
      action.type === useExpanded.types.toggleExpand
    ) {
      return {
```

```
        ...action.internalChanges,
        // 👇 override internal update
        expanded: false
      }
    }
    return action.internalChanges
  }
...
```

By doing this we may extend our custom hook's functionality to expose a `override` callback like this:

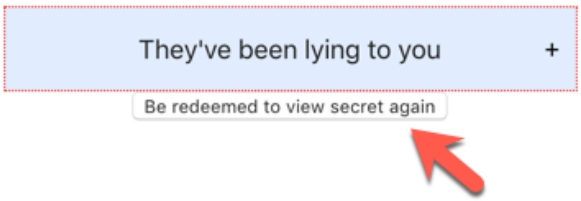
```
// useExpanded.js
...
  // exposed callback 👇
  const override = useCallback(
    () => setExpanded({ type: useExpanded.types.override }),
    []
  )

...
useExpanded.types = {
  toggleExpand: 'EXPAND',
  reset: 'RESET',
  override: 'OVERRIDE' // 👈 new "OVERRIDE" type
}
```

The `override` type is then handled via our internal reducer as follows:

```
...
case useExpanded.types.override:
      return {
        ...state,
```

```
        expanded: !state.expanded // update state
      }
...
```

What this allows for is a way to override changes. For example, the hacker may decide to show a button a user may click in order to view the secret again.




```
// note override being retrieved for our hooks call 👇
const { expanded, toggle, override, reset, resetDep = 0 } = useExpanded(
    false,
    appReducer
  )

...
// button is only shown if reader has viewed secret
{ hasViewedSecret.current && (
        <button onClick={override}>
            Be redeemed to view secret again
       </button>
)}
```

Note how the button's call back function is `override`.

This allows for the expanded container to be open – and letting the use view the secret one more time. Here's a GIF showing how that works.

Cleanups

Right now, if some user other than this crazy hacker uses our custom function without passing a reducer, their app breaks.

```
useExpanded(false) // 👈 user doesn't need reducer feature
```

Let's provide a default reducer within our custom hook i.e for users who don't need this feature.

```
// userExpanded.js
function useExpanded (
  initialExpanded = false,
  userReducer = (s, a) => a.internalChanges // 👈 default
) {
  ...
}
```

Using the ES6 default parameters a default reducer has been provided. The default user reducer takes in state and action, then returns our `internalChanges`.

With this fix, users who don't need the reducer feature don't get an error.

This has been such an interesting discourse!

Remember, the benefit of the state reducer pattern is in the fact that it allows the user "control" over how internal state updates are made.

Our implementation has made our custom hook so much more reusable. However, how we update state is now part of the exposed API and if you make changes to that if may be a breaking change for the user.

Regardless, this is still such a good pattern for complex hooks and components.

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, and 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.

Last Words

Thank you for joining me on this journey to Reintroducing React. As I seat in my room typing this last words with eyes pale from working since midnight, I do hope that this book has helped you learn one or two things. *Heck! more than two things, I'd say.*

If that's the case, I couldn't be any happier.

Do have a successful career, and make to practice what you've learnt here.

Cheers,

- Ohans Emmanuel