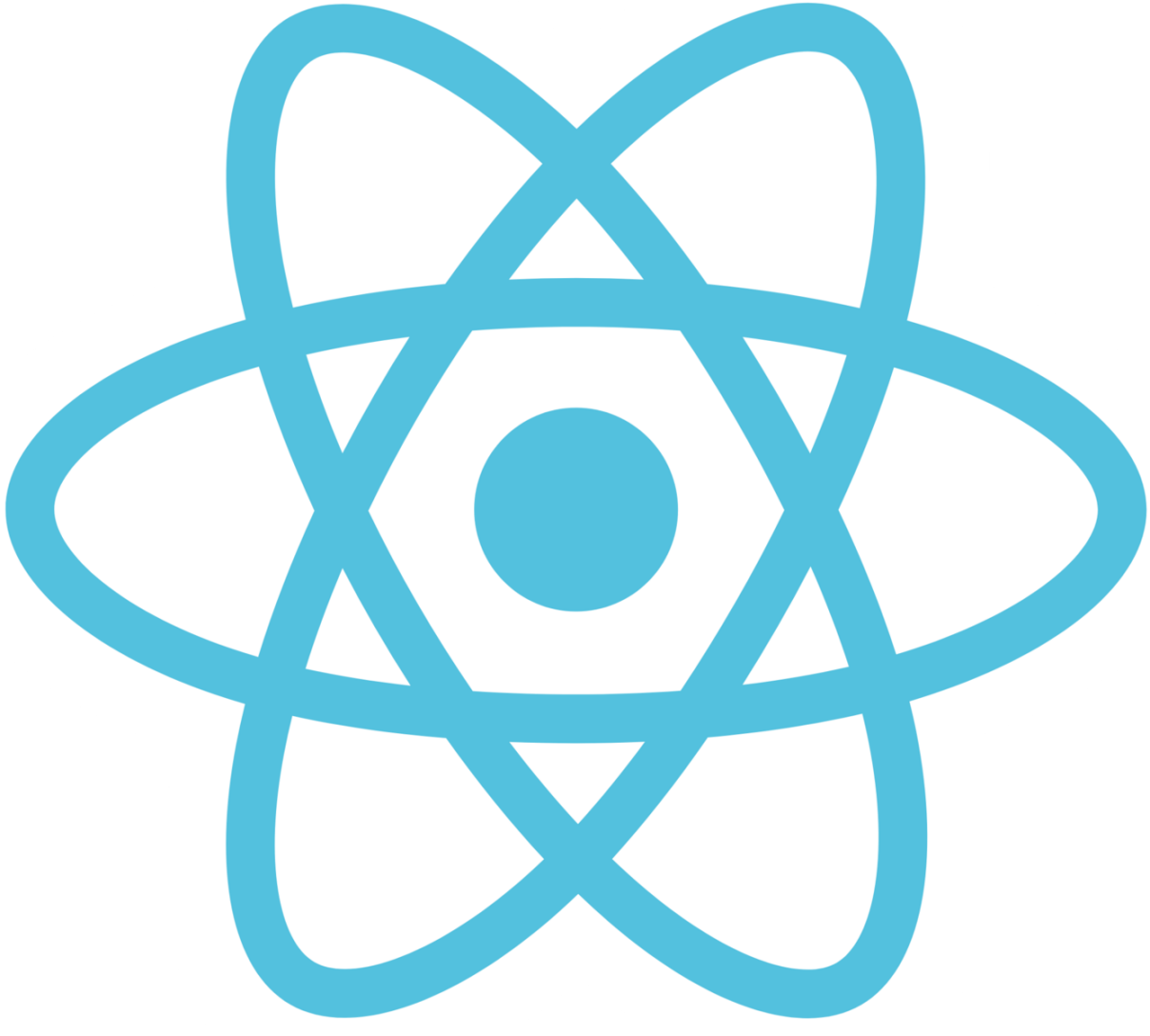


The React fundamentals.

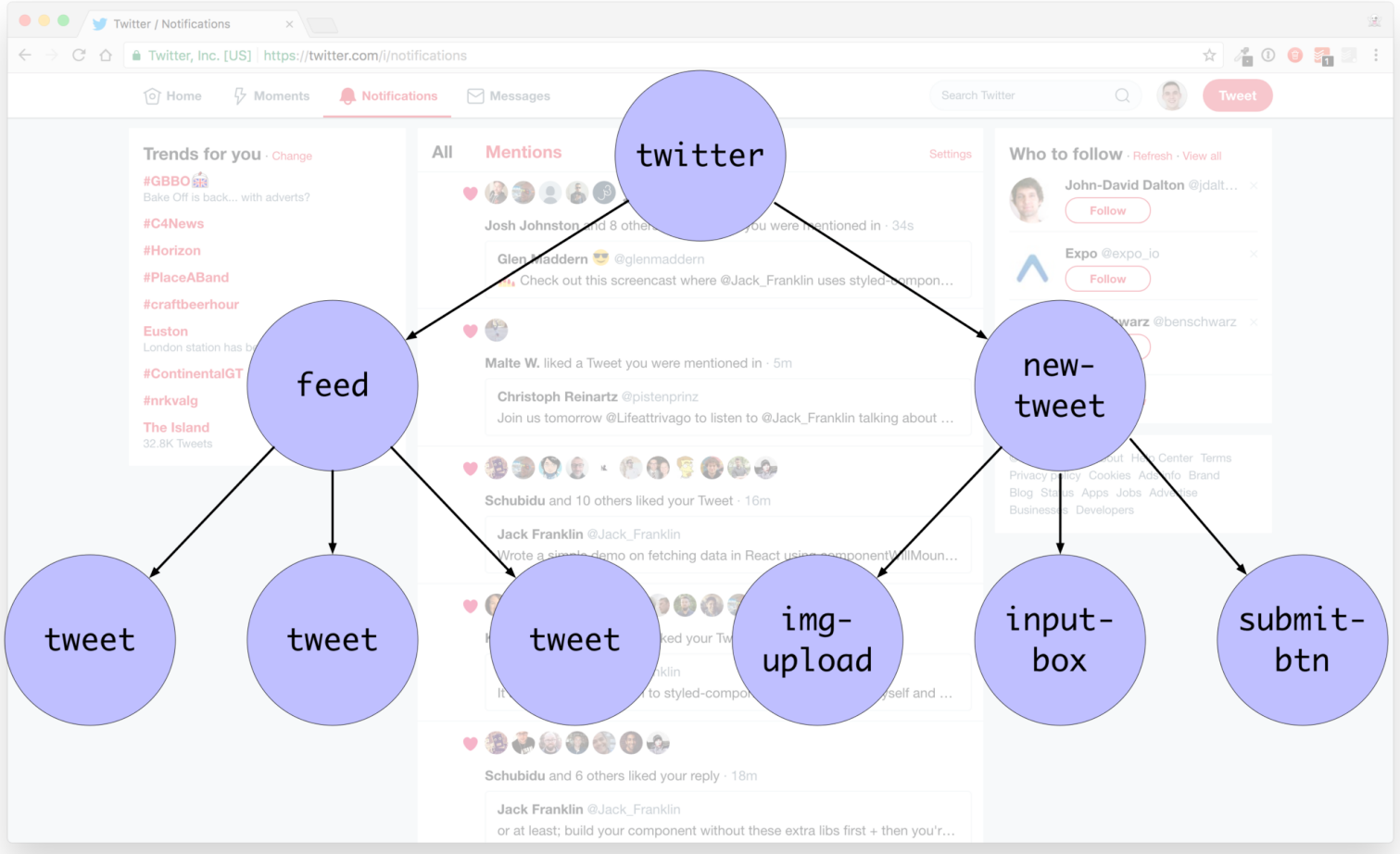
We're going to talk about *only* React for now.

Not a single third party library*

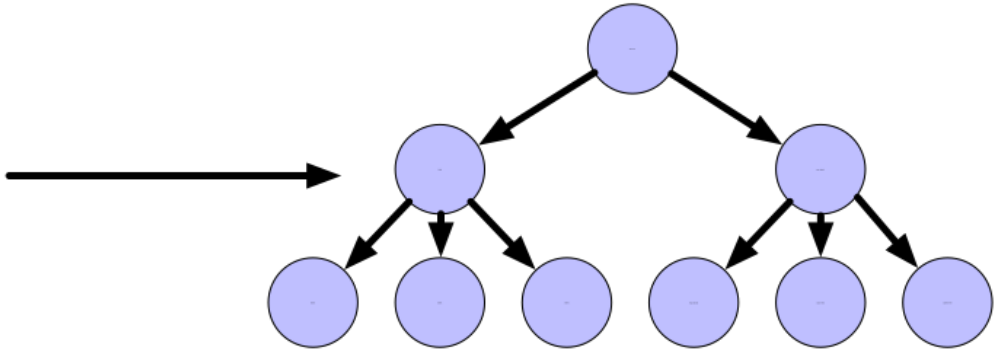
*ok there's one but I wrote it so that's allowed

By learning the core React framework and ideas, you'll be equipped to pick up, learn and work with any additional React libraries.

Components


twitter
feed
new-tweet
tweet
tweet
tweet
img-upload
input-box
submit-btn

```
{
  url: '/notifications',
  userId: 12345,
  avatar: 'foo.jpg',
  latestTweets: [
    {
      id: 456,
      text: 'Hello world',
    },
    …
  ],
}
```

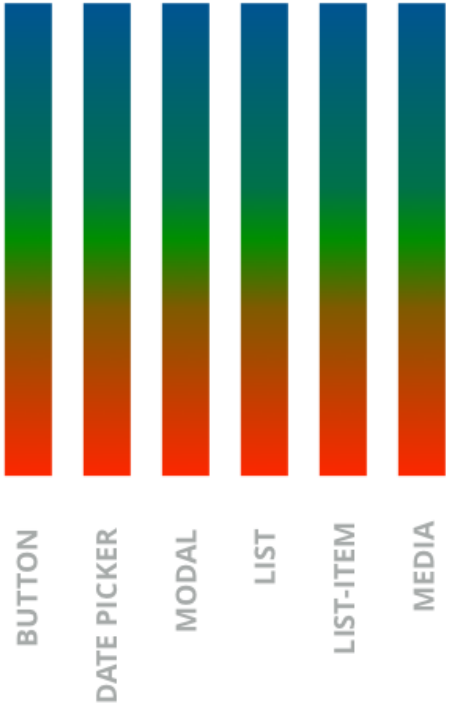


State → UI

Separation of Concerns

Separation of Concerns

(only, from a different point of view)



by Cristiano Rastelli, @areaweb

React + the DOM

- Tell React what each component should render.
- React takes care of the DOM for you.

Let's get started!

Today we're going to be building a journal app that will let you keep a daily diary.

We'll start small and build it out as we learn more about React and what it can do for us.

Here's the code for exercise 1. Let's walk through it together first.

```
import ReactDOM from 'react-dom'
import React from 'react'

const JournalApp = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(JournalApp),
  document.getElementById('react-root')
)
```

```
import ReactDOM from 'react-dom'
import React from 'react'

const JournalApp = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(JournalApp),
  document.getElementById('react-root')
)
```

```
import ReactDOM from 'react-dom'
import React from 'react'

const JournalApp = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(JournalApp),
  document.getElementById('react-root')
)
```

```
import ReactDOM from 'react-dom'
import React from 'react'

const JournalApp = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(JournalApp),
  document.getElementById('react-root')
)
```

Let's render our first component and style the header correctly.

```
npm run exercise react 1
```

You'll find the exercise running on localhost:1234. If anything isn't working, please shout!

(Don't worry if it isn't working, it's normal for every workshop to have a few glitches at the start)

React.createElement is quite verbose...

Exercise 2

```
const HelloWorld = ()
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return <h1>Hello World</h1>
}

ReactDOM.render(<HelloWorld />,
 document.getElementById('react-root')
)
```

JSX!

<HelloWorld /> is equivalent to:
React.createElement(HelloWorld)

<h1 className="foo">Hello</h1> is equivalent to
React.createElement('h1', { className: 'foo' }, 'Hello')

remember, JSX gets compiled to React.createElement.

It's just JavaScript!

```
const HelloWorld = props => {
  // TODO: pass through another prop to customise the greet
  // rather than it always be hardcoded as Hello
  return <h1>Hello, {props.name}</h1>
}

ReactDOM.render(
  <HelloWorld name="Jack" />,
  document.getElementById('react-root')
)
```

we can pass properties (or, props) through to components to configure them or change their behaviour

remember, props are all just JavaScript, so you can pass through any data type - these aren't HTML attributes

Exercise 3

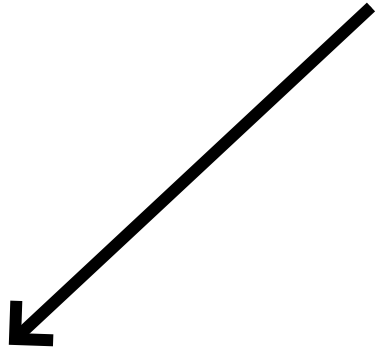
```
const bunchOfProps = {
  name: 'Jack',
  age: 25,
  colour: 'blue',
}

ReactDOM.render(
  <HelloWorld
    name={bunchOfProps.name}
    age={bunchOfProps.age}
    colour={bunchOfProps.colour}
  />,
  document.getElementById('react-root')
)
```

this is quite a lot to type and means manual work if the object gets more properties that you want to be props

Exercise 4

the JSX spread syntax saves us manually listing all props and lets us apply an object as props to a component

```
const bunchOfProps = {
  name: 'Jack',
  age: 26,
  colour: 'blue',
}

ReactDOM.render(
  <HelloWorld {...bunchOfProps} />,
  document.getElementById('react-root')
)
```



Exercise 4

it's easy to forget to pass a prop that a component needs, or make a typo

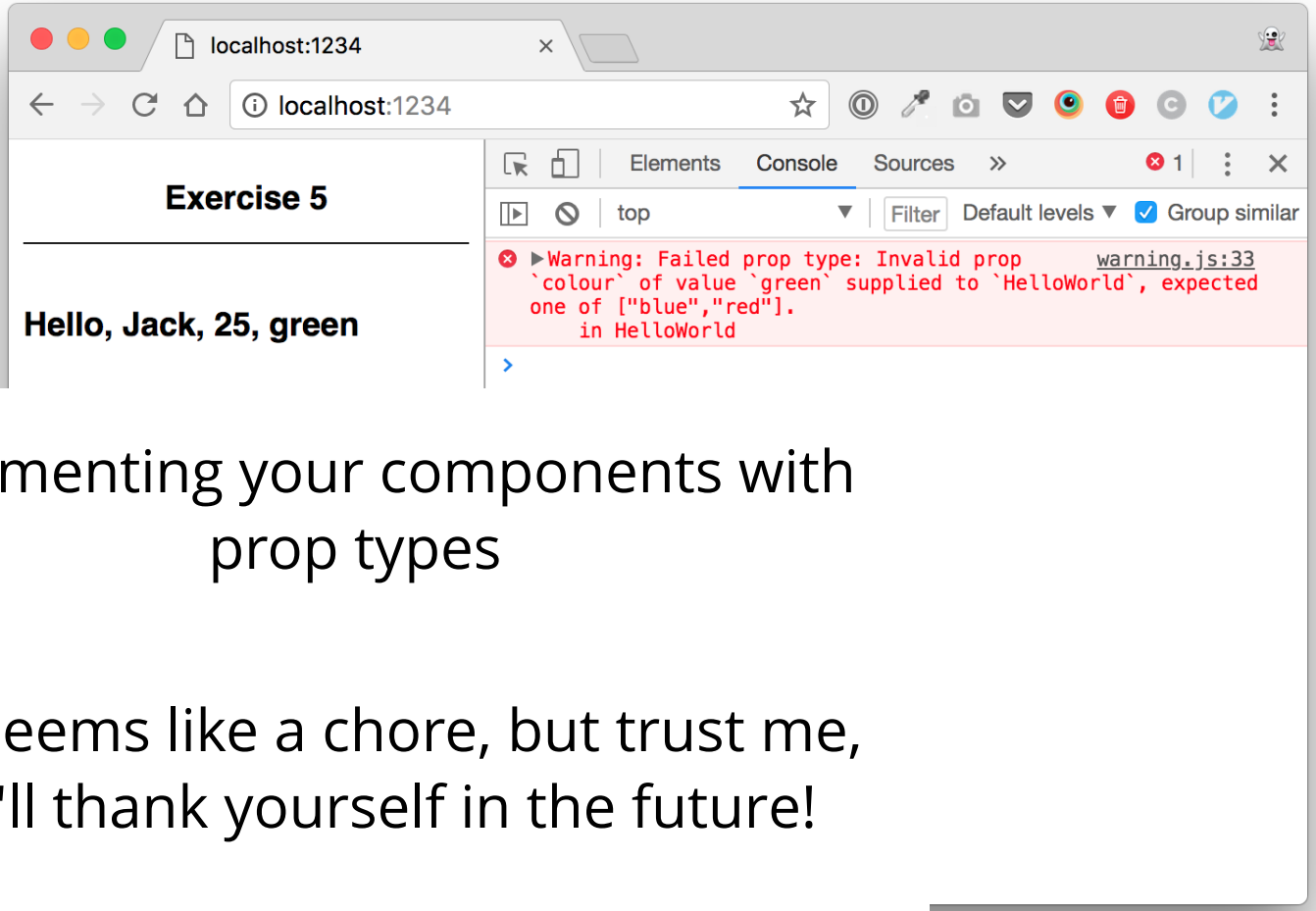
```
1  import ReactDOM from 'react-dom'
2  import React from 'react'
3  import PropTypes from 'prop-types'
4
5  const JournalApp = props => {
6     ...
7  }
8
9  JournalApp.propTypes = {
10   // TODO: define the prop type for the name, age and locatic
11 }
```

```
1  import ReactDOM from 'react-dom'
2  import React from 'react'
3  import PropTypes from 'prop-types'
4
5  const JournalApp = props => {
6      ...
7  }
8
9  JournalApp.propTypes = {
10   // TODO: define the prop type for the name, age and locatic
11 }
```

```
1  import ReactDOM from 'react-dom'
2  import React from 'react'
3  import PropTypes from 'prop-types'
4
5  const JournalApp = props => {
6     ...
7  }
8
9  JournalApp.propTypes = {
10   // TODO: define the prop type for the name, age and locatio
11 }
```

```
1  import ReactDOM from 'react-dom'
2  import React from 'react'
3  import PropTypes from 'prop-types'
4
5  const JournalApp = props => {
6     ...
7  }
8
9  JournalApp.propTypes = {
10   // TODO: define the prop type for the name, age and locatic
11 }
```

```
HelloWorld.propTypes = {
  name: PropTypes.string.isRequired,
  colour: PropTypes.oneOf(['blue', 'red']).isRequired,
}
```

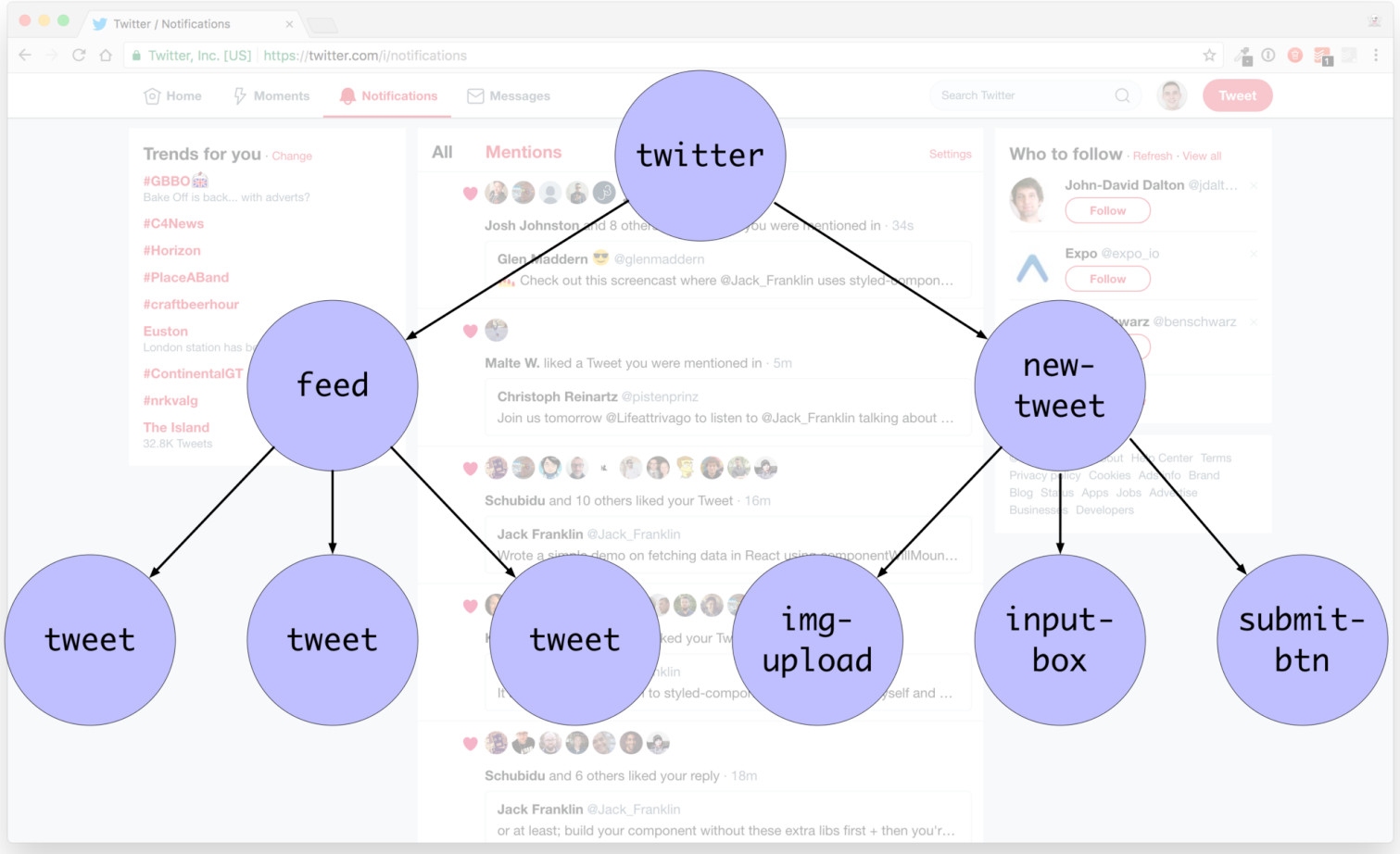


Documenting your components with prop types

This seems like a chore, but trust me, you'll thank yourself in the future!

https://reactjs.org/docs/typechecking-with-proptypes.html

note that not all my examples use prop-types today, but that's just to keep them focused. My real code always uses them!


twitter
feed
new-tweet
tweet
tweet
tweet
img-upload
input-box
submit-btn

```
const AskQuestion = () => {
  return <p>How is your day going today?</p>
}

const HelloWorld = props => {
  return (
    <div>
      <AskQuestion />
      <h1>
        {props.greeting}, {props.name}
      </h1>
    </div>
  )
}
```

defining and then using a component

Components can render other components.

React components must start with a capital letter.

Exercise 6

Managing state

Props

Data a component is given and uses but **cannot change.**

State

Data a component **owns** and can **change.**

Who has heard of React hooks?

In React 16.7 and before

Functional components

What we've used so far. These components **cannot have state.**

Class components

We define our components as classes. These components **can have state.**

In React 16.7 and before

```
import React, { Component } from 'react'

class MyComponent extends Component {
  constructor(props) {
    super(props)

    this.state = {...}
  }

  render() {
    return <p>Hello world</p>
  }
}
```

this is boilerplate you don't need to worry about

this is like the body of the functional components we have been using so far!

Listening to user events

In React 16.7 and before

```
onButtonClickIncrement() {
}

render() {
  return (
    <div>
      <p>current count: {this.state.count}</p>
      <button onClick={this.onButtonClickIncrement.bind(this)}>
        Click to increment
      </button>
    </div>
  )
}
```

we have to bind to ensure the right scope within the event handler

(we'll see a nicer way to do this later on)

Updating state

In React 16.7 and before

To update the state, we have to use React's method

This ensures React knows about our state change, and can act accordingly.

In React 16.7 and before

Updating state

```
this.setState({
  newValue: 3,
})
```

when the new state doesn't depend on the old state

React 16.8

Hooks!

Hooks solve a wide variety of seemingly unconnected problems in React that we've encountered over five years of writing and maintaining tens of thousands of components.

We are going to exclusively use hooks for this workshop

Because they have been widely adopted and are considered superior to the class based approach.

And conceptually they are easier to learn and understand :)

Hook one: useState

Counter component

```
1  import ReactDOM from 'react-dom'
2  import React, { Component, useState } from 'react'
3  import PropTypes from 'prop-types'
4
5  const Counter = props => {
6    const [count, setCount] = useState(props.start)
7
8    const onIncrementClick = () => {
9      setCount(oldCount => oldCount + 1)
10   }
11   // TODO: add another button that decrements the count
12
13   return (
14     <div>
15       <p>current count: {count}</p>
16       <button onClick={onIncrementClick}>Click to increment</button>
17     </div>
18   )
19 }
```

Counter component

```
1  import ReactDOM from 'react-dom'
2  import React, { Component, useState } from 'react'
3  import PropTypes from 'prop-types'
4
5  const Counter = props => {
6    const [count, setCount] = useState(props.start)
7
8    const onIncrementClick = () => {
9      setCount(oldCount => oldCount + 1)
10   }
11   // TODO: add another button that decrements the count
12
13   return (
14     <div>
15       <p>current count: {count}</p>
16       <button onClick={onIncrementClick}>Click to increment</button>
17     </div>
18   )
19 }
```

Counter component

```
1  import ReactDOM from 'react-dom'
2  import React, { Component, useState } from 'react'
3  import PropTypes from 'prop-types'
4
5  const Counter = props => {
6    const [count, setCount] = useState(props.start)
7
8    const onIncrementClick = () => {
9      setCount(oldCount => oldCount + 1)
10   }
11   // TODO: add another button that decrements the count
12
13   return (
14     <div>
15       <p>current count: {count}</p>
16       <button onClick={onIncrementClick}>Click to increment</button>
17     </div>
18   )
19 }
```

Counter component

```
1  import ReactDOM from 'react-dom'
2  import React, { Component, useState } from 'react'
3  import PropTypes from 'prop-types'
4
5  const Counter = props => {
6    const [count, setCount] = useState(props.start)
7
8    const onIncrementClick = () => {
9      setCount(oldCount => oldCount + 1)
10   }
11   // TODO: add another button that decrements the count
12
13   return (
14     <div>
15       <p>current count: {count}</p>
16       <button onClick={onIncrementClick}>Click to increment</button>
17     </div>
18   )
19 }
```

Counter component

```
1  import ReactDOM from 'react-dom'
2  import React, { Component, useState } from 'react'
3  import PropTypes from 'prop-types'
4
5  const Counter = props => {
6    const [count, setCount] = useState(props.start)
7
8    const onIncrementClick = () => {
9      setCount(oldCount => oldCount + 1)
10   }
11   // TODO: add another button that decrements the count
12
13   return (
14     <div>
15       <p>current count: {count}</p>
16       <button onClick={onIncrementClick}>Click to increment</button>
17     </div>
18   )
19 }
```

useState

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

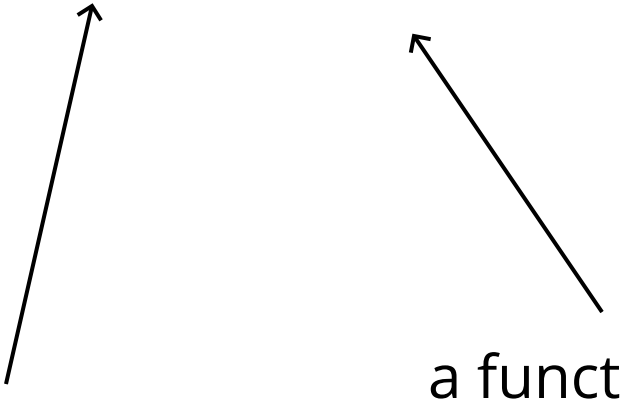
these are equivalent

but destructuring is much nicer

```
const countState = useState(0)
const count = countState[0]
const setCount = countState[1]
```

useState

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



the value itself

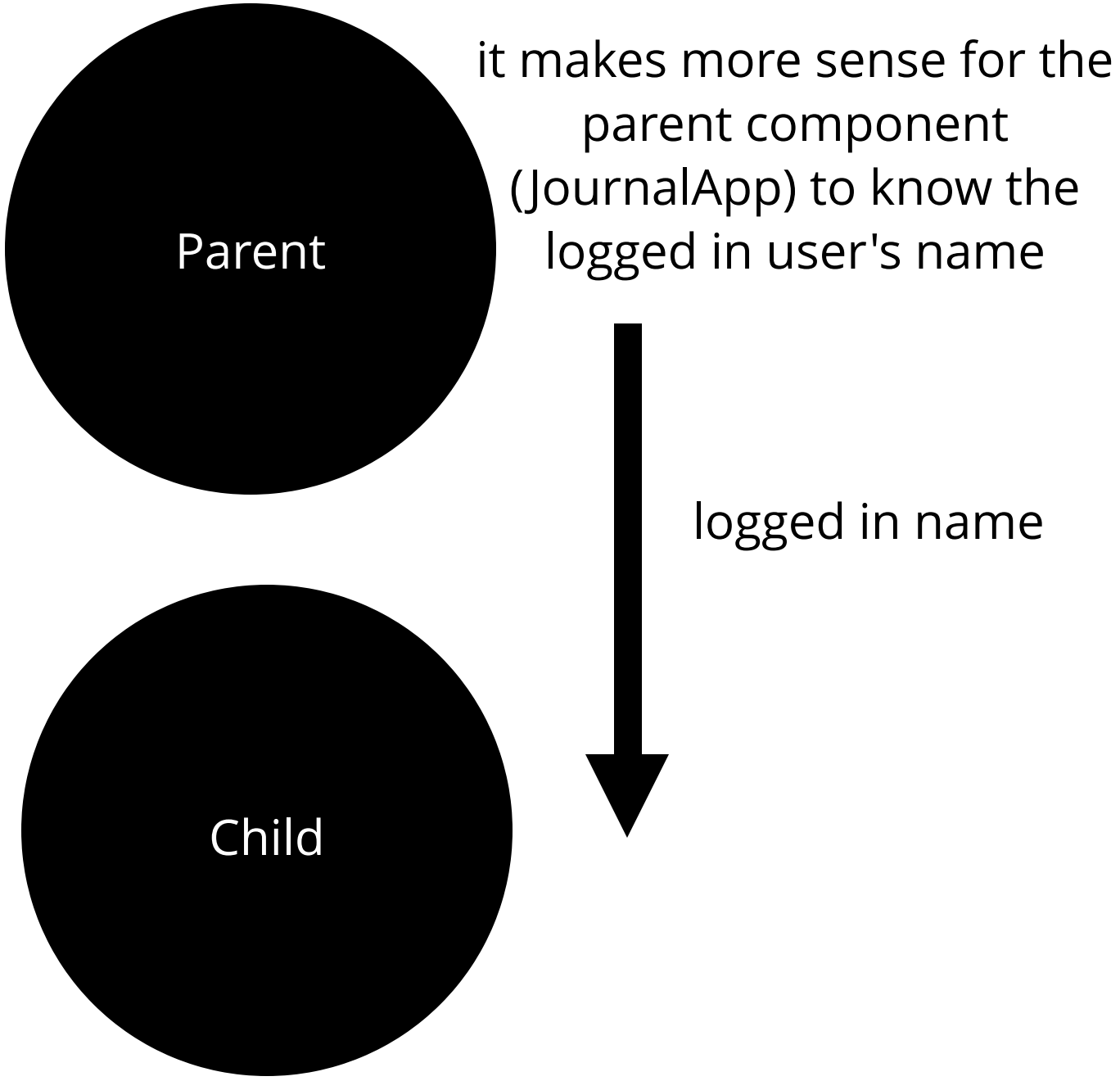
a function we use to set the value

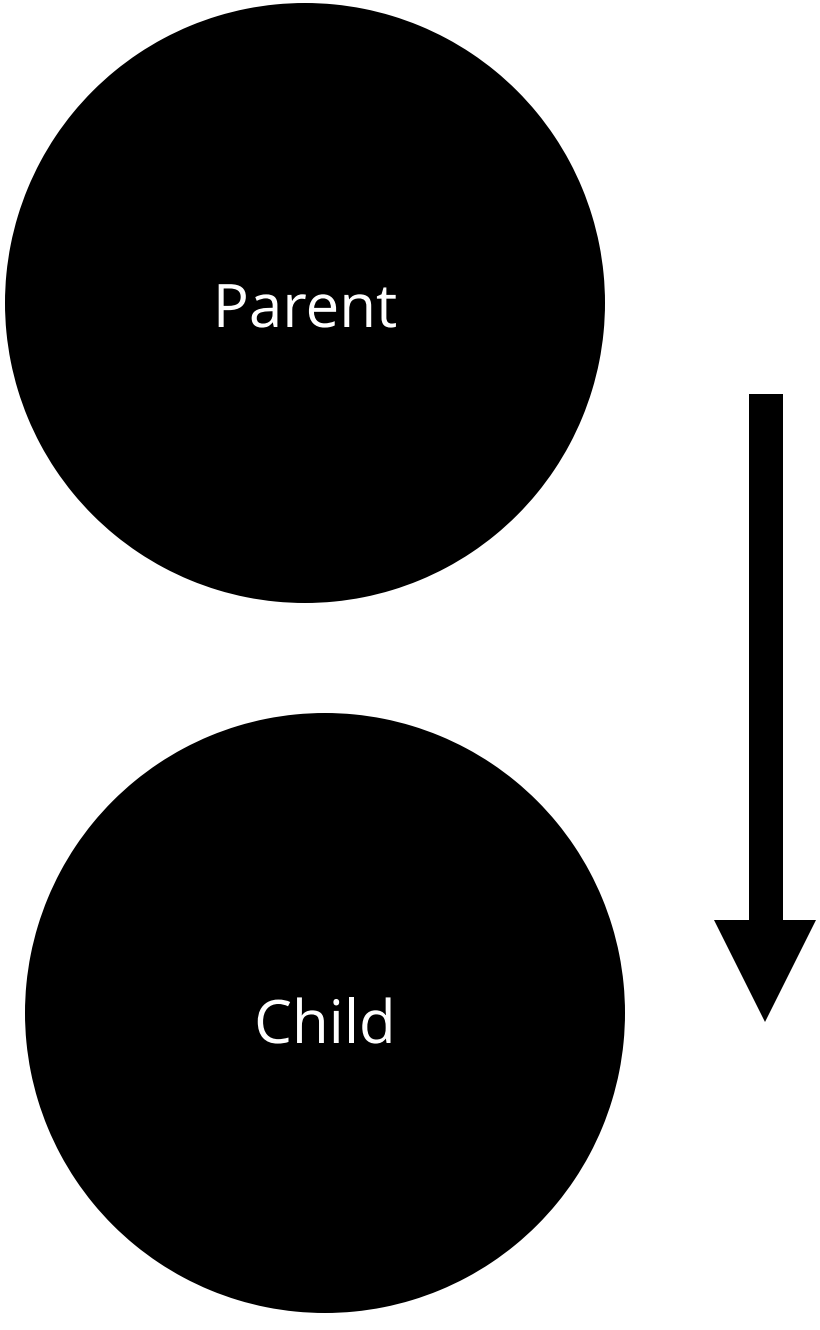
```jsx
const JournalHeader = () => {
  // TODO: can you get rid of the `name` constant, and instead create some
  // state using useState ? and then when the login button is clicked, set
  // name of the logged in person

  const name = 'Jack'

  const login = () => {
    console.log('I was clicked!')
  }

  return (
    <div className="journal-header-wrapper">
      <h1 className="journal-header">Journal App</h1>
      <h2 className="journal-subheader">Journal for {name}</h2>
      <button className="journal-login" onClick={login}>
        Login
      </button>
    </div>
  )
}
```

Exercise 7

Passing state to child components

```
const Parent = () => {
  return <Child />
}

const JournalApp = () => {
  return <JournalHeader />
}
```


Parent
it makes more sense for the parent component (JournalApp) to know the logged in user's name
logged in name
Child



this technique is known as lifting state up, and it's an important one to be comfortable with.

```
<div>
  <JournalHeader name={name} />
</div>
```

If we have `name` as a piece of state, we can pass that as a prop to the journal header

but: how can we now have the login button update the state? **Don't worry about this for now! That's the next exercise!**

for now: const [name, setName] = useState('Jack') lets you define the starting value of a piece of state

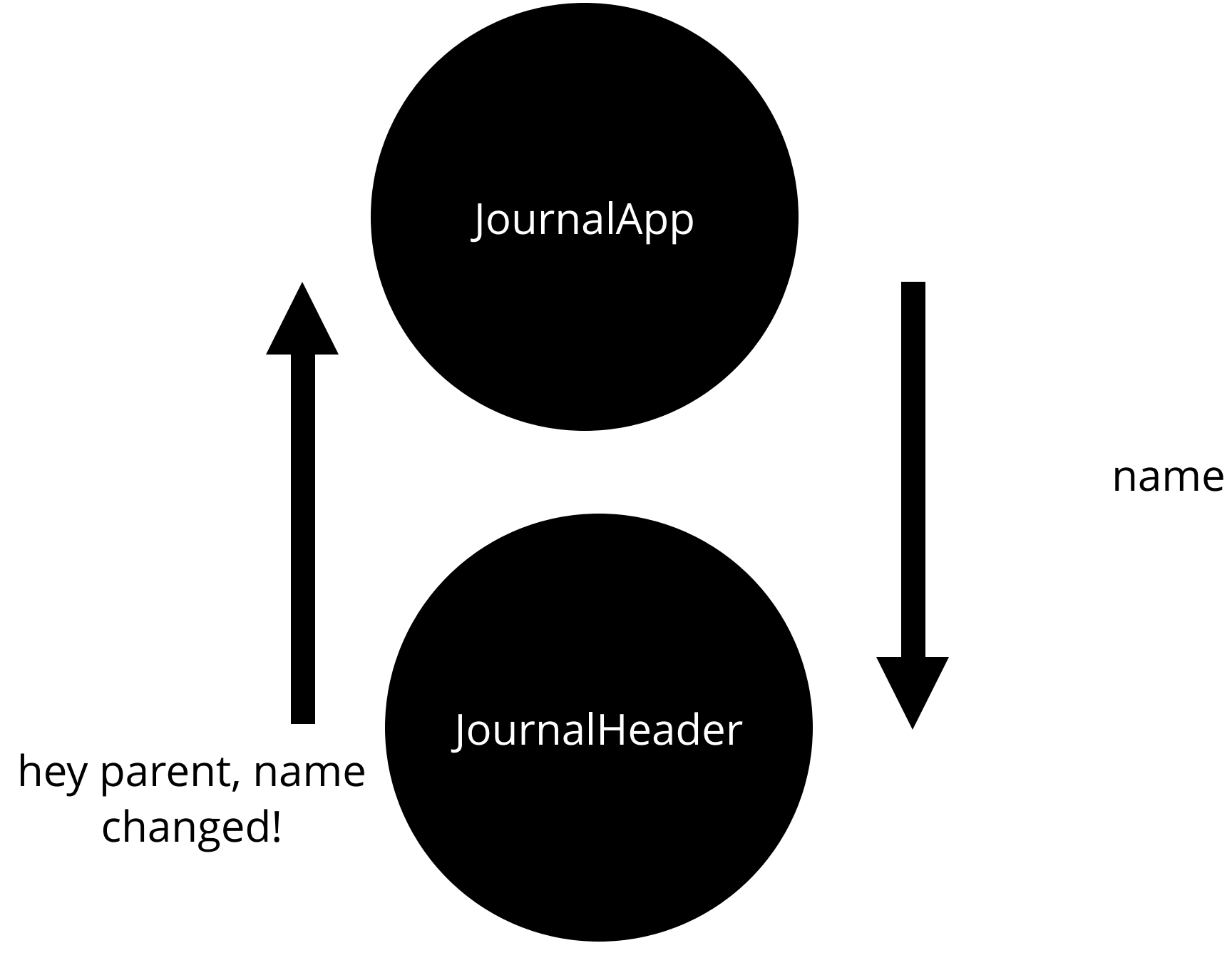
This is a very common, and powerful pattern.

Take state that's in the parent, and pass it as a prop to a child (or many children).

Parent and child communication

Sometimes we'll have state in the parent that we give to the child

And sometimes the child needs to let us know that the state has changed.


JournalApp
name
JournalHeader
hey parent, name changed!

```
<JournalHeader
  name={name}
  onNameChange={setName}
/>
```

here's some data for you to render

and when that data changes, this is how you tell me about it

Exercise 9

Rendering lists of data

```
const posts = [
  { id: 1, title: 'A fun day at the beach', date: '2019-04-10', body: '...' },
  { id: 2, ... },
  ...
]
```

often we have lists of data that we want to render

and we want to automatically render new posts if the length changes

mapping over arrays in JavaScript

```
const numbers = [1, 2, 3]

const newNumbers = numbers.map(function(x)
  return x * 2
})

// or

const newNumbers = numbers.map(x => x * 2)

newNumbers === [2, 4, 6]
```

so could we take our array of posts, and map them into an array of s ?

```
<ul>{posts.map((post, index) => {
  return (
    <li>{post.title}</li>
  )
})}</ul>
```

Exercise 10

Creating a <Post /> component to render a post

Creating a <Post /> component to render a post

(We will start splitting up our components into multiple files soon!)

Exercise 11

you should get familiar with spotting when a bunch of HTML that's being rendered belongs in a separate component. Don't fear having lots of little components!

Quick aside

Tooling

This workshop purposefully avoids talking about tooling and set up. We're focusing purely on React today (but questions are welcome!)

But I want to take a few minutes to quickly talk about some of the things going on in the workshop behind the scenes.

Bundler

A tool that takes all of our code and generates a bundle of HTML, CSS and JS for us.

Today I'm using https://parceljs.org/, but I'm also a big fan of https://webpack.js.org/

Transpiler

A tool that takes our modern JS and converts it back into JS that older browsers can use. Depending on what browsers you support, these will do different transforms.

Today we're using https://babeljs.io/ with some presets.

Talk to me about this later if you're interested :)

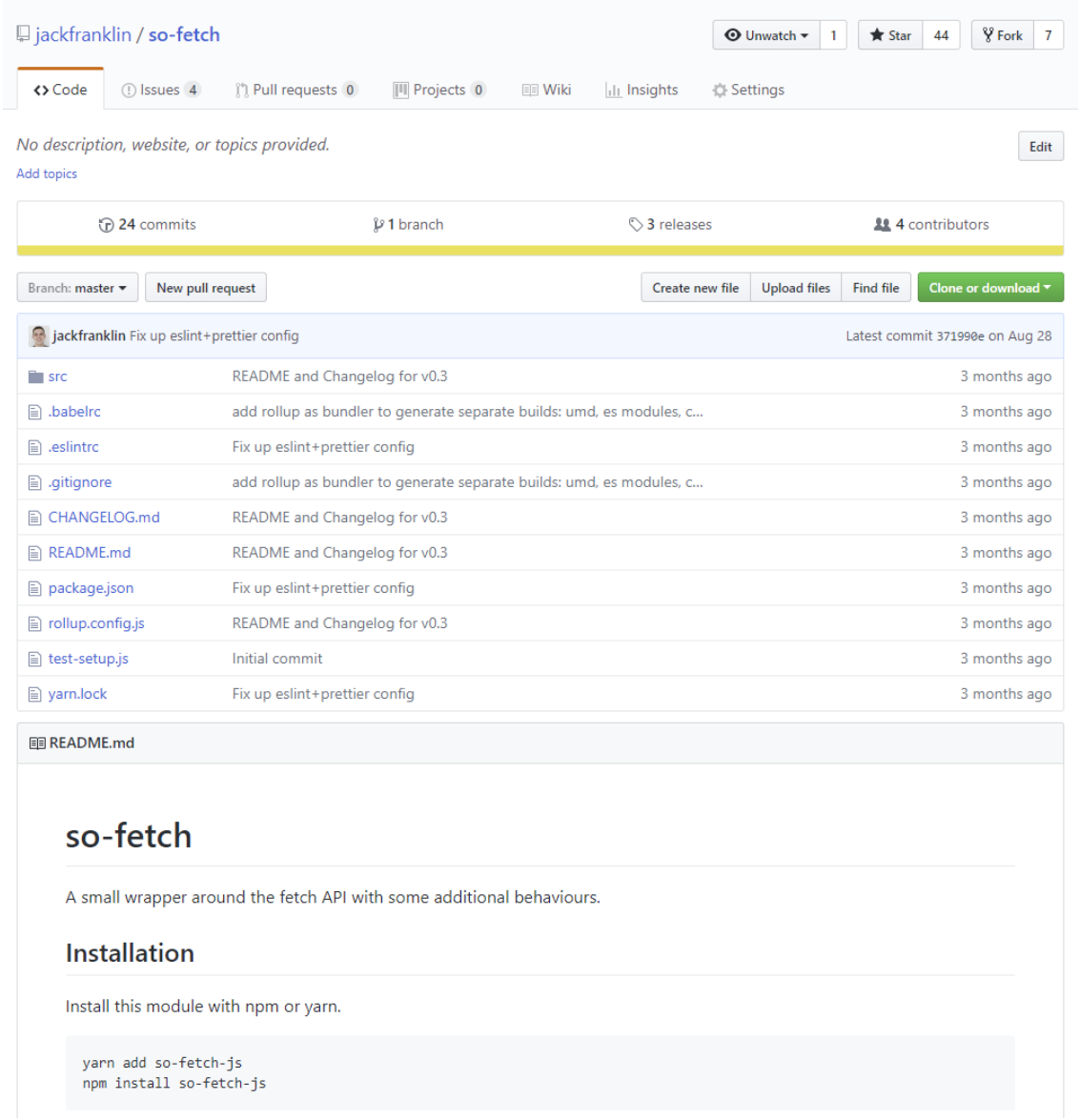
Code formatting

I find formatting code rather mundane and boring - so I let tools do it for me!

https://prettier.io/ is fantastic and there are editor plugins available for all popular editors.

Getting data from a source.

so-fetch-js



jackfranklin / so-fetch

Unwatch 1 | Star 44 | Fork 7

Code | Issues 4 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

No description, website, or topics provided.

Edit

Add topics

24 commits | 1 branch | 3 releases | 4 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

jackfranklin Fix up eslint+prettier config — Latest commit 371990e on Aug 28

src	README and Changelog for v0.3	3 months ago
.babelrc	add rollup as bundler to generate separate builds: umd, es modules, c...	3 months ago
.eslintrc	Fix up eslint+prettier config	3 months ago
.gitignore	add rollup as bundler to generate separate builds: umd, es modules, c...	3 months ago
CHANGELOG.md	README and Changelog for v0.3	3 months ago
README.md	README and Changelog for v0.3	3 months ago
package.json	Fix up eslint+prettier config	3 months ago
rollup.config.js	README and Changelog for v0.3	3 months ago
test-setup.js	Initial commit	3 months ago
yarn.lock	Fix up eslint+prettier config	3 months ago

README.md

so-fetch

A small wrapper around the fetch API with some additional behaviours.

Installation

Install this module with npm or yarn.

```
yarn add so-fetch-js
npm install so-fetch-js
```

```
import fetch from 'so-fetch-js'

fetch('/users').then(response => {
  console.log(response.data)
})
```

You all have a local API running on your machine :)

http://localhost:3000/posts

Before React hooks

Component lifecycle

https://reactjs.org/docs/react-component.html#the-component-lifecycle

Before React hooks componentDidMount

If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

https://reactjs.org/docs/react-component.html#componentdidmount

WITH React hooks 

useEffect

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you're used to calling these operations "side effects" (or just "effects"), you've likely performed them in your components before.

useEffect

```
const [posts, setPosts] = useState(null)

useEffect(() => {
  console.log('I get run on every render')
})
```

by default, useEffect runs on every single render

useEffect

```
const [posts, setPosts] = useState(null)

useEffect(() => {
  console.log('I get run on every render')
})
```

this can be dangerous and lets you easily
get into an infinite loop situation!

useEffect

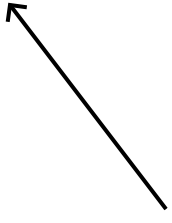
```
const [posts, setPosts] = useState(null)

useEffect(() => {
  console.log('I get run on every render')
})
```

so useEffect takes a second argument: a list of dependencies, or:

things that if they change, we should re-run our effect.

useEffect

```
const [posts, setPosts] = useState(null)

useEffect(() => {
  console.log('I get run on every render')
}, [])
```



empty array = there is nothing that would cause this effect to have to run again.

so it will only run once!

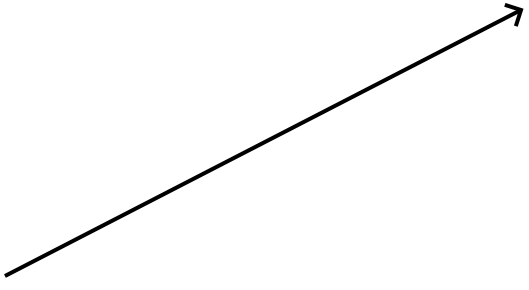
let's use useEffect to fetch some posts

Exercise 12

modelling the loading state and showing a spinner

default state of posts

```
const [posts, setPosts] = useState(null)
```



default state of null, not of an empty array

Conditional rendering in JSX

```
return post ? (
  <div>
    <h1>{post.title}</h1>
    <p>{post.body}</p>
  </div>
) : (
  <p>Loading</p>
)
```

you get used to the ternaries! They fit really nicely into JSX.

Update our journal to show a loading spinner whilst the posts are loading.

Hint: test the spinner by removing the `setPosts` call - else the posts load too quickly for you to see the spinner!

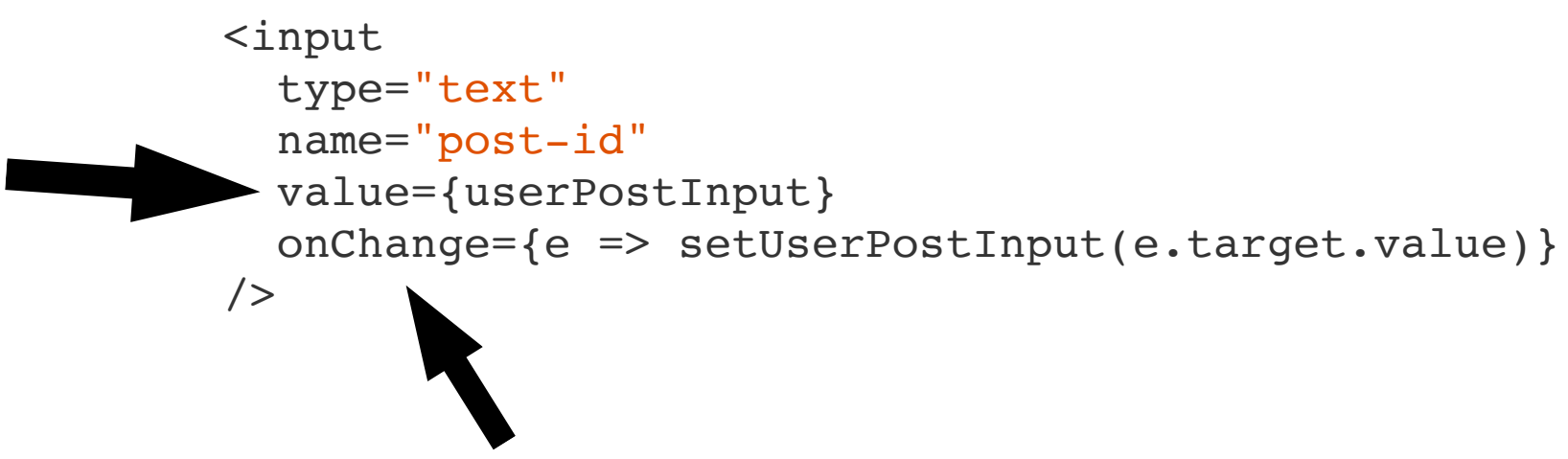
Exercise 13

forms in React

Controlled inputs

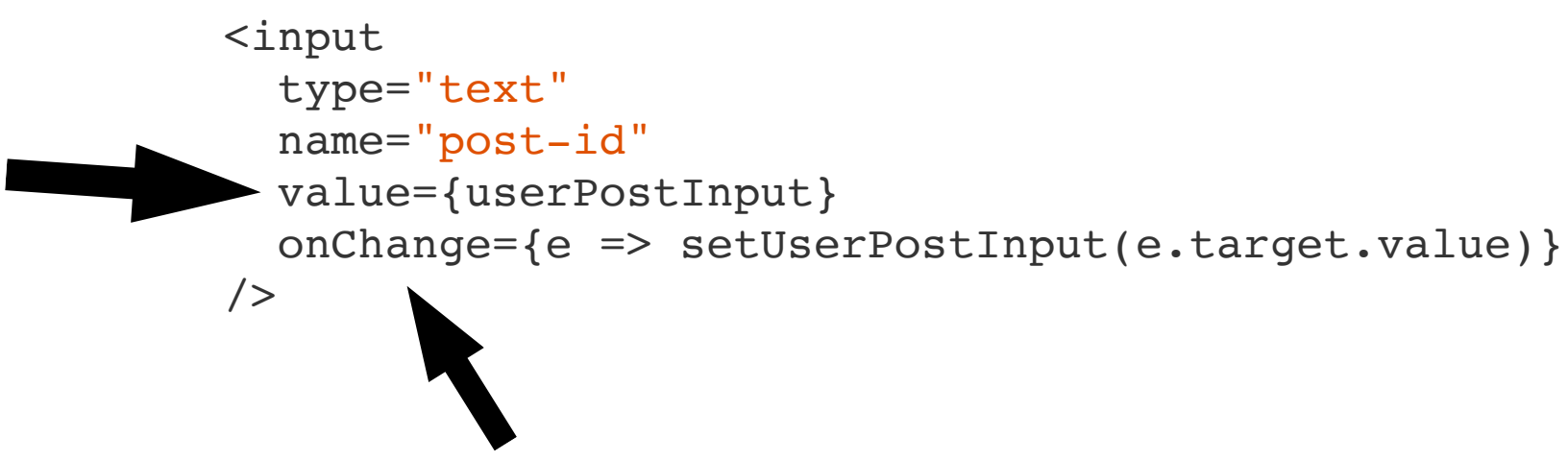
React controls the **value** of an input

And React controls the **onChange** event of an input.



value: the actual value of the input box

onChange: what React calls when the user types in the box, so we can update the value



e.target.value : the value inside the input at the time of the event

let's let the user log in and tell us their name!


Cancel
Login to your Journal.
jack
Login

```
<input
  type="text"
  name="post-id"
  value={userPostInput}
  onChange={e => setUserPostInput(e.target.value)}
/>
```

hook up the form in the login modal to log the user in

Exercise 14

Getting posts per user.

/posts?userId=1

fetches just posts for that user

PS: this is not a very secure API... 😱

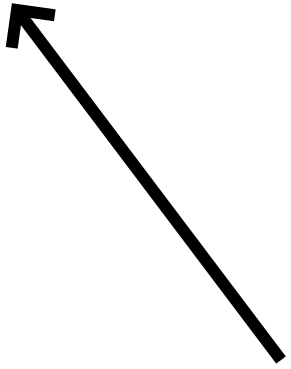
so, when a user logs in, let's get their posts

```
1    const userIdForName = (name) => {
2      return {
3        'alice': 1,
4        'bob': 2,
5        'charlotte': 3
6      }[name]
7    }
```

how do we make useEffect re-run when something changes?

the dependency array!

```
1  useEffect(() => {
2    const userId = userIdForName(name)
3    if (!userId) return
4
5    fetch(`http://localhost:3000/posts?userId=${userId}`).then(response => {
6      setPosts(response.data)
7    })
8  }, [name])
```



this is the important bit

Exercise 15

useEffect is incredibly powerful

multiple components in multiple files

ES2015 Modules

```
export default class Post extends Component {
    ...
}
```

```
import Post from './post'
```

My convention

File is named with the same name as the component, in lowercase.

posts.js = Posts component

Your convention

You should come up with your own rules that suit you and your team!

Let's tidy up our code

Can you extract the JournalHeader component into its own file?

Extracting components

The journal header now contains a modal which could exist on its own

This would be cleaner and nicer:

```
const JournalHeader = props => {
  const [isShowingModal, setIsShowingModal] = useState(false)

  const showModal = () => setIsShowingModal(true)

  return (
    <div className="journal-header-wrapper">
      <h1 className="journal-header">Journal App</h1>
      <h2 className="journal-subheader">Journal for {props.name}</h2
      <button className="journal-login" onClick={showModal}>
        Login
      </button>

      <LoginModal
        isShowing={isShowingModal}
        onSubmit={name => props.setName(name)}
        onClose={() => setIsShowingModal(false)}
      />
      )}
    </div>
  )
}
```

This would be cleaner and nicer:

```
const JournalHeader = props => {
  const [isShowingModal, setIsShowingModal] = useState(false)

  const showModal = () => setIsShowingModal(true)

  return (
    <div className="journal-header-wrapper">
      <h1 className="journal-header">Journal App</h1>
      <h2 className="journal-subheader">Journal for {props.name}</h2
      <button className="journal-login" onClick={showModal}>
        Login
      </button>

      <LoginModal
        isShowing={isShowingModal}
        onSubmit={name => props.setName(name)}
        onClose={() => setIsShowingModal(false)}
      />
      )}
    </div>
  )
}
```

This would be cleaner and nicer:

```
const JournalHeader = props => {
  const [isShowingModal, setIsShowingModal] = useState(false)

  const showModal = () => setIsShowingModal(true)

  return (
    <div className="journal-header-wrapper">
      <h1 className="journal-header">Journal App</h1>
      <h2 className="journal-subheader">Journal for {props.name}</h2
      <button className="journal-login" onClick={showModal}>
        Login
      </button>

      <LoginModal
        isShowing={isShowingModal}
        onSubmit={name => props.setName(name)}
        onClose={() => setIsShowingModal(false)}
      />
      )}
    </div>
  )
}
```

```
const JournalHeader = props => {
  const [isShowingModal, setIsShowingModal] = useState(false)

  const showModal = () => setIsShowingModal(true)

  return (
    <div className="journal-header-wrapper">
      <h1 className="journal-header">Journal App</h1>
      <h2 className="journal-subheader">Journal for {props.name}</h2>
      <button className="journal-login" onClick={showModal}>
        Login
      </button>

      <LoginModal
        isShowing={isShowingModal}
        onSubmit={name => props.setName(name)}
        onClose={() => setIsShowingModal(false)}
      />
      )}
    </div>
  )
}
```

can you extract LoginModal so this works?

Exercise 17

```
const JournalHeader = props => {
  const [isShowingModal, setIsShowingModal] = useState(false)

  const showModal = () => setIsShowingModal(true)

  return (
    <div className="journal-header-wrapper">
      <h1 className="journal-header">Journal App</h1>
      <h2 className="journal-subheader">Journal for {props.name}</h2>
      <button className="journal-login" onClick={showModal}>
        Login
      </button>

      <LoginModal
        isShowing={isShowingModal}
        onSubmit={name => props.setName(name)}
        onClose={() => setIsShowingModal(false)}
      />
      )}
    </div>
  )
}
```

can you extract LoginModal so this works?

Exercise 17

Showing posts

We'd like to update the <Post /> component so you can click on a post and expand it to see its contents.

We'd like to update the <Post /> component so you can click on a post and expand it to see its contents.

Can you introduce some state and a UI to the <Post /> component to make this happen?

Advanced React

If you stopped now, you'd be set to build lots of React apps.

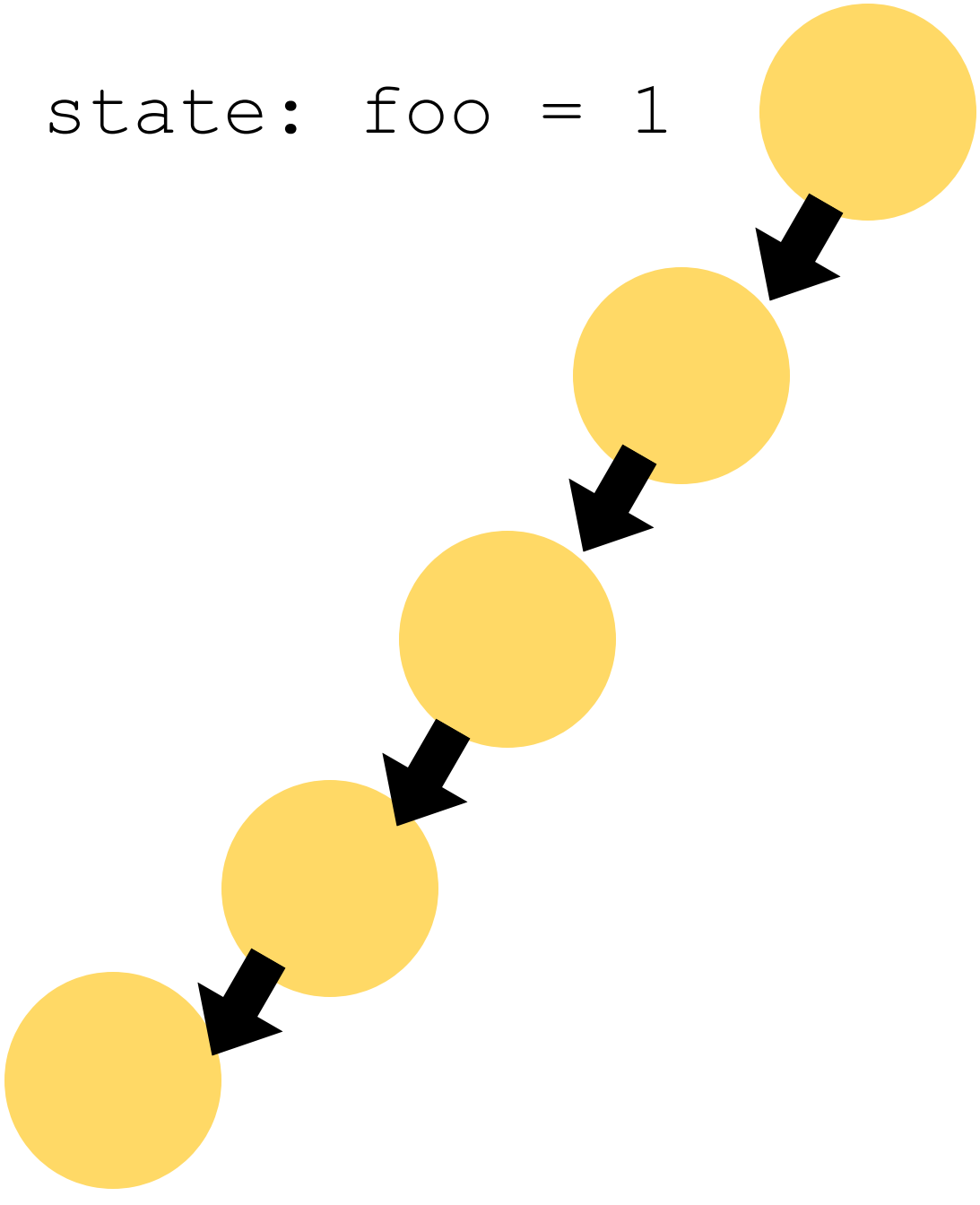
React's context API

Or, since hooks: the useContext hook.

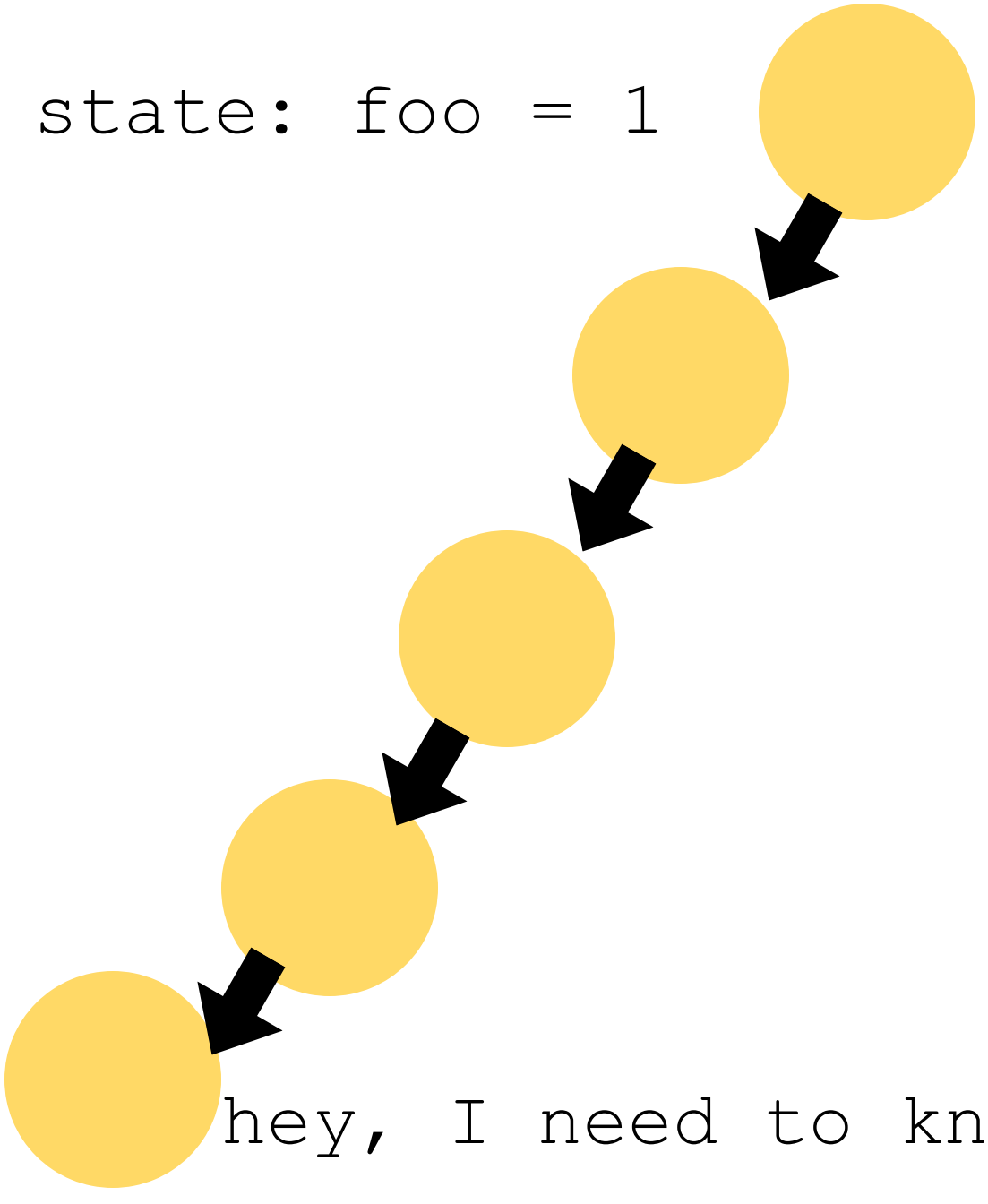
What is context?

A way to share data in a big tree of components

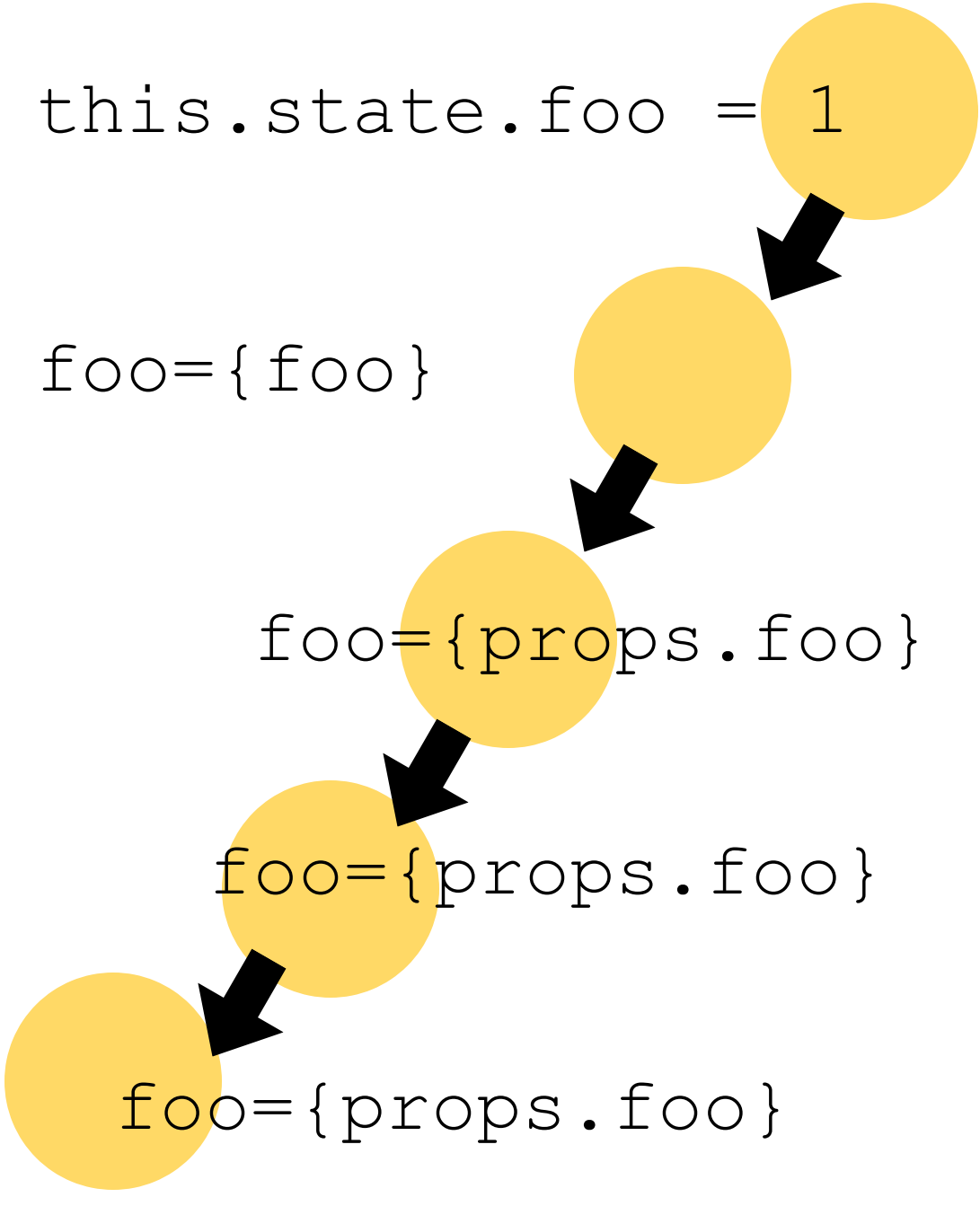
state: foo = 1

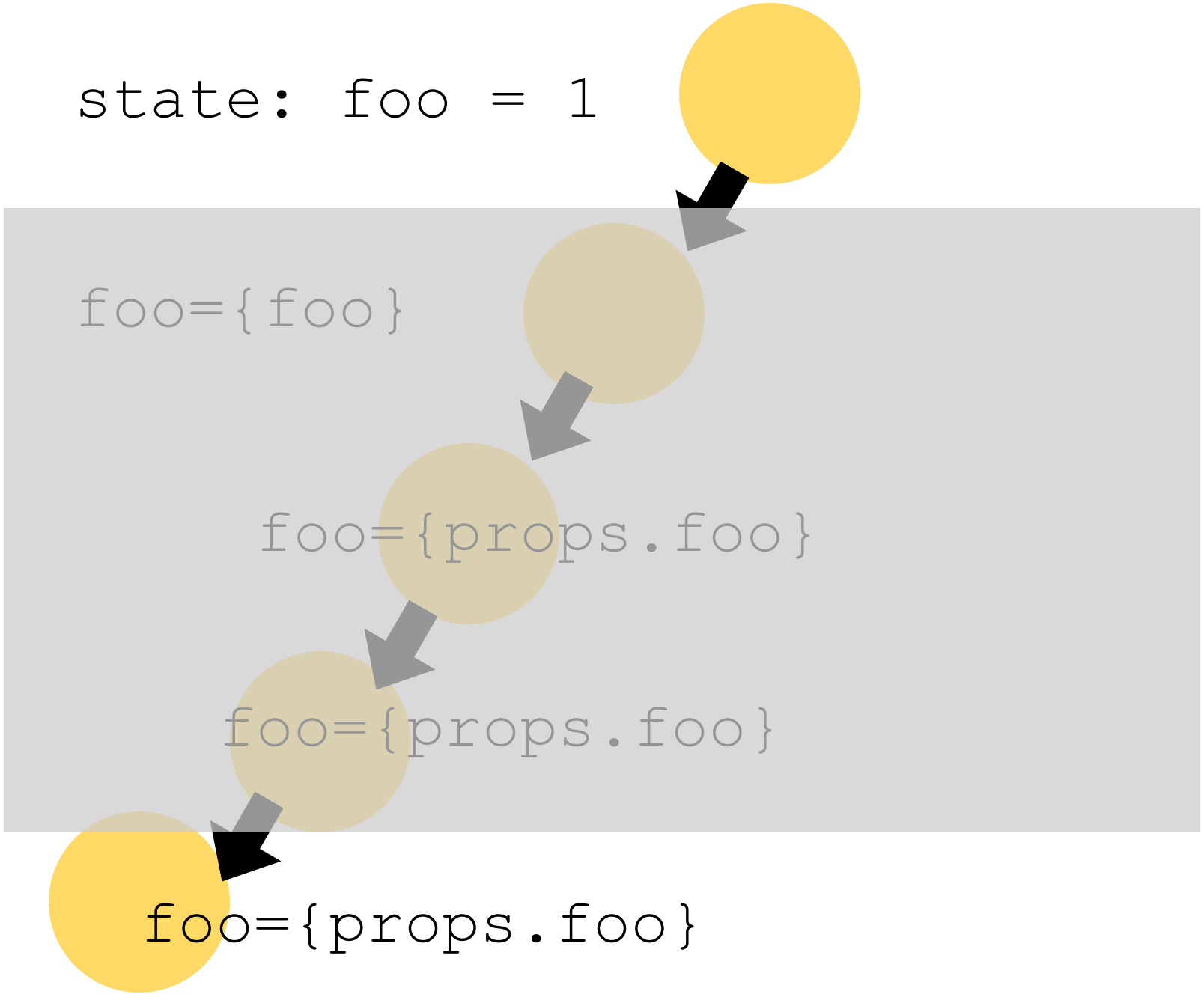


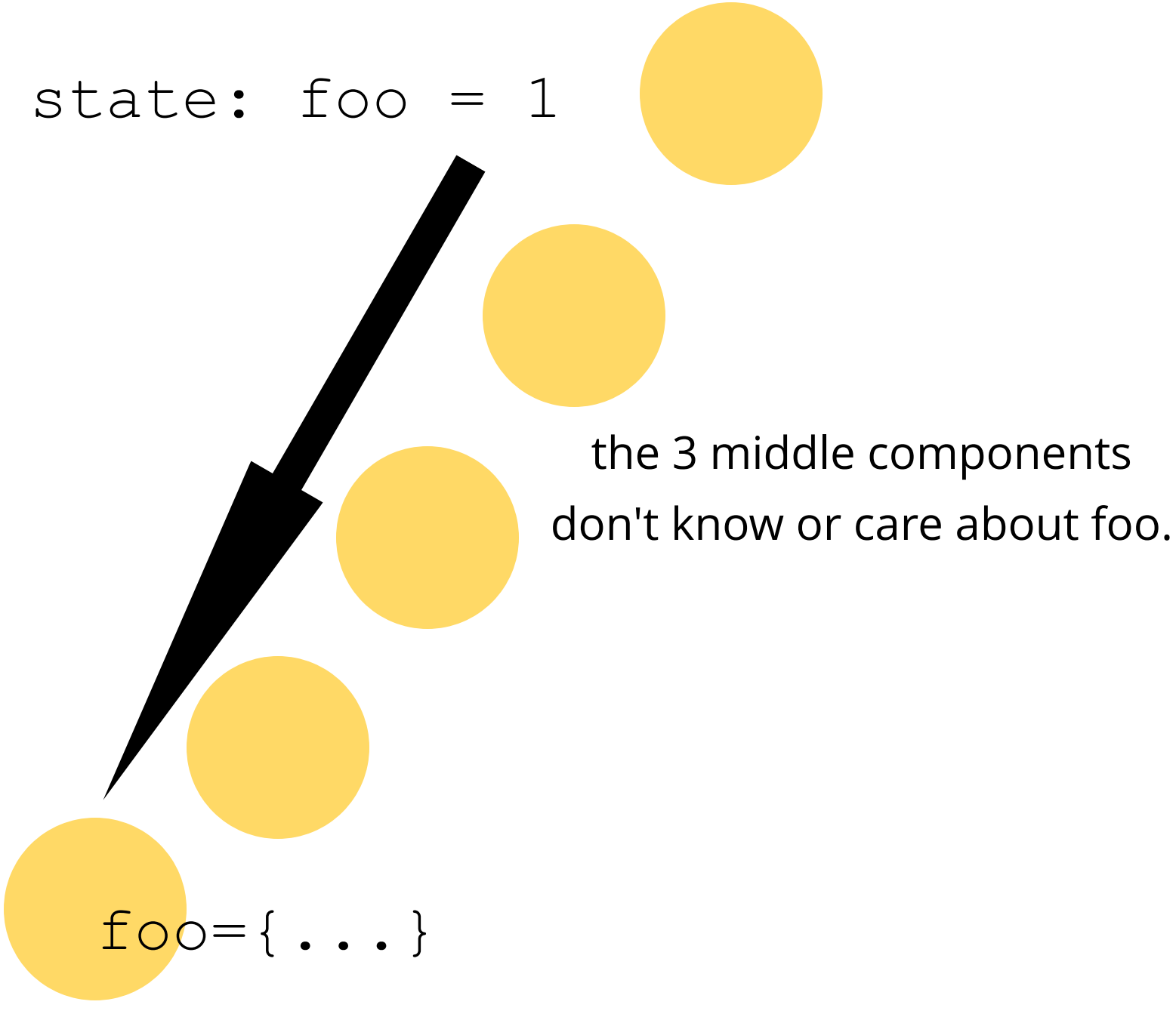
state: foo = 1



hey, I need to know `foo` !


this.state.foo = 1
foo={foo}
foo={props.foo}
foo={props.foo}
foo={props.foo}


state: foo = 1
foo={foo}
foo={props.foo}
foo={props.foo}
foo={props.foo}


state: foo = 1
the 3 middle components
don't know or care about foo.
foo={...}

```
const Context = React.createContext(defaultValue);
```

React lets us define contexts that allow data to be shared without having to pass it through every layer on the component tree.

```
<ThemeContext.Provider value={'light'}>
  <MyApp />
</ThemeContext.Provider>

// and then later on in any child component

<ThemeContext.Consumer>
  {theme => (
    <h1 style={{ color: theme === 'light' ? '#000' : '#fff' }}
      Hello world
    </h1>
  )}
</ThemeContext.Consumer>
```

Creating a context gives you:

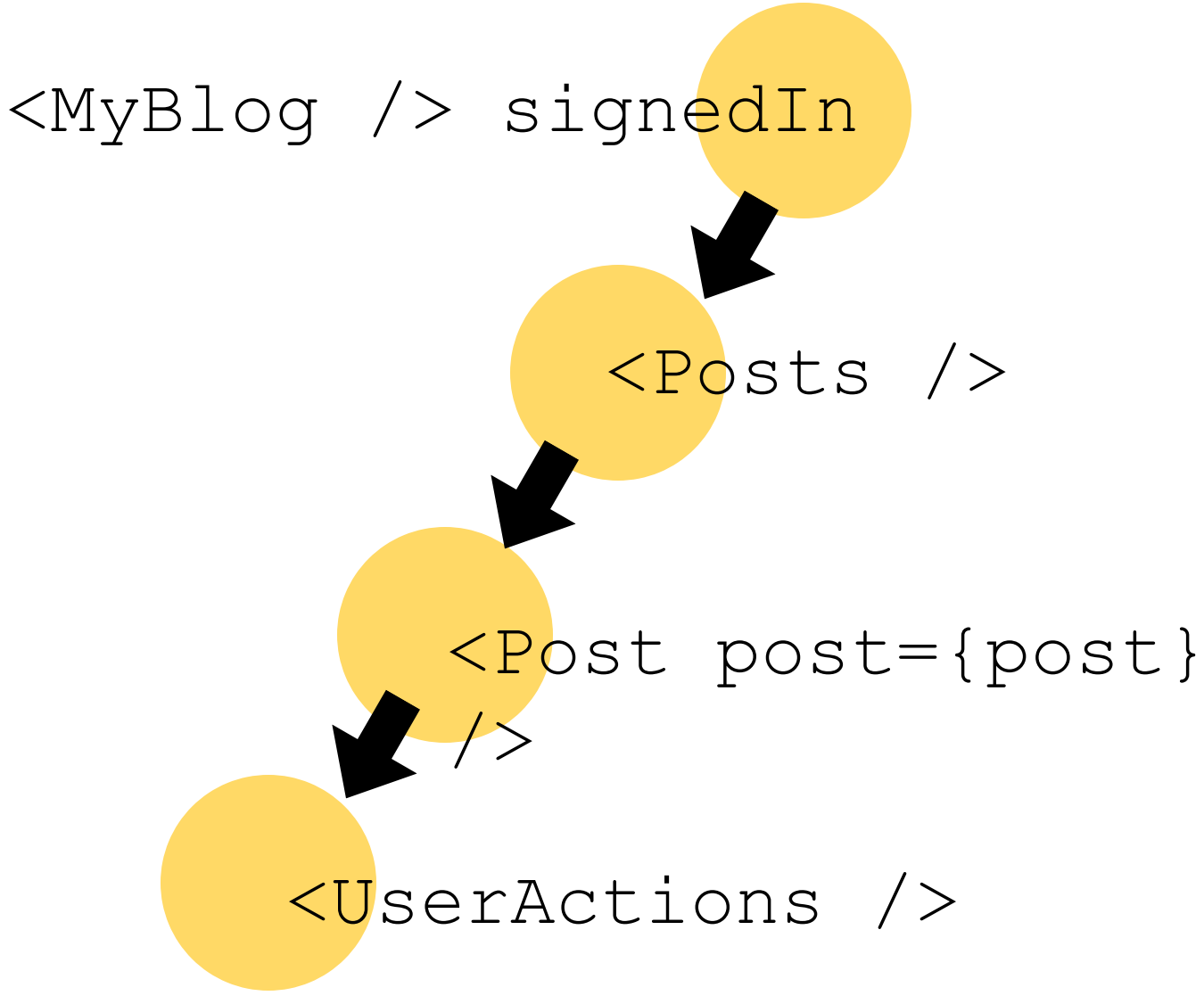
Provider

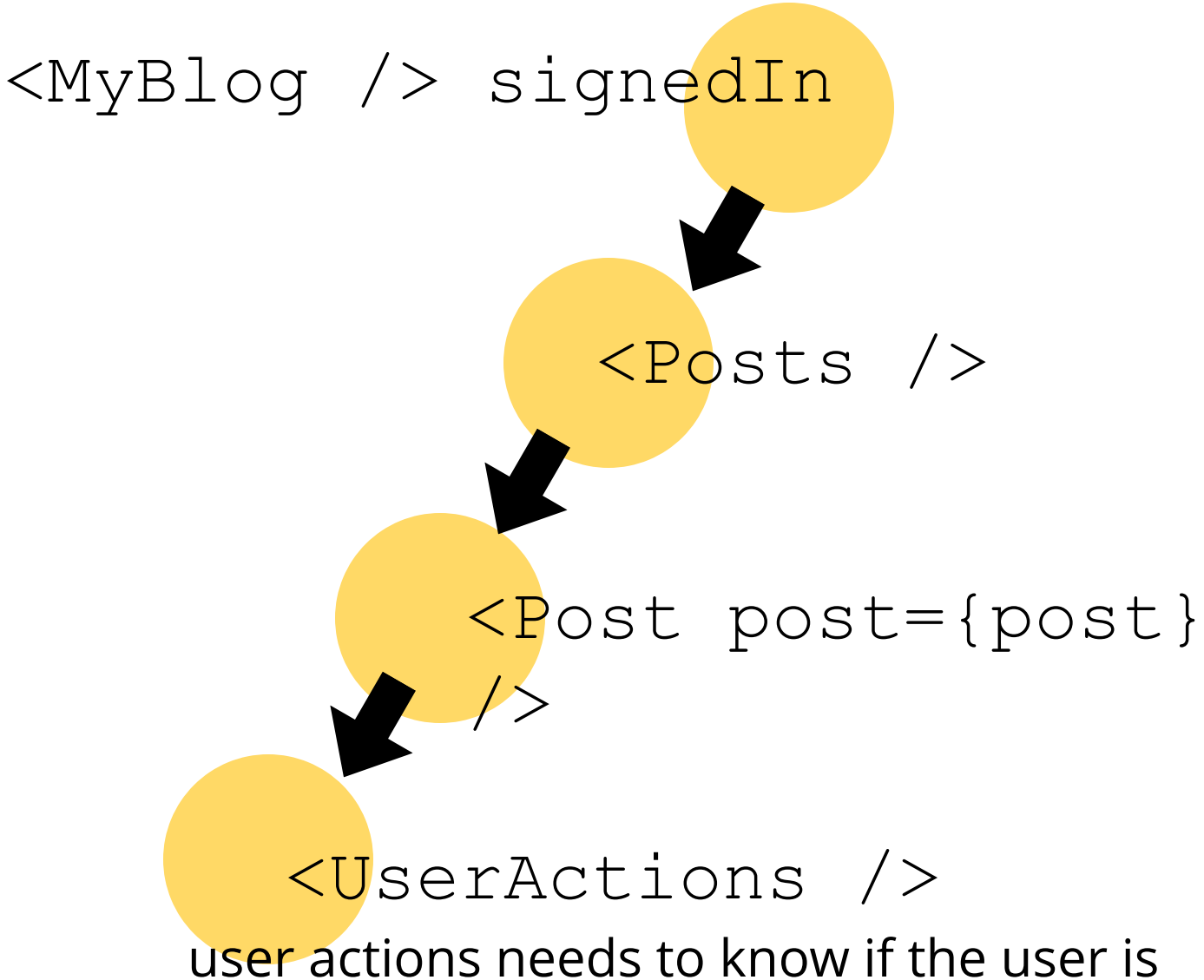
The component that makes a value accessible to all consumers in the component tree.

Consumer

The component that can read the value of a piece of state.

and you can read from a piece of context using the useContext hook! 🎉


<MyBlog /> signedIn
<Posts />
<Post post={post}
/>
<UserActions />



user actions needs to know if the user is signed in, to know if they can perform the actions

```
import React from 'react'

const AuthContext = React.createContext(false)

export default AuthContext
```

default value

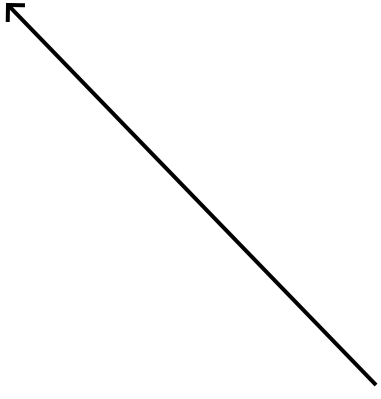
```
import AuthContext from './auth-context'
```

value for any consumers

```
   <div>
          <h1>Blog posts by Jack</h1>
          <AuthContext.Provider value={signedIn}>
            <Posts />
          </AuthContext.Provider>
        </div>
```

any consumer in <Posts /> or below can now consume our auth context

```
import AuthContext from './auth-context'
import React, { useContext } from 'react'

const signedIn = useContext(AuthContext)
```



get the latest value of the AuthContext

let's rework our user login system using context

AuthContext for our Journal

```
import { createContext } from 'react'

const AuthContext = createContext({
  loggedInUserName: null,
})

export default AuthContext
```

Create the context based off the name state.

```
const [name, setName] = useState('')

 const authContextValue = {
    loggedInUserName: name,
  }
```

Wrap our app in the provider.

```
<AuthContext.Provider value={authContextValue}>
  <JournalHeader name={name} setName={setName} />
  ...
</AuthContext.Provider>
```

Update <JournalHeader /> to read the name from the context, and not be passed it as a prop

```
import AuthContext from './auth-context'
import React, { useContext } from 'react'

const authContext = useContext(AuthContext)

const name = authContext.loggedInUserName
```

Exercise 19

Q: what are the pros and cons of context over passing props?

Updating the name via the context

Right now we read the name from context, but set it via a prop that's passed down. Let's fix that.

Update our AuthContext so it exposes the logged in name and a function that <JournalHeader /> can call to set it.

Avoiding additional work when possible.

Only updating the context if values we care about change.

Calculate context on *every render.*

```
console.log('Updating the context')
const authContextValue = {
  loggedInUserName: name,
  setLoggedInUser: setName,
}
```

But we only want to update the context if the `name` changes. So we're wasting effort here.

```
console.log('Updating the context')
const authContextValue = {
  loggedInUserName: name,
  setLoggedInUser: setName,
}
```

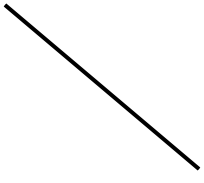
useMemo to the rescue

memo = memoization

useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

useMemo to the rescue

```
const authContextValue = useMemo(() => {
  console.log('Updating the context')
  return {
    loggedInUserName: name,
    setLoggedInUser: setName,
  }
}, [name])
```



the values that, when changed, should cause the context to be recalculated

(just like with useEffect!)

can you update the code to use useMemo?

```
const authContextValue = useMemo(() => {

}, [name])
```

Exercise 21

Custom hooks

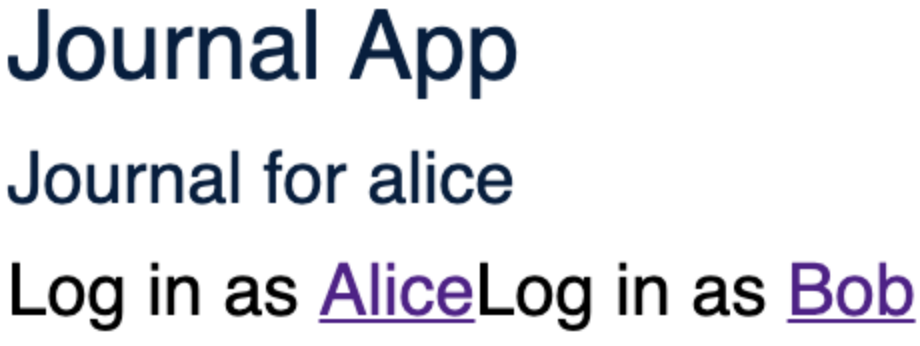
Hooks are powerful, but we can create our own hooks too

Let's say we want to build a hook that fetches posts and caches them

(This is going to not be a proper implementation - just a simple one for demo purchases!)

We've built a tool to let us "login" as a specific user.

(Our app is very secure...)



```
const posts = usePostsLoader(userId)
```

let's think about what we'd like our API to be.

Hooks are great for pulling out "boring" details and lets you not worry about how they work behind the scenes.

```
const usePostsLoader = userId => {
  const [postsCache, setPostsCache] = useState({})

  useEffect(() => {
    ...
  }, [userId])

  return postsCache[userId]
}
```

custom hooks *can use other hooks* internally!

```
postsCache: {
  1: [...],
  2: [...]
}

/* if we have the cache, return it
 * else make a network request.
 */
```

custom hooks *can use other hooks* internally!

```
const usePostsLoader = userId => {
  const [postsCache, setPostsCache] = useState({})

  useEffect(() => {
    if (!userId) return

    if (postsCache[userId]) {
      return
    } else {
      fetch(`http://localhost:3000/posts?userId=${userId}`).then
        setPostsCache(cache => ({
          ...cache,
          [userId]: response.data,
        }))
      })
    }
  }, [userId, postsCache])

  return postsCache[userId]
}
```

the last line of a custom hook should be what the hook returns. This can be anything you want!

```
const usePostsLoader = userId => {
  const [postsCache, setPostsCache] = useState({})

  useEffect(() => {
    if (!userId) return

    if (postsCache[userId]) {
      return
    } else {
      fetch(`http://localhost:3000/posts?userId=${userId}`).then(response => {
        setPostsCache(cache => ({
          ...cache,
          [userId]: response.data,
        }))
      })
    }
  }, [userId, postsCache])

  return postsCache[userId]
}
```

your task: implement the usePostsLoader!

Exercise 22

Controlled vs uncontrolled components

An uncontrolled component *controls its own state.*

A controlled component *is told what its state is.*

Why is this important?

Our app now lets us toggle between showing all posts, or just published posts.

we have an *uncontrolled* toggle component

```
1 const Toggle = props => {
2   const [on, setOn] = useState(false)
3
4   useEffect(() => {
5     props.onToggleChange(on)
6   }, [on, props])
7
8   return <span onClick={() => setOn(o => !o)}>{on ? 'YES' : 'NO'}</span>
9 }
```

so in the main app we have the ` publishedOnly ` state

```
1   const [publishedOnly, setPublishedOnly] = useState(false)
2
3
4   Only published posts? <Toggle onToggleChange={setPublishedOnly} />
```

spot the problem here

```
1   const [publishedOnly, setPublishedOnly] = useState(false)
2
3
4   Only published posts? <Toggle onToggleChange={setPublishedOnly} />
```

the same state is in two places

this is a recipe for bugs!

lets make the component *controlled*

```
1 const Toggle = props => {
2
3   return (
4     <span onClick={() => props.onChange(!props.on)}>
5       {props.on ? 'YES' : 'NO'}
6     </span>
7   )
8 }
```

no state in sight!

a prop tells us if we are on or not

we have a prop to tell our parent when we should change the state

lets make the component *controlled*

```
1 const Toggle = props => {
2
3   return (
4     <span onClick={() => props.onChange(!props.on)}>
5       {props.on ? 'YES' : 'NO'}
6     </span>
7   )
8 }
```

no state in sight!

a prop tells us if we are on or not

we have a prop to tell our parent when we should change the state

lets make the component *controlled*

no state in sight!

a prop tells us if we are on or not

a prop to tell our parent
when we should change the
state

Exercise 23

More hooks

useRef

useRef to get at DOM elements

If you pass a ref object to React with , React will set its .current property to the corresponding DOM node whenever that node changes.

Autofocusing the login box

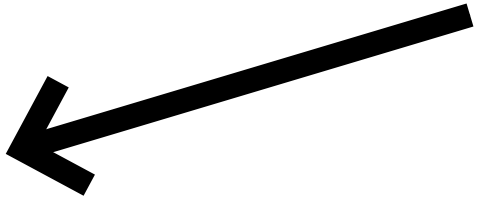
If you pass a ref object to React with , React will set its .current property to the corresponding DOM node whenever that node changes.

Autofocusing the login box

```
1 const inputEl = useRef(null)
```

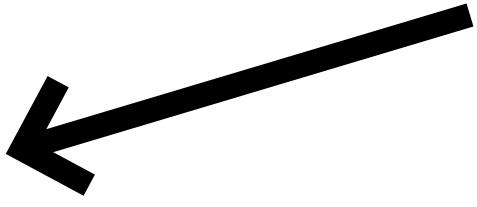
Autofocusing the login box

```
1 <input
2   type="text"
3   ref={inputEl}
4   value={loginName}
5   placeholder="jack"
6   onChange={e => setLoginName
7 />
```



Autofocusing the login box

```
1 <input
2   type="text"
3   ref={inputEl}
4   value={loginName}
5   placeholder="jack"
6   onChange={e => setLoginName
7 />
```



And how do we focus it? useEffect!

```
1   useEffect(() => {
2     if (inputEl.current && props.isShowing) {
3       inputEl.current.focus()
4     }
5   }, [inputEl, props.isShowing])
```

And how do we focus it? useEffect!

```
1   useEffect(() => {
2     if (inputEl.current && props.isShowing) {
3       inputEl.current.focus()
4     }
5   }, [inputEl, props.isShowing])
```

Focus the input when I load the modal

```
1   useEffect(() => {
2     if (inputEl.current && props.isShowing) {
3       inputEl.current.focus()
4     }
5   }, [inputEl, props.isShowing])
```

Focus the input when I load the modal

```
1   useEffect(() => {
2     if (inputEl.current && props.isShowing) {
3       inputEl.current.focus()
4     }
5   }, [inputEl, props.isShowing])
```

useRef can hold onto any value

From the docs...

The “ref” object is a generic container whose current property is mutable and can hold any value, similar to an instance property on a class.

Why is this useful?

```
useEffect(() => {
  const timer = setTimeout(() => {
    setState(count => count + 1)
  }, 1000)

  return () => clearTimeout(timer)
})
```

normally when you clear out timers via useEffect, you do so by returning an unsubscribe function.

But what if we want to allow the user to click to clear the timer?

```
useEffect(() => {
  const timer = setTimeout(() => {
    setState(count => count + 1)
  }, 1000)

  return () => clearTimeout(timer)
})
```

how do we get at the timer ID from an event handler?

But what if we want to allow the user to click to clear the timer?

```
const countTimerId = useRef(null)

useEffect(() => {
  const timer = setTimeout(() => {
    setState(count => count + 1)
  }, 1000)

  countTimerId.current = timer

  return () => clearTimeout(timer)
})
```

we can store the timer ID in a ref!

But what if we want to allow the user to click to clear the timer?

```
const stopCounting = () => {
  clearTimeout(countTimerId.current)
}
```

we can store the timer ID in a ref!

Extracting context usage into hooks

Hooks are great for hiding implementation details away.

```
1 const { loggedInUserName, setLoggedInUser } = useContext(AuthContext)
```

this doesn't feel like we're hiding things away.

```
1 const { loggedInUserName, setLoggedInUser } = useContext(AuthContext)
2
3 // compared to:
4
5 const { loggedInUserName, setLoggedInUser } = useAuth()
6
```

this feels better (fewer implementation details)

```
1 import { useContext } from 'react'
2 import AuthContext from './auth-context'
3
4 const useAuth = () => {
5   const context = useContext(AuthContext)
6   return context
7 }
8
9 export { useAuth }
```

let's use our useAuth hook

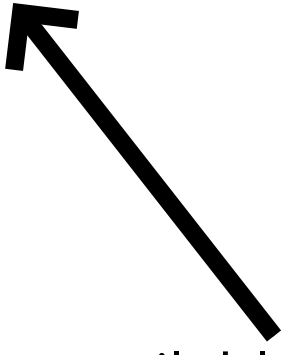
Hiding more context details in the hook.

Hiding the creation of AuthContext

```
1 const AuthContext = createContext()
2
3 const useAuth = () => {
4   const context = useContext(AuthContext)
5   return context
6 }
```

We'll need to create a provider.

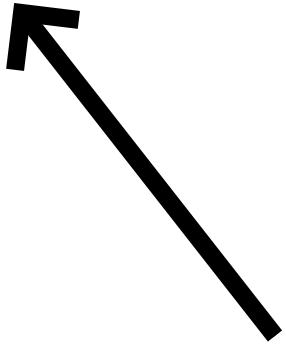
```
1 return (
2     <div>
3       <AuthContext.Provider value={authContextValue
4         <JournalHeader />
5         ...
```



we don't have this Provider available in our components now the context is created in use-auth.js

props.children

```
1 <MyCustomComponent>
2   <p>hello world</p>
3 </MyCustomComponent>
```



you can refer to the given children as props.children within a component

this allows your custom component to take children and render them, but wrapped in something

our AuthProvider

```
1. Declare the initial value for the auth context
2. Have a piece of state that can be updated to update the context.
3. Wraps the children it is given in the AuthContext.Provider component.
```

our AuthProvider

```
1  const AuthProvider = props => {
2    const [loggedInUserName, setLoggedInUserName] = useState('')
3
4    const authContext = useMemo(() => {
5      return {
6        loggedInUserName,
7        setLoggedInUserName,
8      }
9    }, [loggedInUserName])
10
11   const { children, ...otherProps } = props
12
13   return (
14     <AuthContext.Provider value={authContext} {...otherProps}>
15       {children}
16     </AuthContext.Provider>
17   )
18 }
```

And then wrap our app in this provider

```
ReactDOM.render(
  <AuthProvider>
    <JournalApp />
  </AuthProvider>,
  document.getElementById('react-root')
)
```

let's walk through this in code

there's no TODO here - we're going to walk through the code on screen and play with it.

Higher order components

Higher order functions

Higher order functions

```
1 const adder = x => y => x + y
2
3 const adder = function(x) {
4   return function(y) {
5     return x + y
6   }
7 }
```

Higher order functions

```
1 const adder = x => y => x + y
2
3 const addTwo = adder(2)
4
5 addTwo(3) // 5
```

Higher order components

A higher order component is (slightly confusingly) a *function that returns a React component.*

Higher order components

Why is this useful?

Remember the AuthProvider from the previous exercise?

Rather than make the end user wrap their component in <AuthProvider />, we could provide a function that does this for them.

```
1 const wrapWithAuth = Component => {
2   return Component
3 }
```

```
1 const wrapWithAuth = Component => {
2   return Component
3 }
4
5
6 // usage:
7 const JournalWithAuth = wrapWithAuth(JournalApp)
```

```
1 const wrapWithAuth = Component => {
2   const ComponentWrapped = props => {
3
4   }
5
6   return ComponentWrapped
7 }
```

```
const wrapWithAuth = Component => {
  const ComponentWrapped = props => {
     return (
       <AuthProvider>
         <Component {...props} />
       </AuthProvider>
     )
  }

  return ComponentWrapped
}

```

```
1  const wrapWithAuth = Component => {
2    const ComponentWrapped = props => {
3       return (
4         <AuthProvider>
5           <Component {...props} />
6         </AuthProvider>
7       )
8    }
9
10   return ComponentWrapped
11 }
12
```

Exercise 28

useReducer

Redux?

From the docs

useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one

A counter component in React

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <Fragment>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </Fragment>
  );
}
```

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <Fragment>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </Fragment>
  );
}
```

A counter component in React

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
```

A counter component in React

```
function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <Fragment>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </Fragment>
  );
}
```

A counter component in React

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <Fragment>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </Fragment>
  );
}
```

A counter component in React

```
1  const initialState = {count: 0};
2
3  function reducer(state, action) {
4    switch (action.type) {
5      case 'increment':
6        return {count: state.count + 1};
7      case 'decrement':
8        return {count: state.count - 1};
9      default:
10       throw new Error();
11   }
12 }
13
14 function Counter({initialState}) {
15   const [state, dispatch] = useReducer(reducer, initialState);
16   return (
17     <Fragment>
18       Count: {state.count}
19       <button onClick={() => dispatch({type: 'increment'})}>+</button>
20       <button onClick={() => dispatch({type: 'decrement'})}>-</button>
21     </Fragment>
22   );
23 }
```

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <Fragment>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </Fragment>
  );
}
```

A counter component in React

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <Fragment>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </Fragment>
  );
}
```

A counter component in React

Don't reach for this too soon!

Don't assume you need this: start with useState and go for this if you then realise you need the structure.

One place where we have slightly more complex state is in our use-posts-loader

So let's update it to use a reducer internally.

```
1  const initialState = {}
2
3  const reducer = (state, action) => {
4
5  }
6
7  const usePostsLoader = userId => {
8    const [postsCache, setPostsCache] = useState({})
9
10   const [ state, dispatch ] = useReducer(reducer, initialState)
```

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'newPostsForUser':
      return {
        ...state,
        [action.userId]: action.posts,
      }
  }
}

// usage:

dispatch({
  type: 'newPostsForUser',
  userId: 1,
  posts: [...]
})
```

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'newPostsForUser':
      return {
        ...state,
        [action.userId]: action.posts,
      }
  }
}

// usage:

dispatch({
  type: 'newPostsForUser',
  userId: 1,
  posts: [...]
})
```

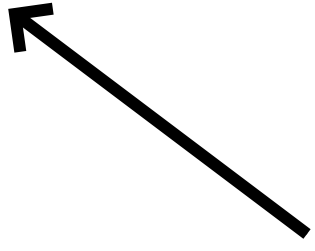
```
const reducer = (state, action) => {
  switch (action.type) {
    case 'newPostsForUser':
      return {
        ...state,
        [action.userId]: action.posts,
      }
  }
}

// usage:

dispatch({
  type: 'newPostsForUser',
  userId: 1,
  posts: [...]
})
```

```
useEffect(() => {
  if (!userId) return

  if (state[userId]) {
    return
  } else {
    fetch(`http://localhost:3000/posts?userId=${userId}`).then(response =>
      dispatch({
        type: 'newPostsForUser',
        userId: userId,
        posts: response.data,
      })
    })
  }
}, [userId, state])
```

```
useEffect(() => {
  if (!userId) return

  if (state[userId]) {
    return
  } else {
    fetch(`http://localhost:3000/posts?userId=${userId}`).then(response =>
      dispatch({
        type: 'newPostsForUser',
        userId: userId,
        posts: response.data,
      })
    })
  }
}, [userId, state])
```

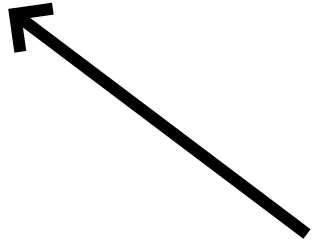
```
useEffect(() => {
  if (!userId) return

  if (state[userId]) {
    return
  } else {
    fetch(`http://localhost:3000/posts?userId=${userId}`).then(response =>
      dispatch({
        type: 'newPostsForUser',
        userId: userId,
        posts: response.data,
      })
    })
  }
}, [userId, state])
```

```
1  const usePostsLoader = userId => {
2    const [postsCache, setPostsCache] = useState({})
3
4    const [state, dispatch] = useReducer(reducer, initialState)
5
6    useEffect(() => {
7      ...
8    }, [userId, state])
9
10   return state[userId]
11 }
```



notice we now return from the redux state

```
1  const usePostsLoader = userId => {
2    const [postsCache, setPostsCache] = useState({})
3
4    const [state, dispatch] = useReducer(reducer, initialState)
5
6    useEffect(() => {
7      ...
8    }, [userId, state])
9
10   return state[userId]
11 }
```



notice we now return from the redux state

And it works 

Worth realising: we just entirely changed how we load and store our posts cache, and **none of our components even know** about it. This is one of the most powerful things about hooks.

Your turn ⬇

```
1   useEffect(() => {
2     if (!userId) return
3
4     if (state[userId]) {
5       return
6     } else {
7       fetch(`http://localhost:3000/posts?userId=${userId}`).then(response => {
8         dispatch({
9           type: 'newPostsForUser',
10          userId: userId,
11          posts: response.data,
12        })
13      })
14    }
15  }, [userId, state])
```

Exercise 29

Lifting state up: reducer style.

User

Which user of our system is currently logged in.

Posts

The posts that we are showing to the given user.

these two bits of state are quite directly related.

So let's lift the state up

Our main component
should house all this state.

We'll say goodbye to the use-posts-loader for now, but we can talk about how you'd bring it back once we've done the work.

We're also going to say goodbye to the useAuth context loader, and use the reducer state instead.

Both useReducer and useContext are useful tools

This exercise isn't me saying that I prefer useReducer over useContext, but showing you a different way you could solve the same problem. Each problem warrants a different solution - there is no direct rule on useReducer vs useContext

```
const initialState = {
  loggedInUser: {
    name: '',
  },
  postsForUser: [],
}

const reducer = (state, action) => {
  switch (action.type) {
    case 'logUserIn':
      return {
        ...state,
        loggedInUser: {
          name: action.newUserName,
        },
      }

    case 'gotPostsForUser':
      return {
        ...state,
        postsForUser: action.posts,
      }

    default: {
      console.error(`Unknown action! ${action}`)
      return state
    }
  }
}
```

```
const initialState = {
  loggedInUser: {
    name: '',
  },
  postsForUser: [],
}

const reducer = (state, action) => {
  switch (action.type) {
    case 'logUserIn':
      return {
        ...state,
        loggedInUser: {
          name: action.newUserName,
        },
      }

    case 'gotPostsForUser':
      return {
        ...state,
        postsForUser: action.posts,
      }

    default: {
      console.error(`Unknown action! ${action}`)
      return state
    }
  }
}
```

```
const initialState = {
  loggedInUser: {
    name: '',
  },
  postsForUser: [],
}

const reducer = (state, action) => {
  switch (action.type) {
    case 'logUserIn':
      return {
        ...state,
        loggedInUser: {
          name: action.newUserName,
        },
      }

    case 'gotPostsForUser':
      return {
        ...state,
        postsForUser: action.posts,
      }

    default: {
      console.error(`Unknown action! ${action}`)
      return state
    }
  }
}
```

```
const initialState = {
  loggedInUser: {
    name: '',
  },
  postsForUser: [],
}

const reducer = (state, action) => {
  switch (action.type) {
    case 'logUserIn':
      return {
        ...state,
        loggedInUser: {
          name: action.newUserName,
        },
      }

    case 'gotPostsForUser':
      return {
        ...state,
        postsForUser: action.posts,
      }

    default: {
      console.error(`Unknown action! ${action}`)
      return state
    }
  }
}
```

we now move loading posts back into a useEffect block that dispatches an action

```
1   useEffect(() => {
2     const userId = userIdForName(state.loggedInUser.name)
3     if (!userId) return
4
5     fetch(`http://localhost:3000/posts?userId=${userId}`).then(response
6       dispatch({
7         type: 'gotPostsForUser',
8         posts: response.data,
9       })
10    })
11  }, [state.loggedInUser.name])
```

we now move loading posts back into a useEffect block that dispatches an action

```
1   useEffect(() => {
2     const userId = userIdForName(state.loggedInUser.name)
3     if (!userId) return
4
5     fetch(`http://localhost:3000/posts?userId=${userId}`).then(response
6       dispatch({
7         type: 'gotPostsForUser',
8         posts: response.data,
9       })
10    })
11  }, [state.loggedInUser.name])
```

we now move loading posts back into a useEffect block that dispatches an action

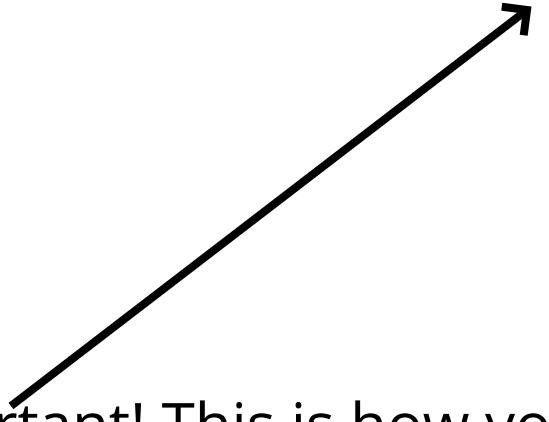
```
1   useEffect(() => {
2     const userId = userIdForName(state.loggedInUser.name)
3     if (!userId) return
4
5     fetch(`http://localhost:3000/posts?userId=${userId}`).then(response
6       dispatch({
7         type: 'gotPostsForUser',
8         posts: response.data,
9       })
10    })
11  }, [state.loggedInUser.name])
```

we now move loading posts back into a useEffect block that dispatches an action

```
1   useEffect(() => {
2     const userId = userIdForName(state.loggedInUser.name)
3     if (!userId) return
4
5     fetch(`http://localhost:3000/posts?userId=${userId}`).then(response
6       dispatch({
7         type: 'gotPostsForUser',
8         posts: response.data,
9       })
10    })
11  }, [state.loggedInUser.name])
```

and we now pass through the loggedIn state and the dispatch function to the JournalHeader

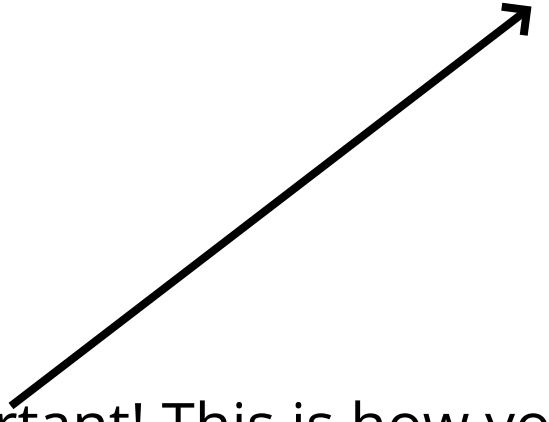
```
1 <JournalHeader loggedInUser={state.loggedInUser} dispatch={dispatch} />
```



this is important! This is how you allow child components to update the state.

and we now pass through the loggedIn state and the dispatch function to the JournalHeader

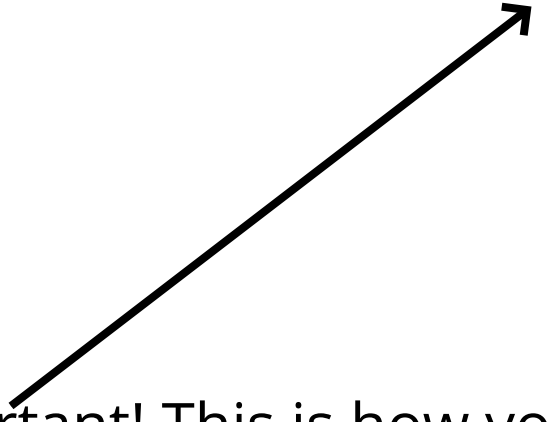
```
1 <JournalHeader loggedInUser={state.loggedInUser} dispatch={dispatch} />
```



this is important! This is how you allow child components to update the state.

and we now pass through the loggedIn state and the dispatch function to the JournalHeader

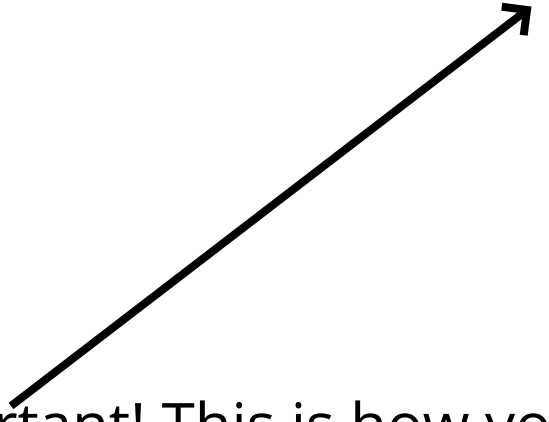
```
1 <JournalHeader loggedInUser={state.loggedInUser} dispatch={dispatch} />
```



this is important! This is how you allow child components to update the state.

and we now pass through the loggedIn state and the dispatch function to the JournalHeader

```
1 <JournalHeader loggedInUser={state.loggedInUser} dispatch={dispatch} />
```



this is important! This is how you allow child components to update the state.

And within JournalHeader we can dispatch an action when someone wants to log in.

```
1   const loggedInUserName = props.loggedInUser.name
2
3   const setLoggedInUser = name => {
4     props.dispatch({
5       type: 'logUserIn',
6       newUserName: name,
7     })
8   }
```

And within JournalHeader we can dispatch an action when someone wants to log in.

```
const loggedInUserName = props.loggedInUser.name

const setLoggedInUser = name => {
  props.dispatch({
    type: 'logUserIn',
    newUserName: name,
  })
}
```

And within JournalHeader we can dispatch an action when someone wants to log in.

```
const loggedInUserName = props.loggedInUser.name

const setLoggedInUser = name => {
  props.dispatch({
    type: 'logUserIn',
    newUserName: name,
  })
}
```

And within JournalHeader we can dispatch an action when someone wants to log in.

```
1   const loggedInUserName = props.loggedInUser.name
2
3   const setLoggedInUser = name => {
4     props.dispatch({
5       type: 'logUserIn',
6       newUserName: name,
7     })
8   }
```

No exercise here, just have a play with the code locally.

What do you think are the pros and cons of useReducer at the top level like this?

Should our publishedOnly toggle state live in the reducer state? What do you think?

Advanced React: fin!