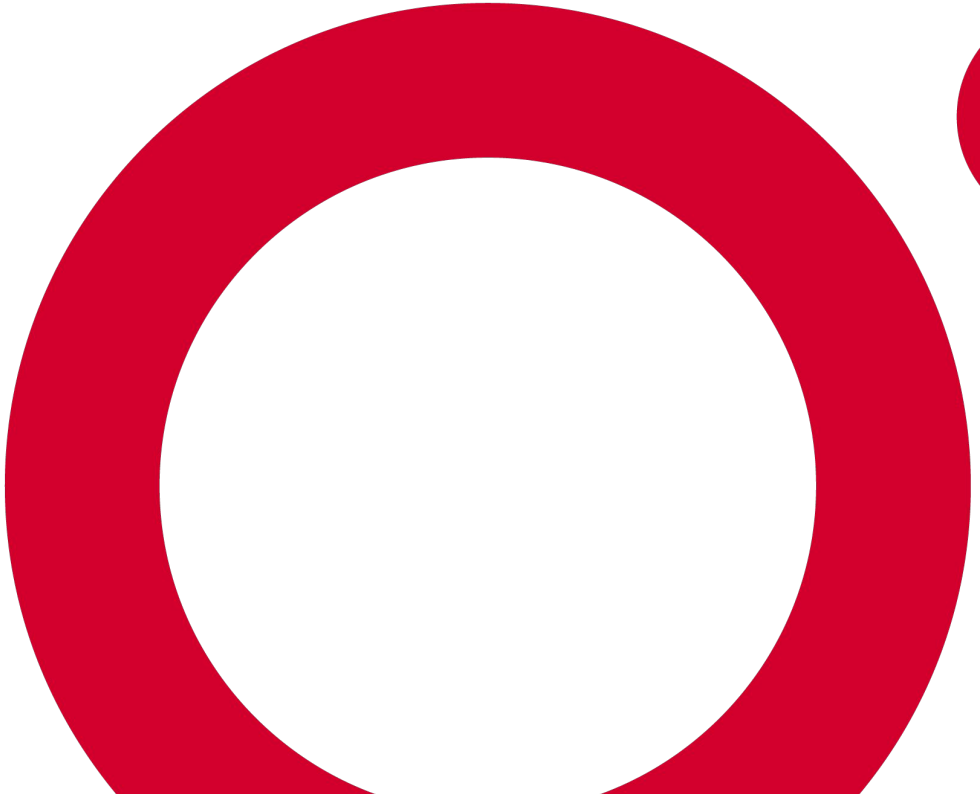
O'REILLY®

ReactJS First Steps

Creating web apps quickly and elegantly



September/2020

Rap - pulse check

- Can you hear me okay? Who is here?

Introduction to React
How to create a React app
Creating a React component
Props
Styling
Events
State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

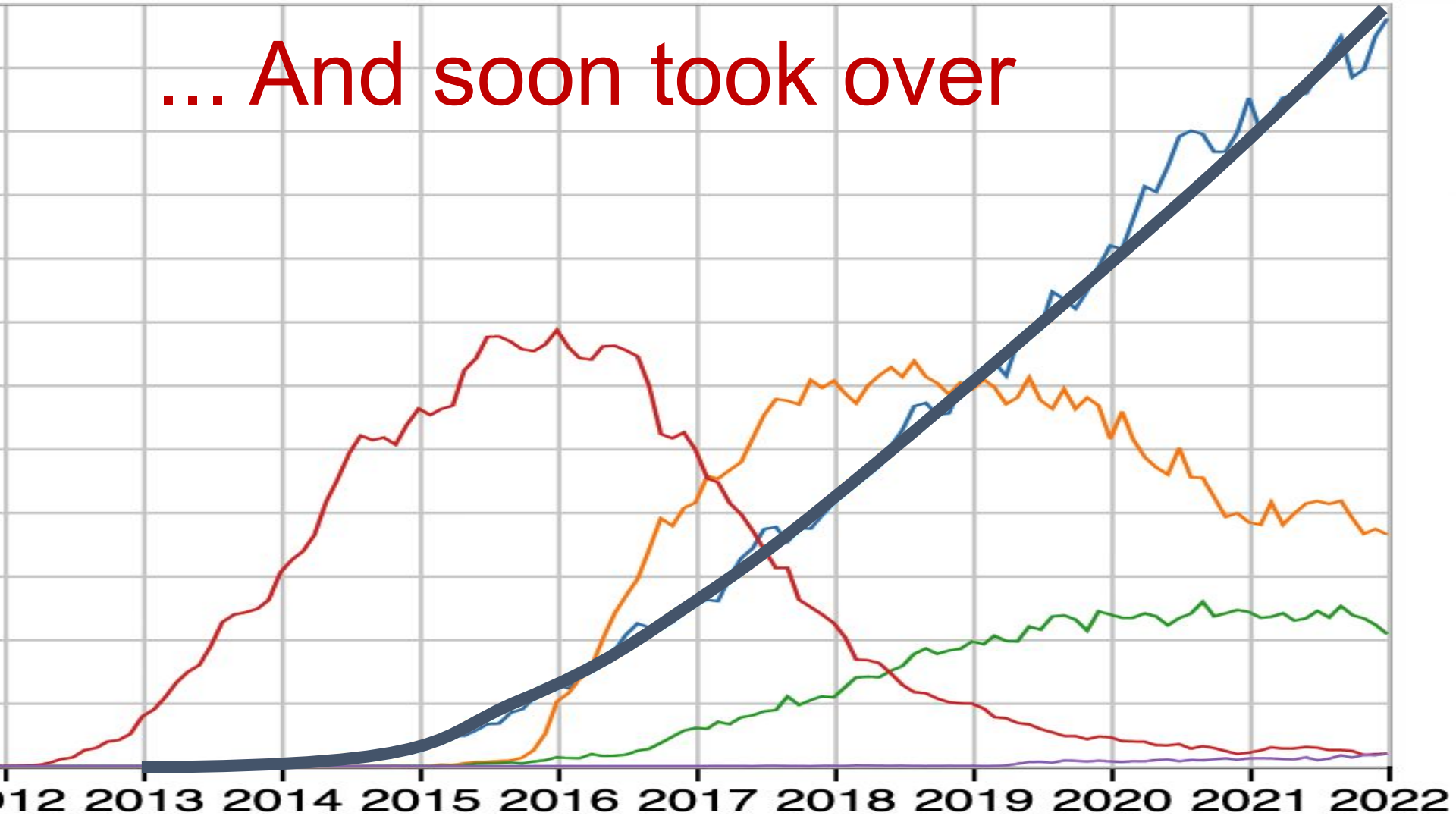
State

tl;dr

- What is React? (A JavaScript library)
- Why React? (to maintain web apps easier)
- The secrets of React
  - How it's so darned fast (The virtual DOM)
  - What is really happening when an app runs (It's a SPA)


REACT IS A
JAVASCRIPT LIBRARY
THAT ALLOWS US TO
CREATE AND THEN
MODIFY DYNAMIC
WEB APPLICATIONS
FASTER AND EASIER

Written in 2011 by
Jordan Walke of
Facebook's Ads team


... And soon took over
012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022

The web has gone to components

Nearly 100% of modern web app development is done using components

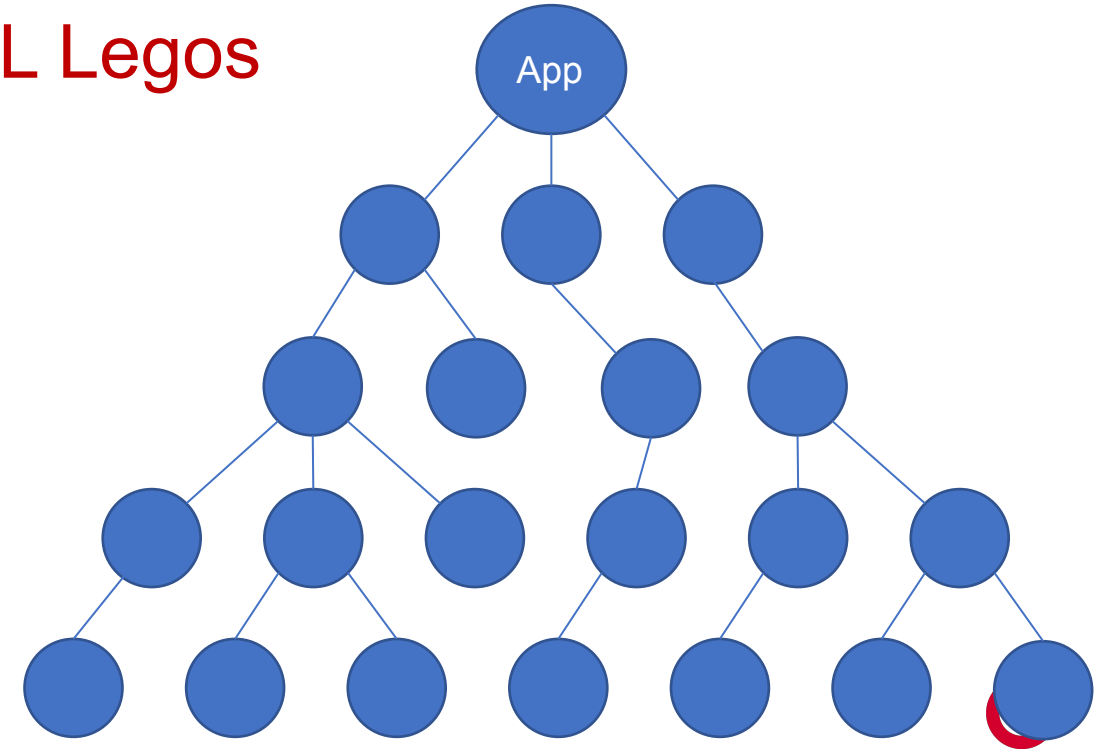
With React, you'll no longer write pages; you'll write components

Each component is self-contained and encapsulated

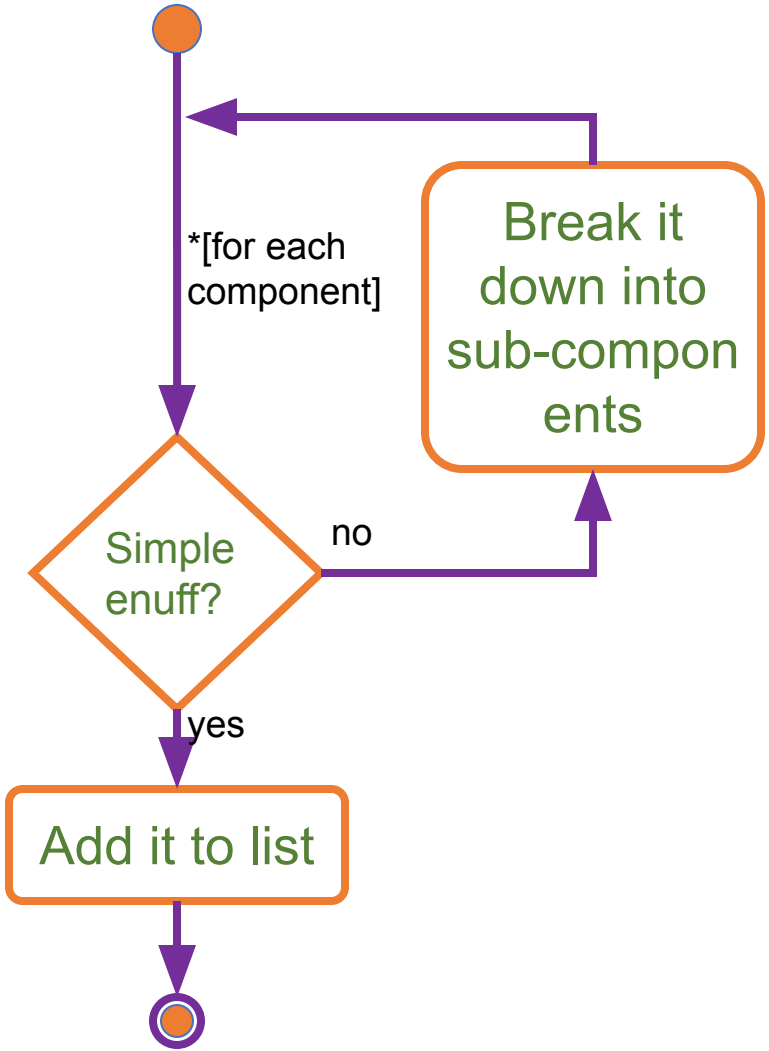
- Even styles are local; CSS no longer cascades through components

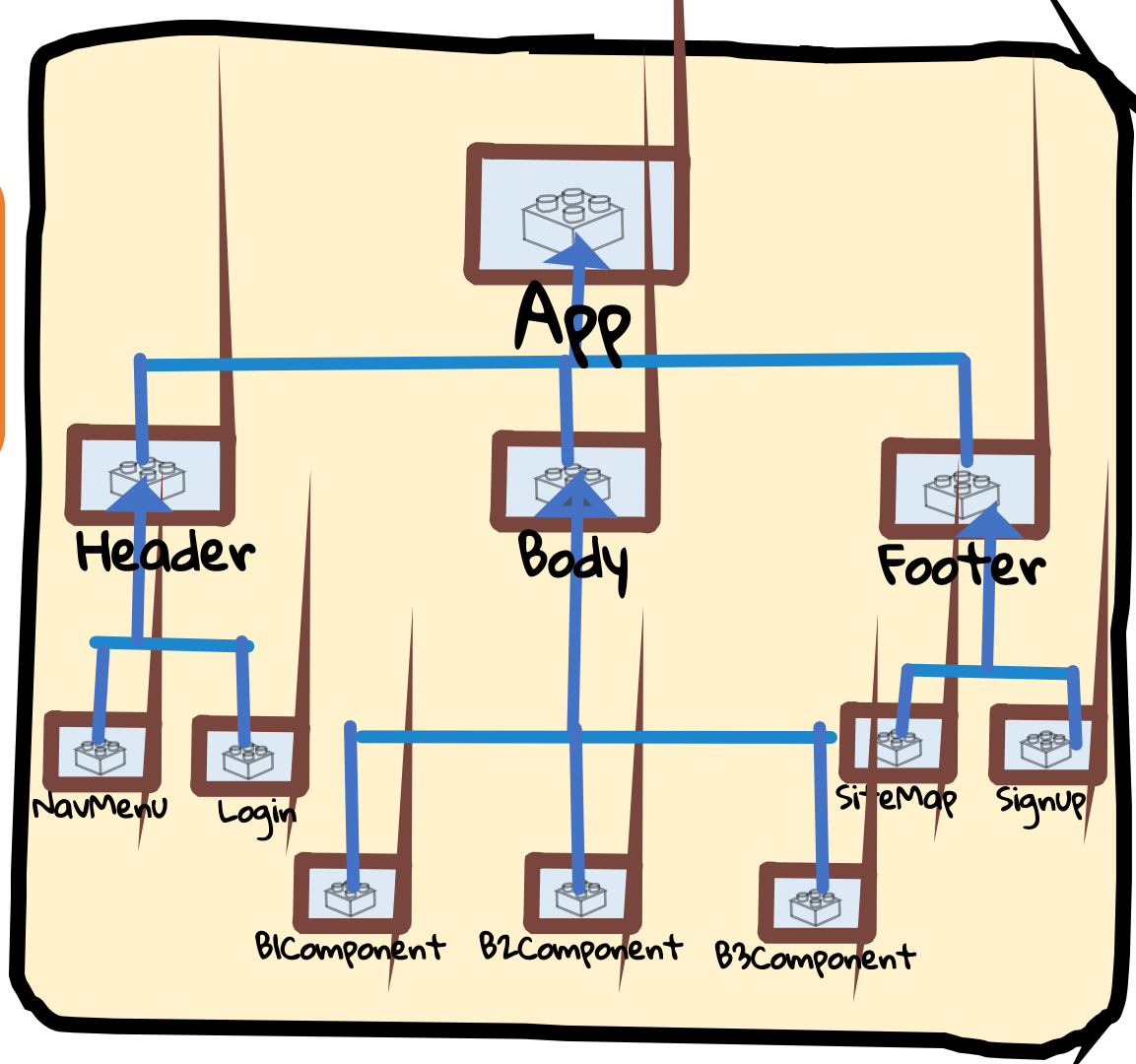
We put components together to create an app

Like working with HTML Legos

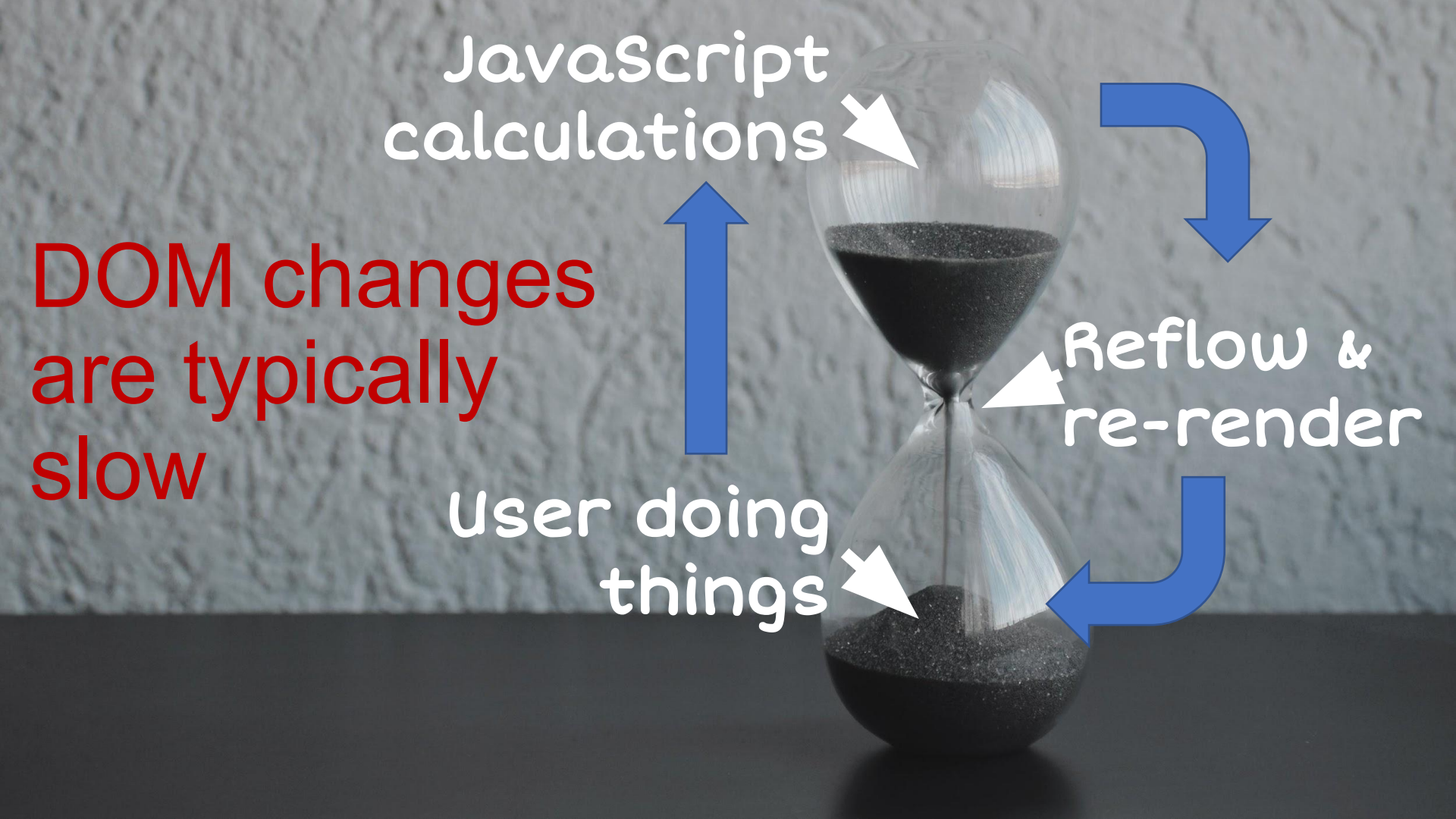



YOU'LL BREAK DOWN THE PRODUCT INTO SECTIONS. THEN YOU BREAK THOSE INTO SMALLER SECTIONS AND SO ON ...
PHOTO: LEGO


*[for each component]
Break it down into sub-components
Simple enuff?
no
yes
Add it to list


App
Header
Body
Footer
NavMenu
Login
SiteMap
SignUp
B1Component
B2Component
B3Component

How is React so darned fast?


JavaScript calculations
Reflow & re-render
User doing things
DOM changes are typically slow

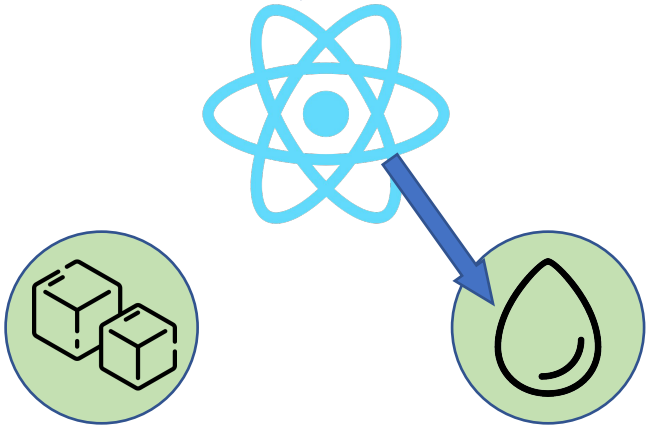
Hey, why don't we create our own little version of the DOM tree?

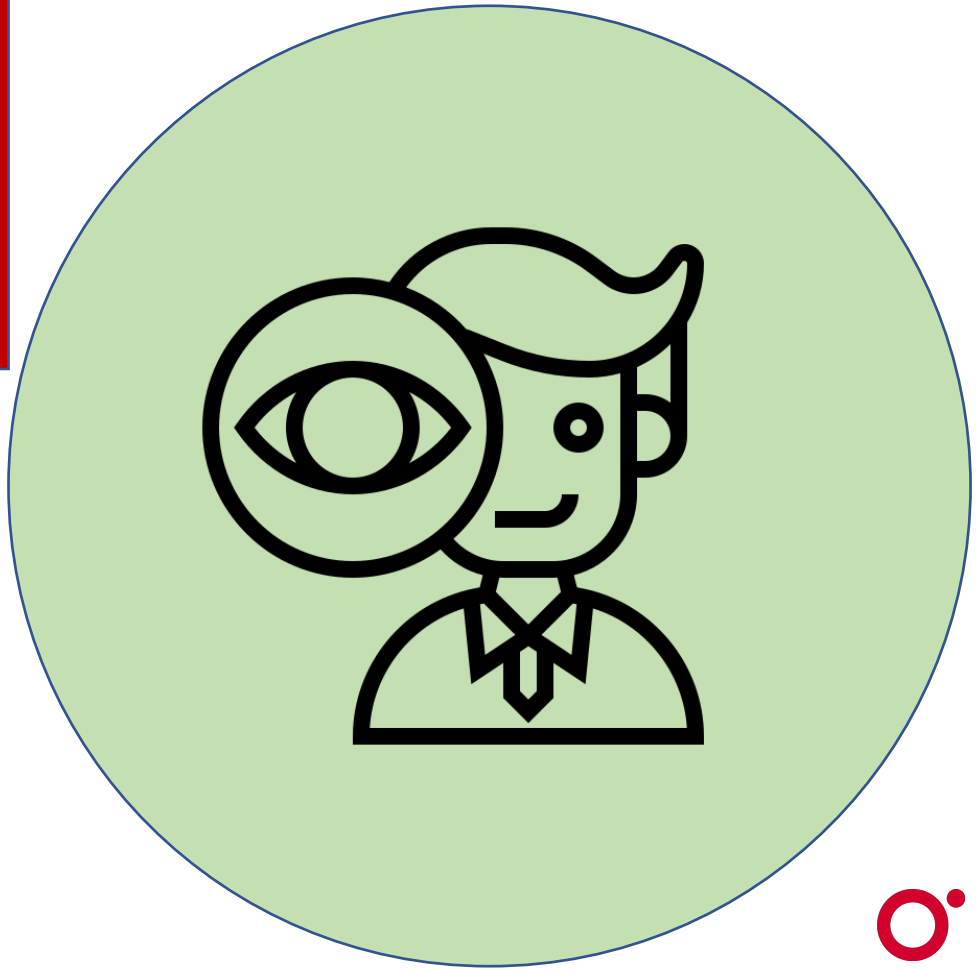
One that has just enough stuff for us to manipulate?

Then we can compare much more quickly and help the browser out with DOM changes!

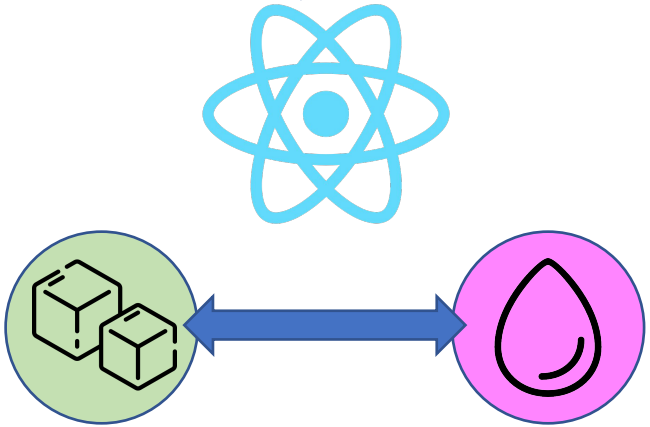
Brilliant idea!

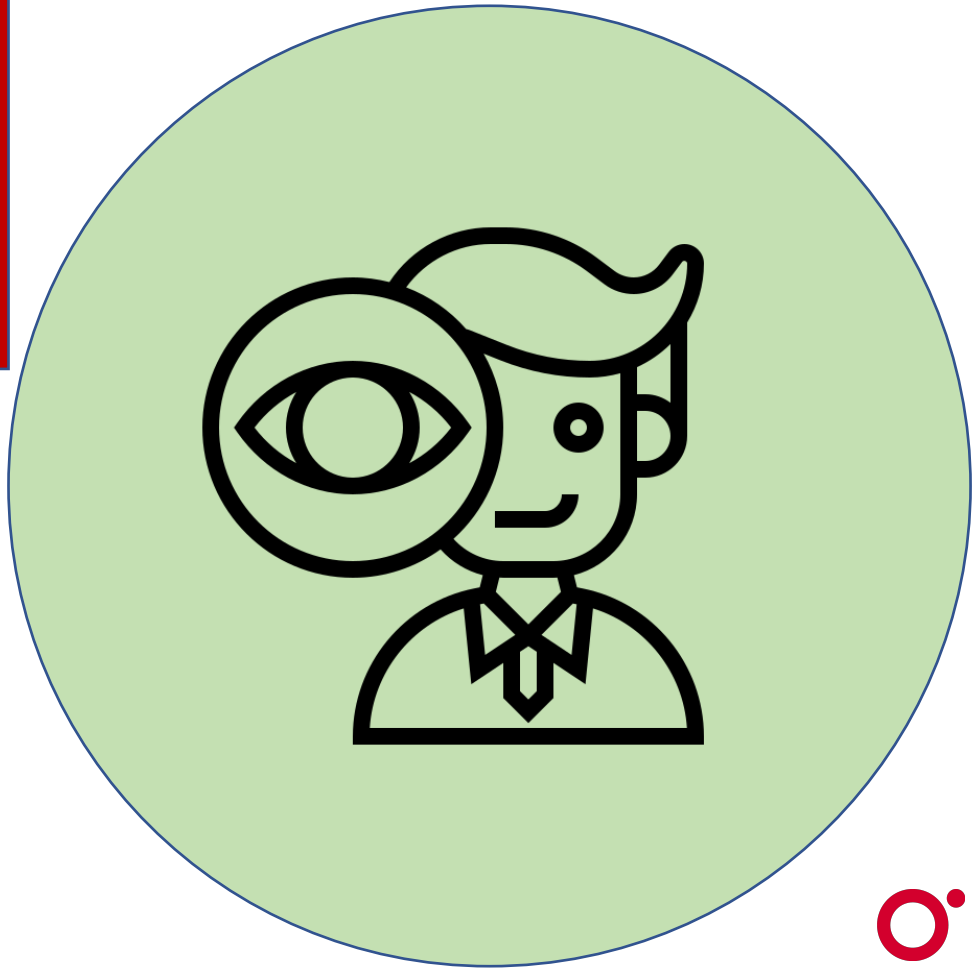
Ooh! User wants me to make these 9,452 changes to the DOM.



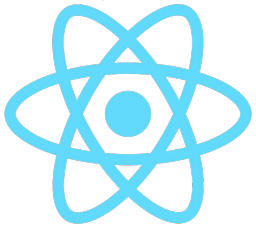


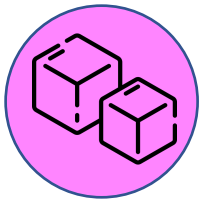
Done. Now,what's the difference between these two?

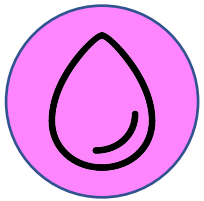


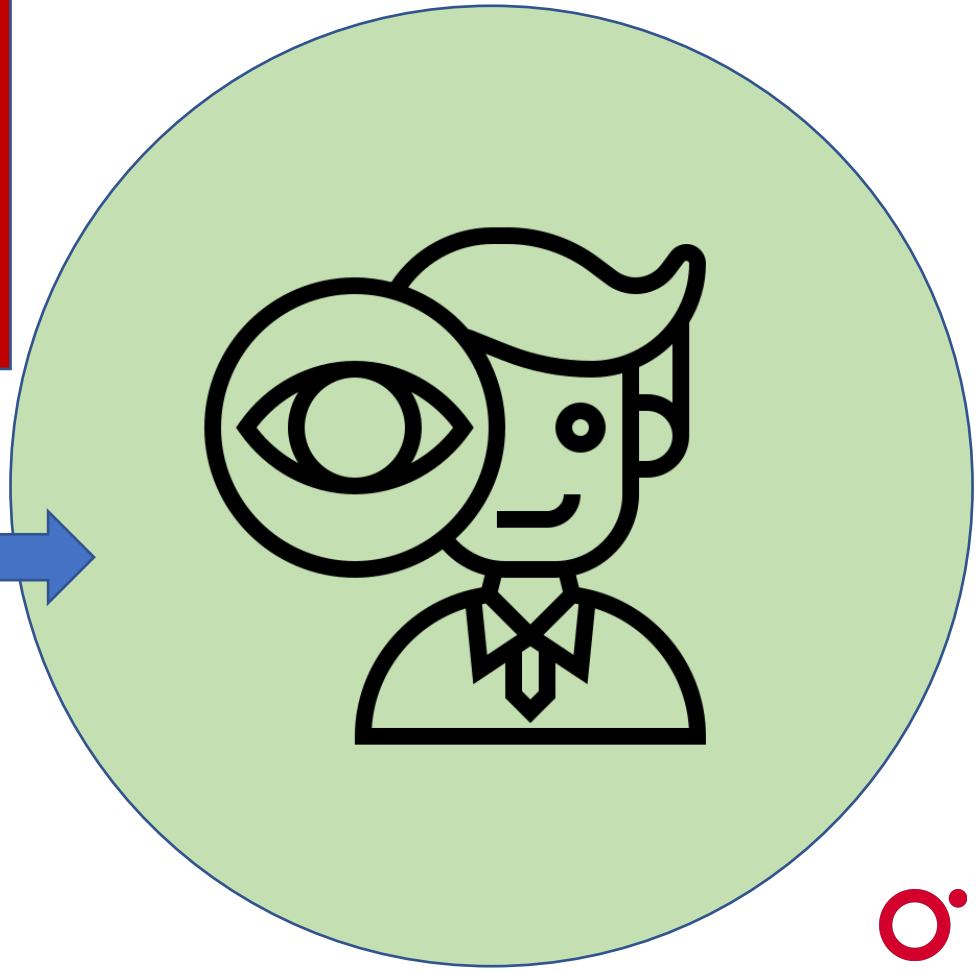


For every diff, I'll apply that to the real DOM ... but only once!









virtual DOM

- Internal algorithms compare the before and after pictures and determines the smallest amount of changes that can be made. And batches them. And then makes the changes in the DOM.
- Sound like a lot of overhead? Yep.
- But it ends up being so much faster!

- Because it diffs, it only makes changes to the part(s) of the page that have changed.
- Thus it only changes when internal state changes (Spoiler: React tracks 2 kinds of data:
  - Data that is sent into a component and does NOT change
  - Data that can change.
- Hint: Keep your components simple ... don't ever calculate diffs. Just redraw the whole component and let React use the virtual DOM to decide what should redraw and what won't need to.

The algorithm

- React maintains an old VDOM and creates a new VDOM based on the new state/props.
- It diffs them, isolating the part(s) that need rerender
- It creates a read DOM for those parts needing rerender
- It injects those into the DOM.

The diffing algorithm

- It starts at the root of A and B.
- For each child ...
- compare each child from A to B
- Different types? B needs rerendering
- Same types? Look at the node properties. What has changed?
- An HTML property has changed? Just change that one property. Don't rebuild.
- Lists and keys ...

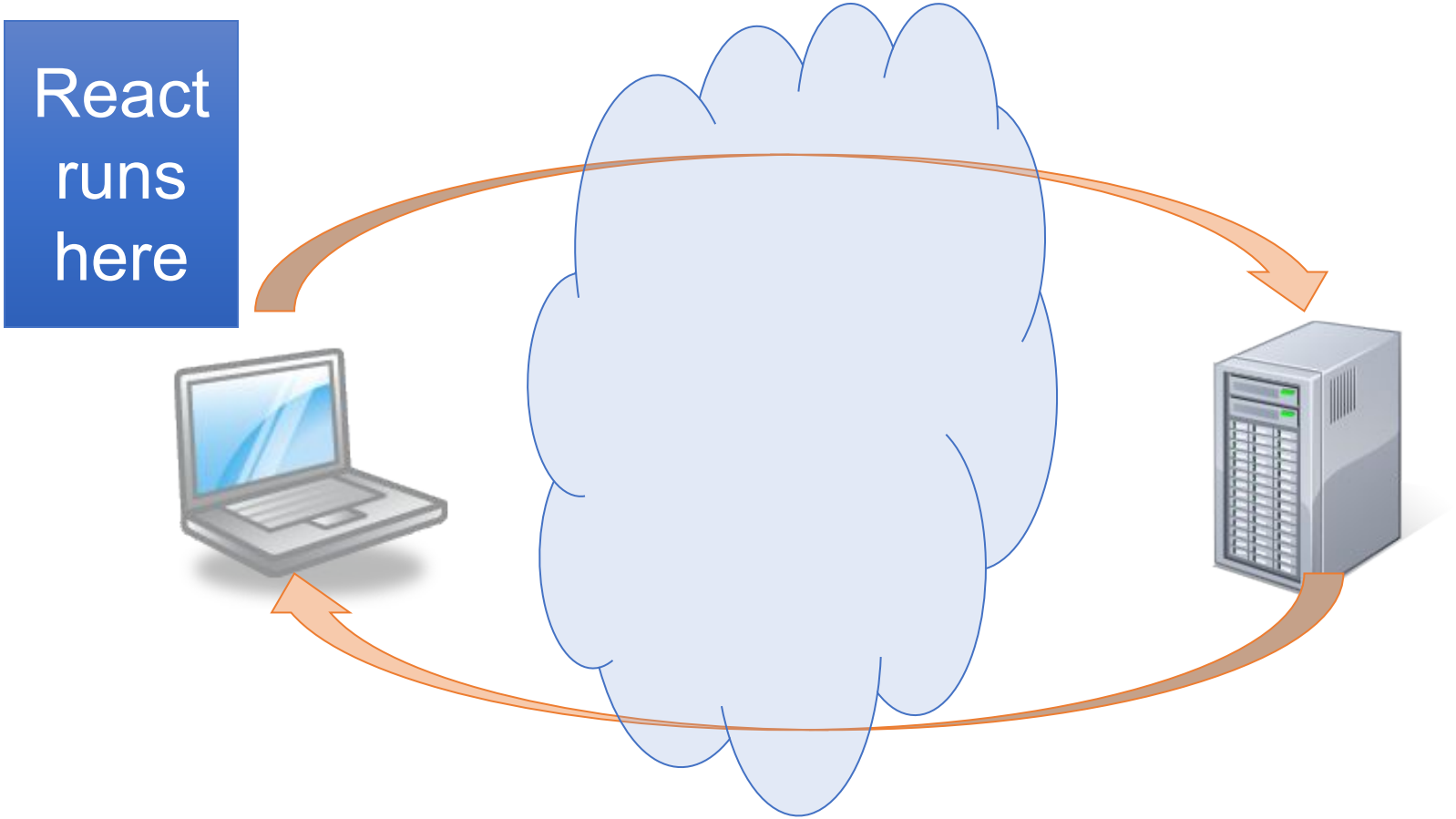
Rerendering a node

- Unmount it
- Run componentWillUnmount
- Destroy it.
- Then what?

React Fiber

- Since JS is single-threaded, you can't interrupt, cancel, resume calls. But React needs this to prioritize and allow, say, a button click to interrupt an animation.
- React Fiber is a low-level reimplementation of the JS Call Stack.
- You can think of React Fiber as the Virtual Call Stack.
- React uses fibers. These are a single unit of work or a virtual stack frame.
    - An animation Frame, eg.

The web uses a thin-client architecture



How React works at runtime

1. User requests *any* url from our site.
2. Server responds with index.html
3. client requests bundle.js which is provided and contains all html, css, JavaScript files, libraries
4. Client runs

```
ReactDOM.render(<App />, someNode)
```

5. This loads your main React component on the page.
6. It loads all subcomponents and so forth and so on.


You mean I write all of this in one big bundle.js file?
Nope! Bundle.js is created from all of your JS, HTML and CSS


Sounds super complex! Is it?
Yup!
But we have npm to automate parts -- if we set it up right...

React demands that you have a very precise, very complex setup with literally thousands of interdependent libraries having dozens of version numbers each.

If one small part of it is not configured properly, the system fails.

Wow! all these parts! Wouldn't it be nice to have a way to automate the creation and scaffolding of the different parts?

Next section!

tl;dr

- What is React? (A JavaScript library)
- Why React? (to maintain web apps easier)
- The secrets of React
  - How it's so darned fast (The virtual DOM)
  - What is really happening when an app runs (It's a SPA)

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

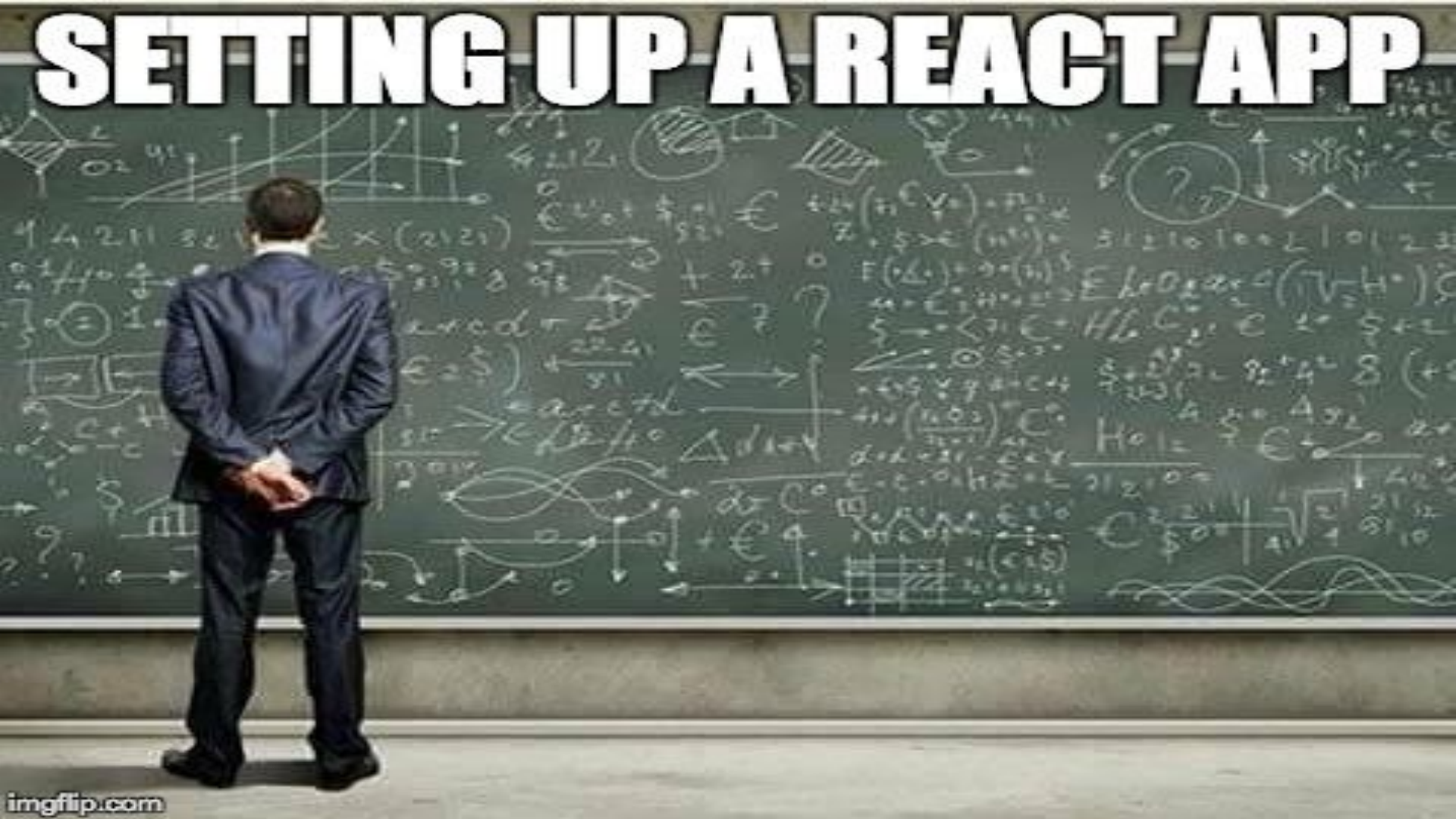
tl;dr

- React is very difficult to configure and brittle.
- create-react-app simplifies creation
- Compile and run in watch mode with `npm start`

React has a lot of moving parts

React uses JSX, which isn't JavaScript ...
So it has to be transpiled by Babel ...
which should be automated by webpack ...
which also minifies, bundles and tree-shakes ...
the CSS, HTML, and hundreds of libraries ...
which must be managed by npm ...
thus carefully configured in package.json ...
linted with eslint and tested with jest and RTL ...
using ES2015 modules and AMD requires ...
run through node processes ...


webpack
package.json
karma
promises
redux
npm
@
testing-library
jest
JSX
react
map()
node
eslint


SETTING UP A REACT APP
imgflip.com

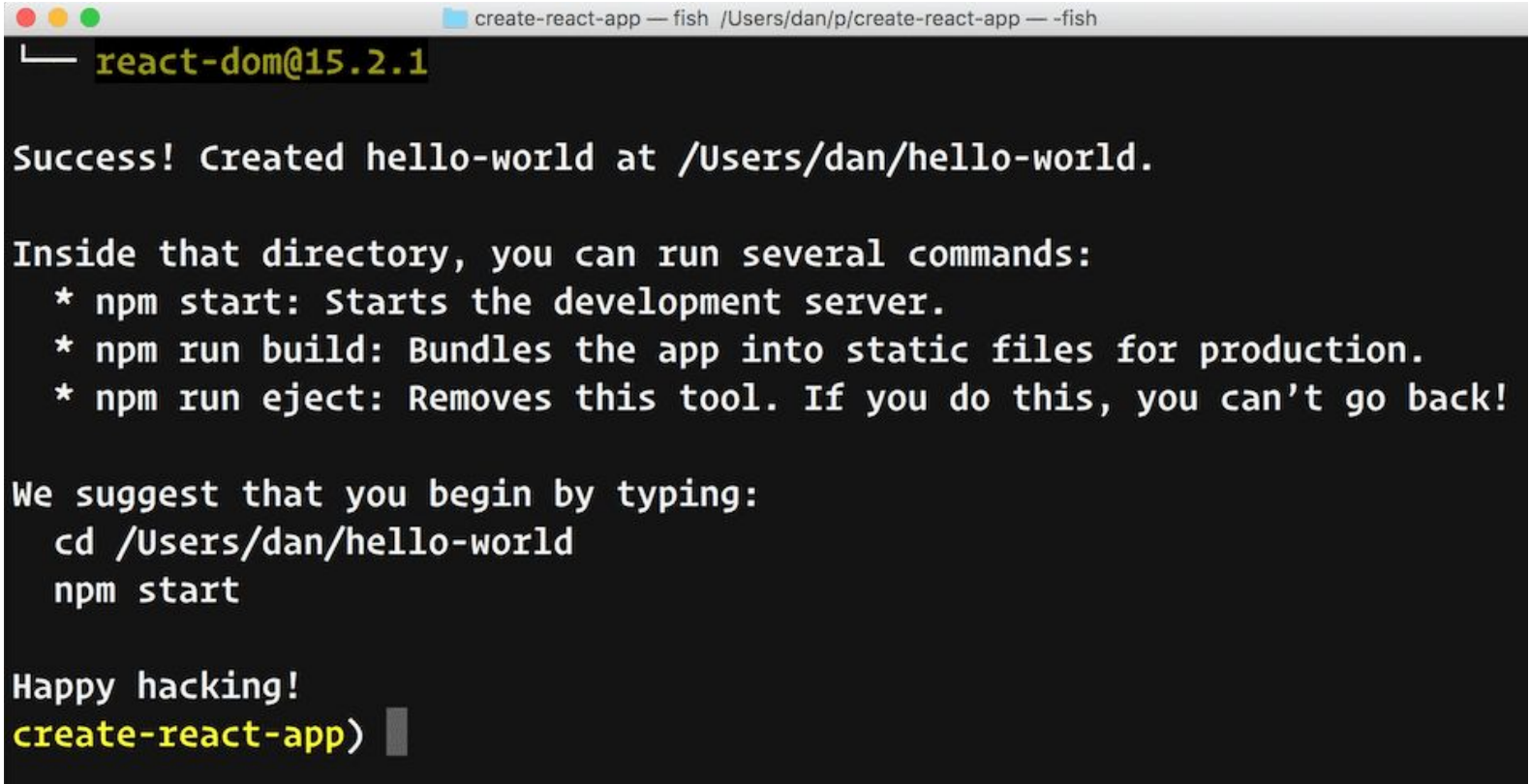
We knew it was frustrating to spend days setting up a project when all you wanted was to learn React. We wanted to fix this.

We loved Ember CLI and Elm Reactor which help users to get started.

So Christopher Chedeau, Kevin Lacker, and I wrote create-react-app

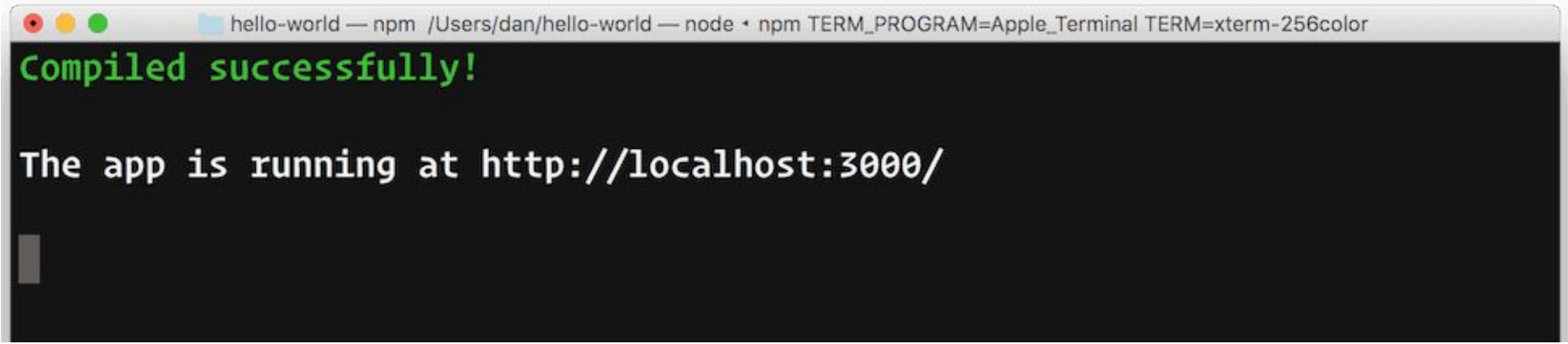
create-react-app makes your project

```
npx create-react-app hello-world
```



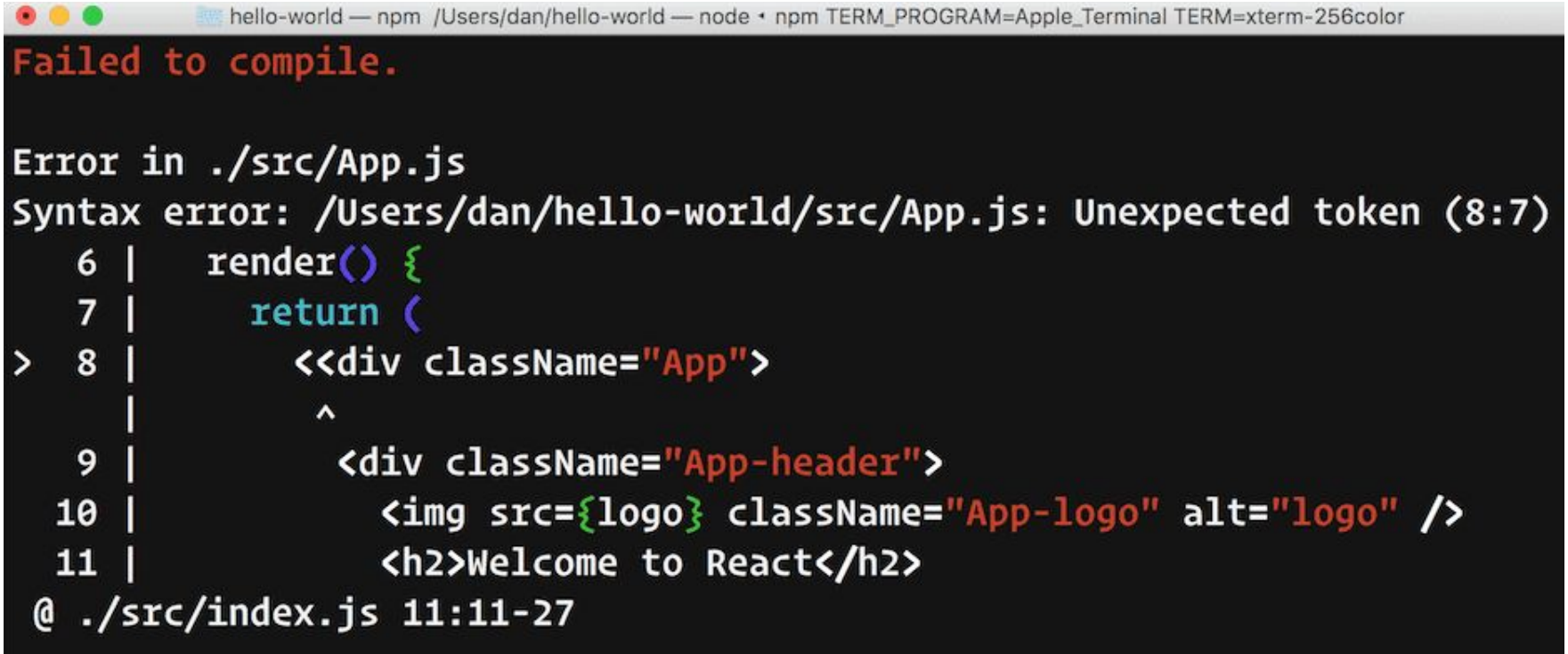
Starting the Server

- Run `npm start` to launch the development server. The browser will open automatically with the created app's URL.



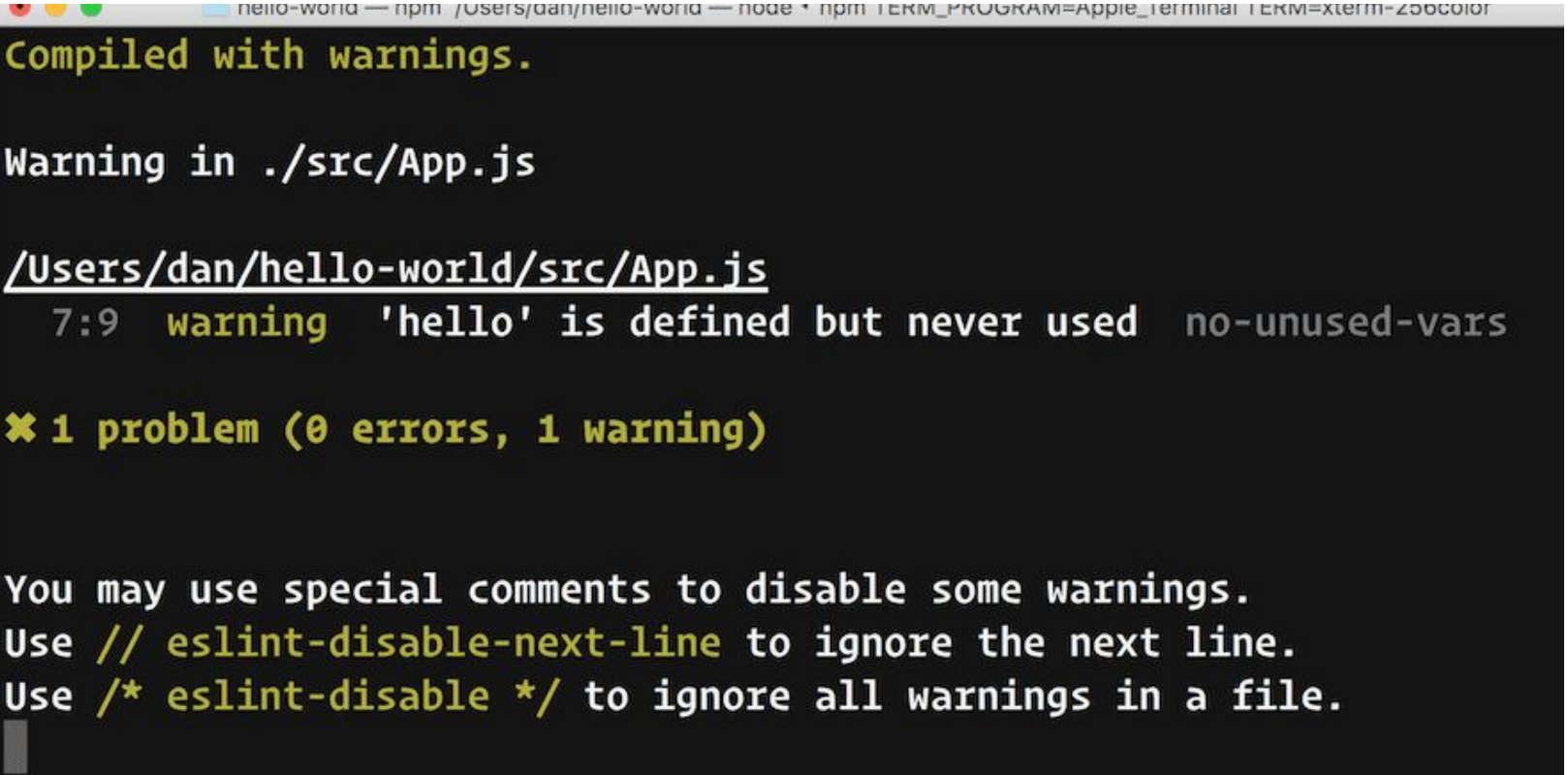
Then you're in watch mode

- When your code changes, webpack lints with eslint and transpiles with babel.



ESLint output shows in the console

- The lint rules are tuned for a React app.



tl;dr

- React is very difficult to configure and brittle.
- create-react-app simplifies creation
- Compile and run in watch mode with `npm start`

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

tl;dr

- The steps to make a React component
- Data binding is done with JSX expressions
- Expressions can handle
  - Conditional rendering
  - Iterating arrays
  - Function calls for complex processing

How to make a React component

1. Create a component file
2. Create a function
3. Return JSX from it

1. Create a component file

- ... in the src folder

Foo.js

```
export function Foo() {
 return <SomeComponent />;
}
```

2. Create a function ...

- Must start with an uppercase letter! Prefer Pascal-cased

Foo.js

```
export function Foo() {
 return <SomeComponent />;
}
```

3. Return JSX from it

Foo.js

```
export function Foo() {
  return <SomeComponent />;
}
```

3. Return JSX from it

When you want a return and then have JSX starting on the next line, wrap them in parentheses or else Babel gets confused.

Foo.js

```
export function Foo() {
 return (
  <SomeComponent />
 );
}
```

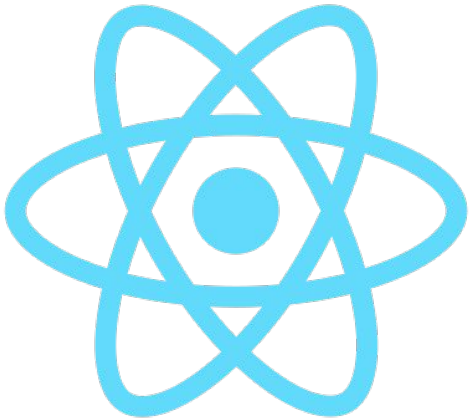
Actually, it returns ...

1. JSX with a single root element
2. An array of JSX elements
3. A string
4. null

- Anything else is an error

Wondering about JSX?
Don't worry, we'll cover it in depth next chapter.

To help with the presentation, React introduces a new DSL* called JSX



* Domain-Specific Language


JavaScript + XML = JSX
It isn't JSH. It is JSX.
HTML is forgiving ... XML is not!

Well-formed XML (and therefore, JSX) Rules

- Elements must be closed.
- Tags are case-sensitive.
- Elements must be properly nested.
- Must have a root element.
- Attribute values must always be quoted.

Data binding

How to display data in a component

Person.js

```
export function Person() {
 const person = getPerson(1234);
 return (
  <section>
   <img src=person.imageUrl />
   <p>person.first person.last</p>
  </section>
 )
}
```

How do you display the person's data?

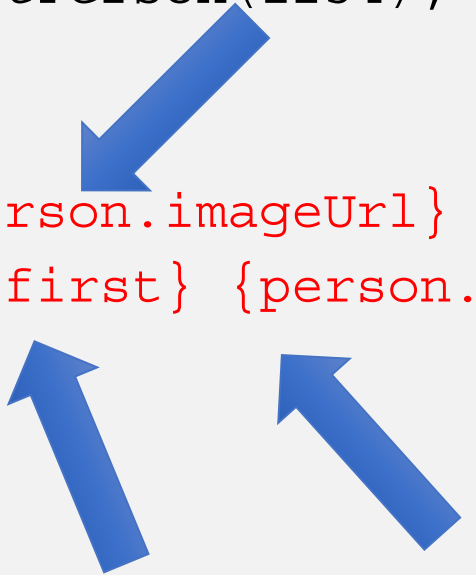
We often want to display live data!

Hey! We could run some JavaScript inside the JSX! Then we could use conditionals, loops, and call functions!

Expressions are merely JavaScript inside of JSX

Person.js

```
export function Person() {
  const person= getPerson(1234);
  return (
    <section>
      <img src={person.imageUrl} />
          {person.first} {person.last}
    </section>
  )
}
```



You can add JavaScript to JSX if you put it in curly braces

Expressions can contain JSX

ListPeople.js

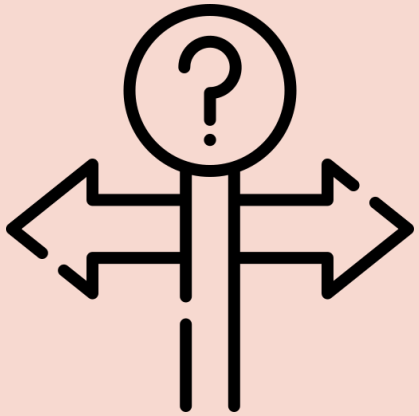
```
const people = getPeople();
export const ListPeople = () => (
 <>
   { people.length ? <People /> :
          <p>No people available</p> }
 </>
)
```

And that JSX can have an expression,

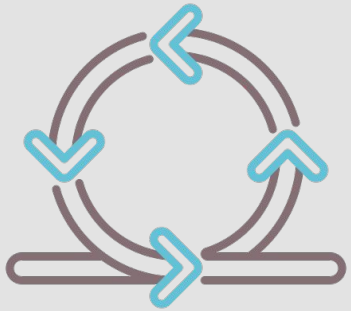
which can have more JSX,
which can have an expression,
which can have more JSX,
which can have an expression,
which can have more JSX,
which can have an expression,
which can have more JSX,
which can have an expression,
which can have more JSX,
which can have an expression,
which can have more JSX,
which can have an expression,
... ad nauseum

Expressions are also super useful for ...

Conditional Rendering



Looping



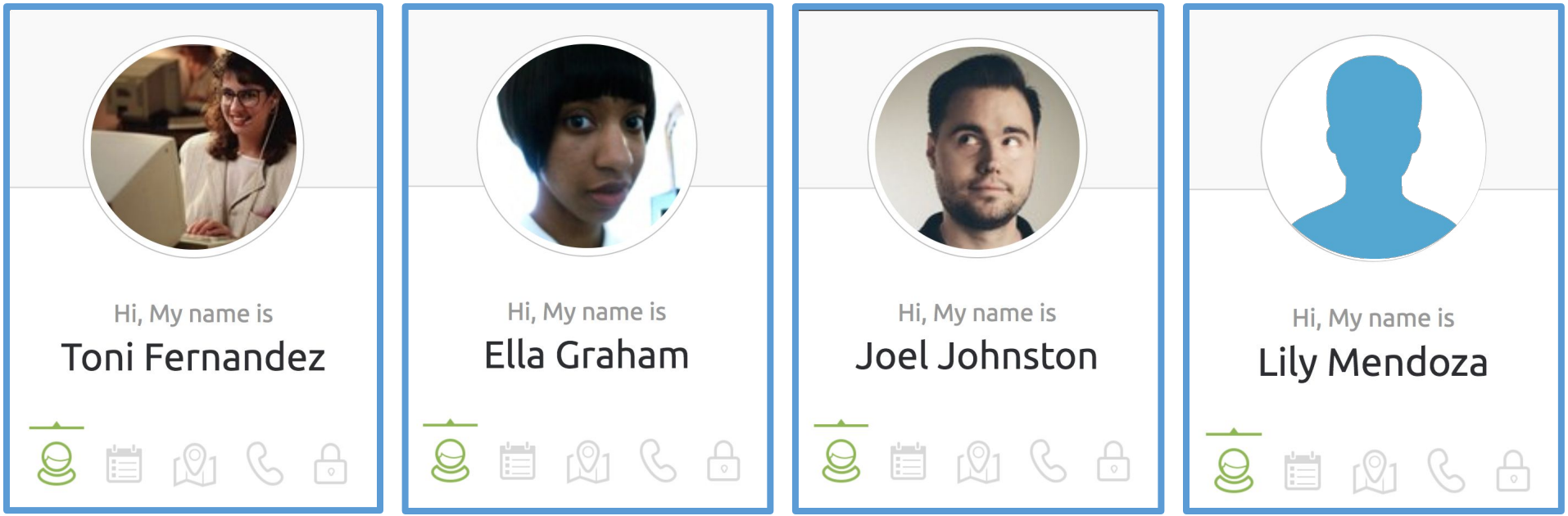
Say we're reading a list of people ...

Conditional Rendering

people.json

```
[{
"name": { "first": "maëlia", "last": "dupuis" },
"email": "maëlia.dupuis@example.com",
"cell": "06-76-31-32-56",
"picture": { "large": "md65.jpg"}
},
{
"name": { "first": "susanne", "last": "scott" },
"email": "susanne.scott@example.com",
"cell": "081-007-7340",
},
{
"name": { "first": "babür", "last": "çörekçi" },
"email": "babür.çörekçi@example.com",
"cell": "(743)-870-9450",
"picture": { "large": "bc65.jpg" }
}]
```

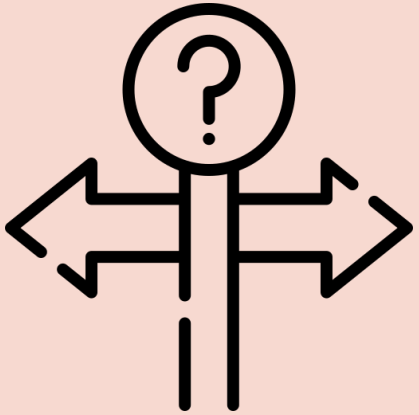
And ListPeople.js has a photo



If the user has a photo, show it. If not, show a generic placeholder image.

An expression evaluates to a single thing which is then substituted back into the JSX

Conditional Rendering



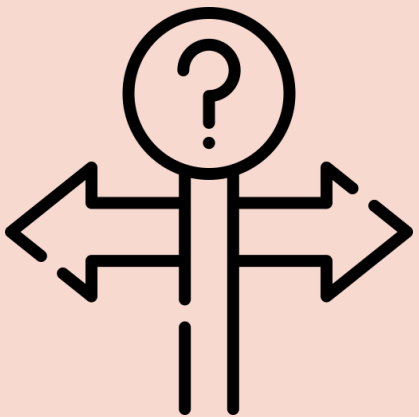
Expressions must be a single JavaScript expression

- ... not a statement
- ... not a block
- ... not an assignment
- ... not a line of code

Generally, expressions are something you'd find on the right side of an "="

We can't do this

Conditional Rendering

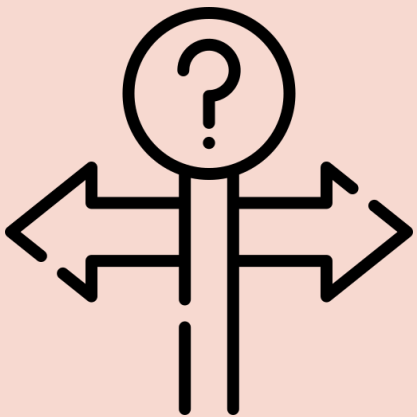


ListPeople.js

```
export function ListPeople() {
 const p = getPerson(1234);
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   <img src={if (p.img) p.img else ph} />
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

But a ternary will work

Conditional Rendering



ListPeople.js

```
export function ListPeople() {
 const p = getPerson(1234);
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   <img src={p.img ? p.img : ph} />
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

Or short-circuiting will work

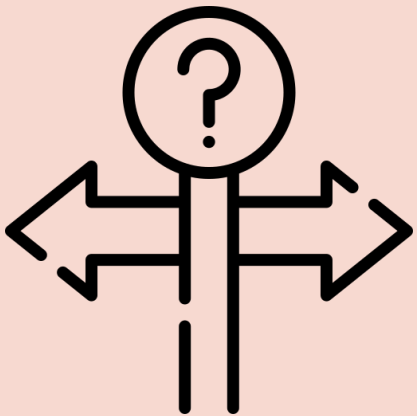
Conditional Rendering

ListPeople.js

```
export function ListPeople() {
 const p = getPerson(1234);
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   <img src={p.img || ph} />
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

Or short-circuiting will work

Conditional Rendering



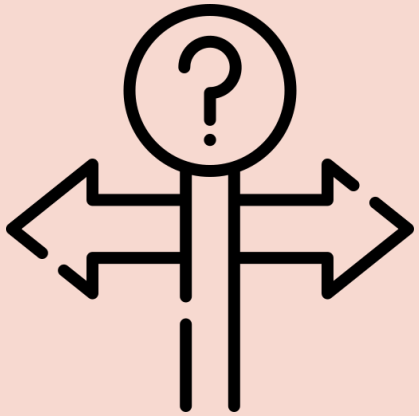
ListPeople.js

```
export function ListPeople() {
 const p = getPerson(1234);
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   {p.img && <img src={p.img} />}
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

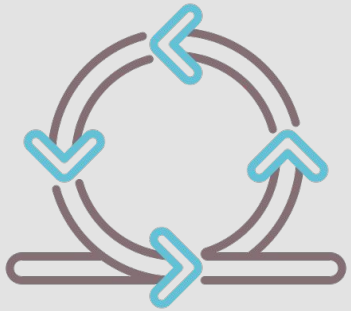
If we don't have an image, just don't put anything in the JSX

Expressions are also super useful for ...

Conditional Rendering

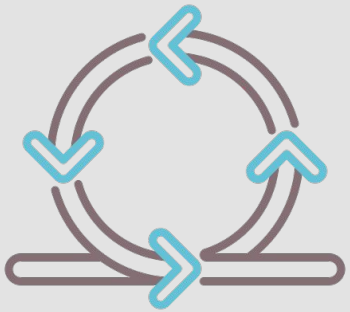


Looping



An expression evaluates to a single thing which is then substituted back into the JSX

Looping



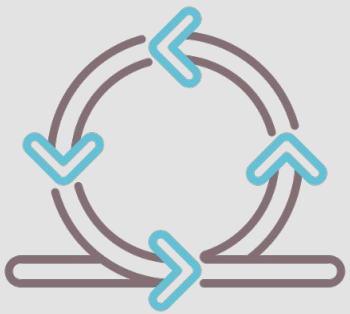
Expressions must be a single JavaScript expression

- ... not a statement
- ... not a block
- ... not an assignment
- ... not a line of code

Generally, expressions are something you'd find on the right side of an "="

We can't do this

Looping

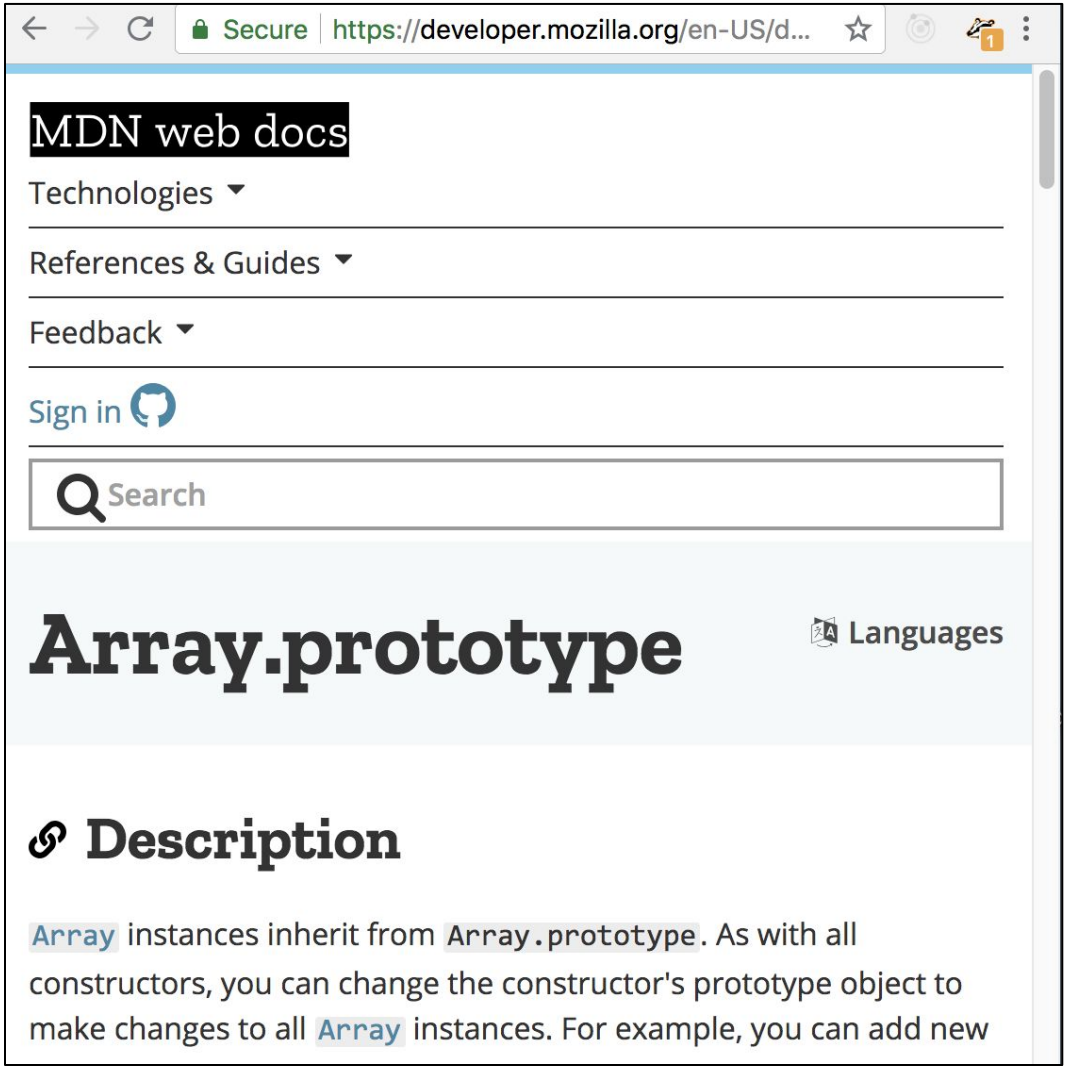


ListPeople.js

```
export function ListPeople() {
 const people = getPeople();
 return (
  <section>
   {for (let p of people)
    <Person person={p} />
   }
  </section>
 )
}
```

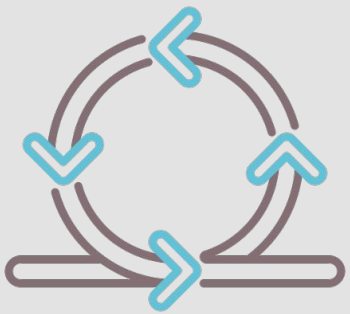
But there are a bunch of JavaScript Array.prototype.* methods that will iterate an array and operate on each thing

- concat()
- filter()
- flat()
- flatMap()
- join()
- slice()



.map() will work great!

Looping



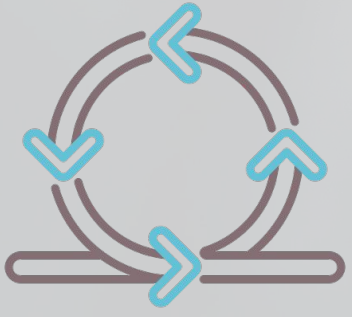
ListPeople.js

```
export function ListPeople() {
 const people = getPeople();
 return (
  <section>
   {people.map((p) => {
    return <Person person={p} />
   })}

   {/* Or more concisely... */}
   {people.map(p=><Person person={p} />)}
  </section>
 )
}
```

You should provide a stable key to each iterated JSX element

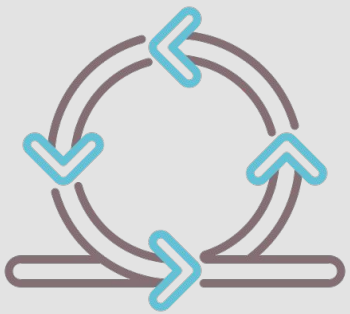
Looping



React uses this for the virtual DOM for performance. It matches the VD element to its DOM element by that ID.

Anything unique can serve as a key

Looping

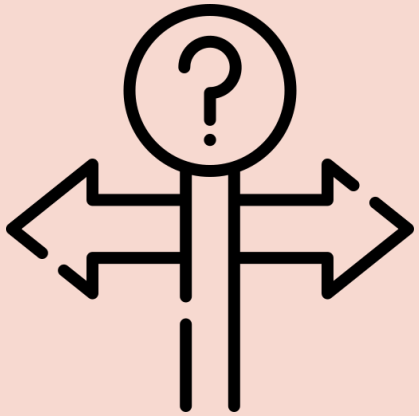


ListPeople.js

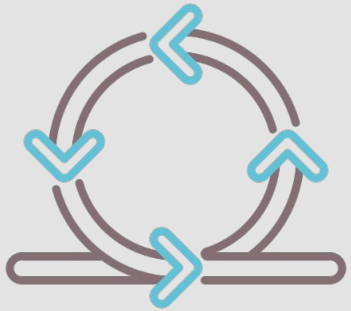
```
export function ListPeople() {
 const people = getPeople();
 return (
  <section>
   {people.map(p => (
      <Person key={p.id} person={p} />
   ))}
  </section>
 )
}
```

Expressions are also super useful for ...

Conditional Rendering

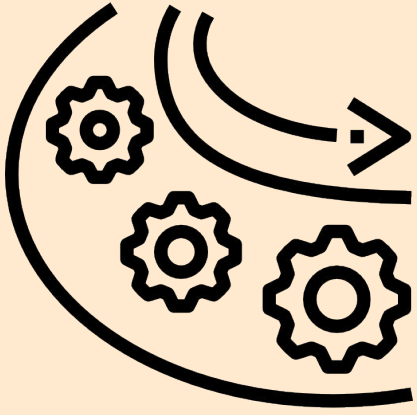


Looping



A function call is a single expression

Calling functions

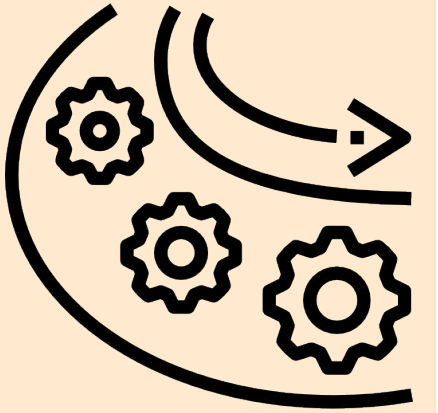


- If the logic you want is too complex for a single expression, you can call a function
- Functions can be as complex as you like!

If it's too complex to be a single expression, but a whole function seems like overkill, remember that an iife is a single expression!

The function must return something that can be displayed in JSX which is ...

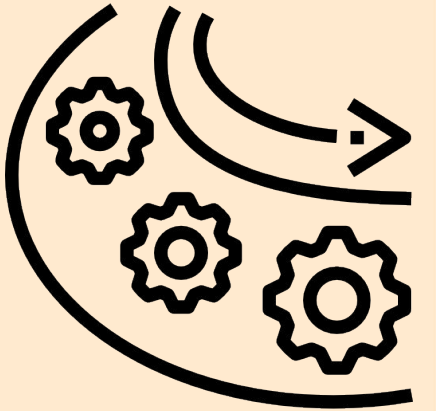
Calling functions



1. More JSX with a single root node
2. An array .... of JSX elements, each having a single root node
3. A string (yawn)
4. null

The function must return JSX

Calling functions



ListPeople.js

```
export function ListPeople() {
 const people = getPeople();
 return <section>
   {specialPeople(people)}
  </section>
}
function specialPeople(people) {
 const ppl = [];
 for (let p of people) {
  p.birthdate === today && ppl.push(p);
  p.city.startsWith("New") &&
ppl.push(p);
 }
 return ppl.map(p => <Person pers={p} />)
}
```

tl;dr

- The four steps to make a React component
- Data binding is done with JSX expressions
- Expressions can handle
  - Conditional rendering
  - Iterating arrays
  - Function calls for complex processing

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

tl;dr

- To pass data down, write the values as attributes in the host and read them in *props* in the inner where they're immutable
- Use props drilling to send down multiple levels

props are essentially the input parameters of a component.

How to pass data from host to inner

CompanyDirectory.js

```
function CompanyDirectory() {
 const [p1,p2] = get2Users();
 return <div>
  <Person hairColor="red" eyeColor="blue"
          first={p1.first} last={p1.last}
          imgSrc={p1.imgSrc} />
  <Person hairColor="brunette"
          eyeColor="brown" imgSrc="noImage.jpg"
          first={p2.first} last={p2.last} />
  </div>;
}
```

To read data in inner from host

Person.js

```
export function Person(props) {
 const { first, last, hairColor, eyeColor } = props;

 return <section>
  <p>{first} has {hairColor} hair and {eyeColor} eyes.</p>
  <img src={props.imgSrc} alt={first + " " + last} />
  </section>
}
```

You always read the data with *props*.

Props are immutable(kind of)

- To add something to the props object is an error

```
TypeError: Cannot add property foo, object is
not extensible
```

- To reassign the value of a prop is an error. They're read-only.

```
TypeError: Cannot assign to read only property
'first' of object '#<Object>'
```

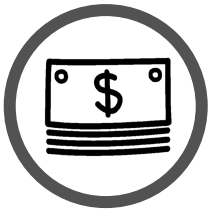
Full disclosure: if the prop is an object, the properties of that object can be changed, just not the prop itself

But what if I have data that needs to change? Like based on user input or Ajax calls or something?

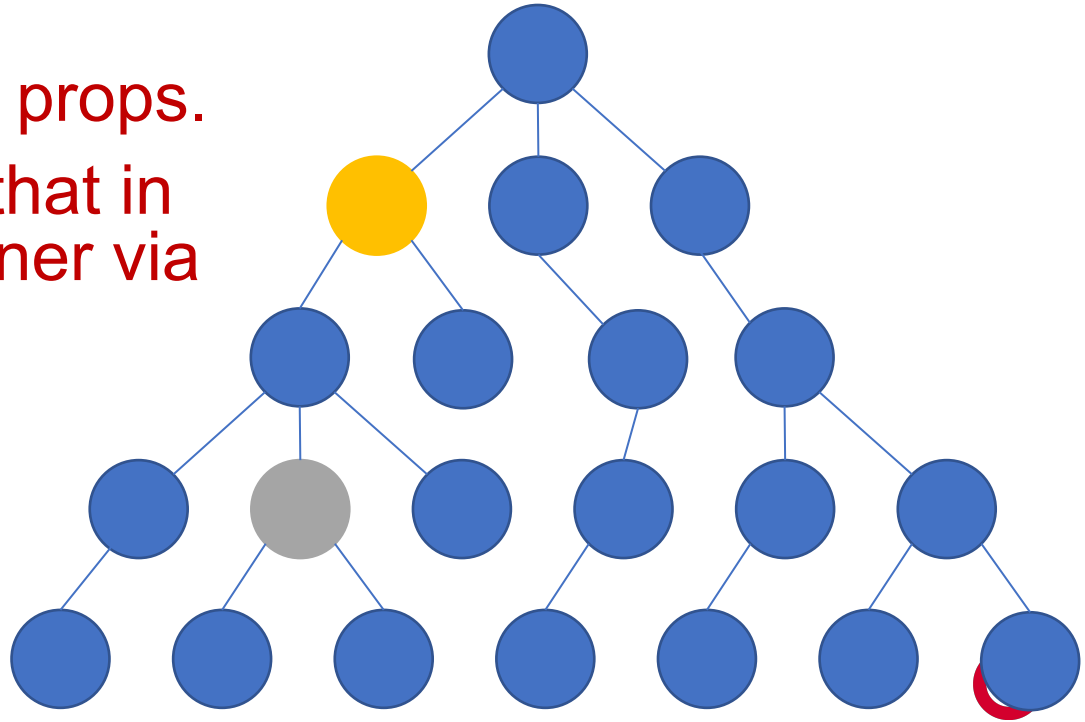
- Well, that is not props. It's state
- It isn't uncommon at all to read props and copy them into state.

What if I need to send data down two levels?

1. From host to inner via props.
2. Then the inner reads that in and sends it to *its* inner via props.

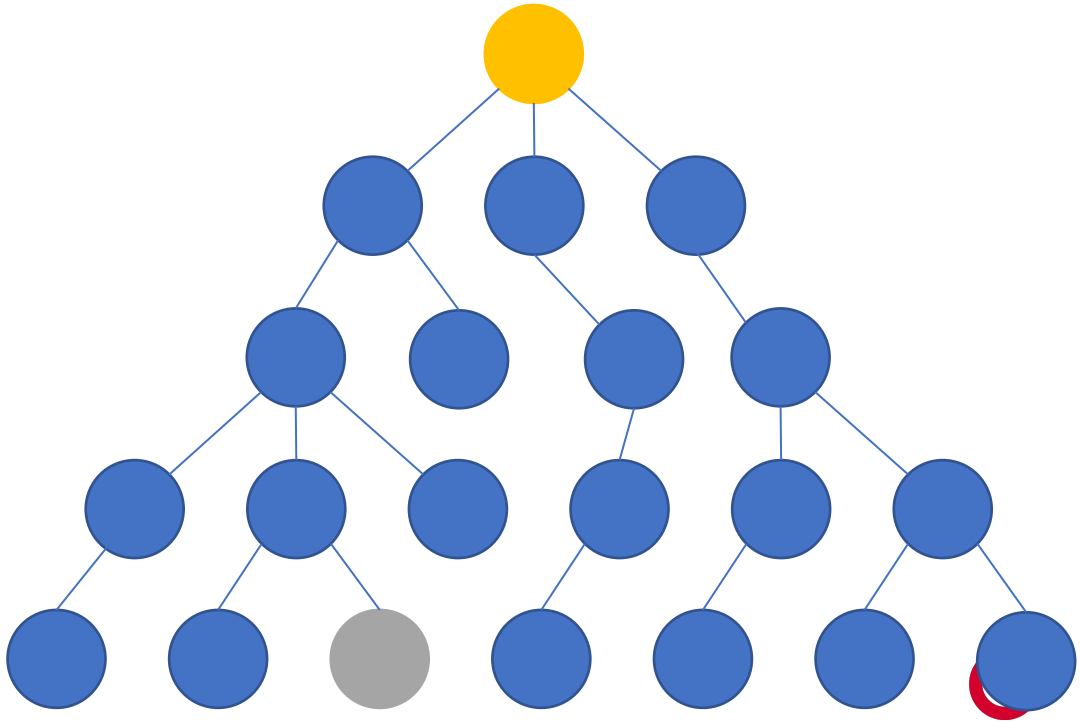


Called 'props drilling'



What if I need to send data down like 10 levels?

- Use props drilling or ...
- Look into the useContext() hook or ...
- Read it from the Redux store



... or if you're using Redux

- At any level, you can get a reference to your Redux store and call getState() to get the data. No need to pass it through props.
- If you're using react-redux library, they have a <Provider> that will get a reference at any level without passing through props. You'll call useSelector(state => state.whateverYouWant)

tl;dr

- To pass data down, write the values as attributes in the host and read them in *props* in the inner where they're immutable
- Use props drilling to send down multiple levels

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

tl;dr

- The 2 primary ways to style React components

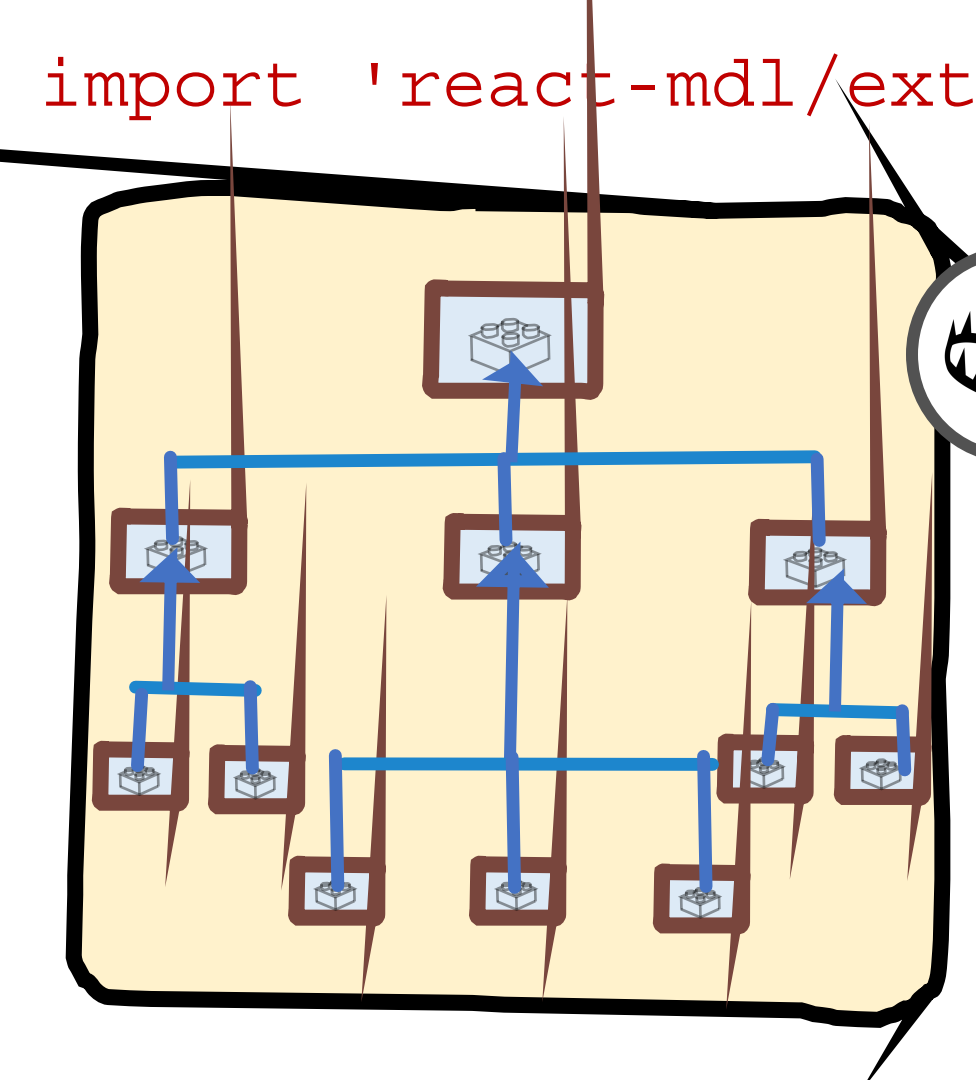
1. Import a CSS file for global styles
2. Inline JavaScript styling for local styles

The two main methods to style React components

1. Importing traditional CSS files
2. Inline JavaScript styles

Importing CSS files

```
import 'react-mdl/extra/material.css';
```



Note: no "from" in the import

Then you're using it just like you do with regular CSS

```
<div className="big fancy">...</div>
<ul className="no-bullets">...</ul>
<input className="form-control" />
<img src="img.jpg" className="rounded" />
```

Remember, you can't use 'class'. You must use 'className'

Inline styling

Box.js

```
import React from 'react';

const divStyle = {
 margin: '40px',
 border: '5px solid pink'
};

export function Box() {
 const pStyle = {
  fontSize: '15px',
  textAlign: 'center'
 };
 return (
  <div style={divStyle}>
   <p style={pStyle}>
    Text goes here
   </p>
 </div>)
}
```

Seems so low-level

So non-SOC

So non-SRP

This is Facebook's recommended method.

tl;dr

- The 2 primary ways to style React components

1. Import a CSS file for global styles
2. Inline JavaScript styling for local styles

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

tl;dr

- JSX strips out the native HTML events and replaces them with their own.
- They're called synthetic events and they don't behave exactly like their native mirrors

Native browser events are stripped out of JSX by React

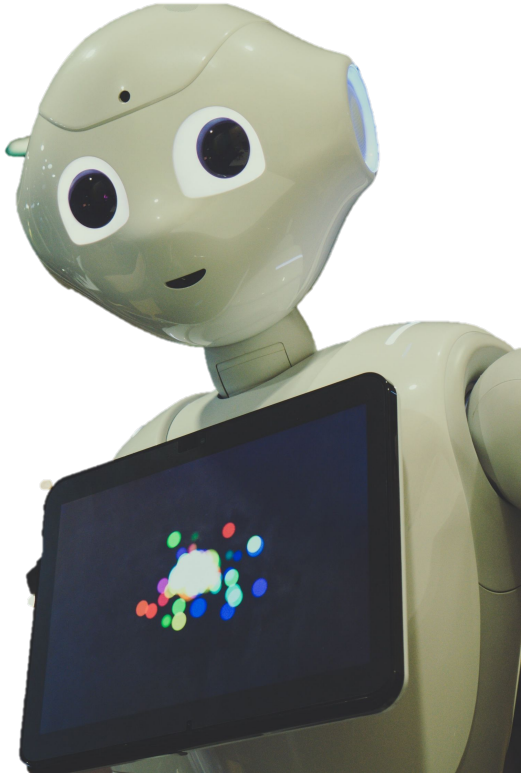
- So if you write

```
<foo onclick="alert('foo')" />
```

- Your onclick will be ignored.
- If it starts with "on", and is not a synthetic event, you're warned in the console and it is stripped.
- We are forced to use React's synthetic events

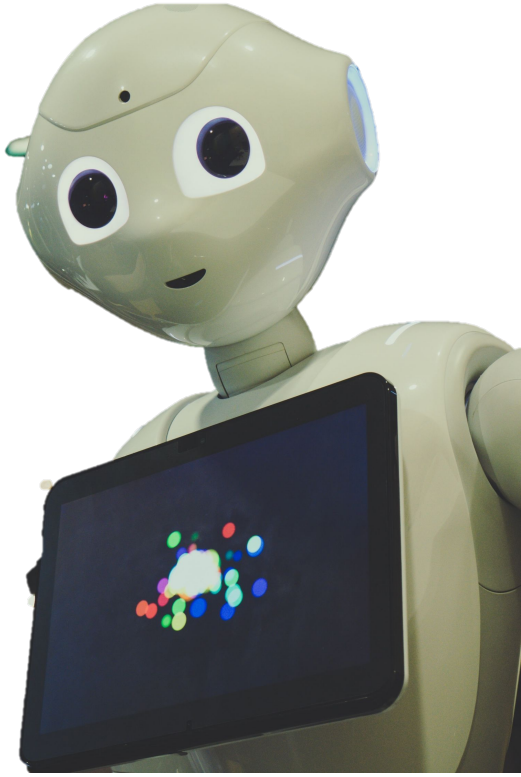
Synthetic

man-made
unnatural
pseudo
simulated
mock



React has created its own synthetic version of most events

- React overrides the native events with their own synthetic events
- Most valid w3c events are re-implemented by React ...



There are unsupported events

1. Window- and Browser-level events
   - beforePrint, hashChange, resize, message, DOMContentLoaded, beforeunload, load,
2. Experimental events
   - They eventually get support after they're mainstream
   - (eg. Device events, Touch events, pointer events are new-ish)

Events that ARE supported

https://reactjs.org/docs/events.html

onCopy onCut onPaste onKeyDown onKeyPress onKeyUp onFocus onBlur onChange onInput onInvalid onReset onSubmit onError onLoad onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut onSelect onTouchCancel onTouchEnd onTouchMove onTouchStart onScroll onWheel onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting onAnimationStart onAnimationEnd onAnimationIteration onTransitionEnd onToggle

Camel case the event and precede it by *on*

```
<any onFoo={bar}></any>
```

Hey, React!
When the user triggers the *foo* event, run the *bar* function.

Examples:

```
<button onClick={doIt}>
  Press me</button>
<img onMouseOver={count} />
<input onBlur={go}
       onKeyUp={run} />
```

Passing values to the handler

Say you have an event handler function:

MyComponent.js

```
...
function addPerson(person) {
  console.log("Person was added", person);
  try {
   insertIntoDB(person);
   return true;
  } catch {
   return false;
  }
}
```

And you have some JSX:

MyComponent.js

```
let person = {};
return (
 <form onSubmit={addPerson}>
  <input value={person.first} />
  <input value={person.last} />
  <input type="submit" />
 </form>
);
```

How does the person object get sent to addPerson?!?

Solution! To pass a parameter, use an arrow function

MyComponent.js

```
let person = {};
return (
 <form onSubmit={() => addPerson(person)}>
  <input value={person.first} />
  <input value={person.last} />
  <input type="submit" />
 </form>
);
```

And to pass the event object ...

MyComponent.js

```
...
return (
 <button onDoubleClick={e => doStuff(e, obj1, obj2)}>
  Click me!
 </button>
);
```

This works because when an event is triggered, the (synthetic) event object is passed into the registered function

tl;dr

- JSX strips out the native HTML events and replaces them with their own.
- They're called synthetic events and they don't behave exactly like their native mirrors

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State

tl;dr

- State is data that changes through the life of your React component.
- The useState() hook allows us to manage state values

State is the displayed, mutable data of your React component

State is totally local to a component

Neither a parent nor a child can know if a component maintains its own state or not.

If you need to pass data from a parent to a child, that is not state. That is props. (More about props later).

State is not props!

state

- Data known by a component
- Can be changed in the component
- Lives 100% within this component

props

- Data known by a component
- Once set, they don't change
- Passed in to this component from its parent

The best way to handle state is with a useState hook

Calling useState returns an array with two things:

```
const theArray = useState(initialState);
const state = theArray[0];
const setState = theArray[1];
```

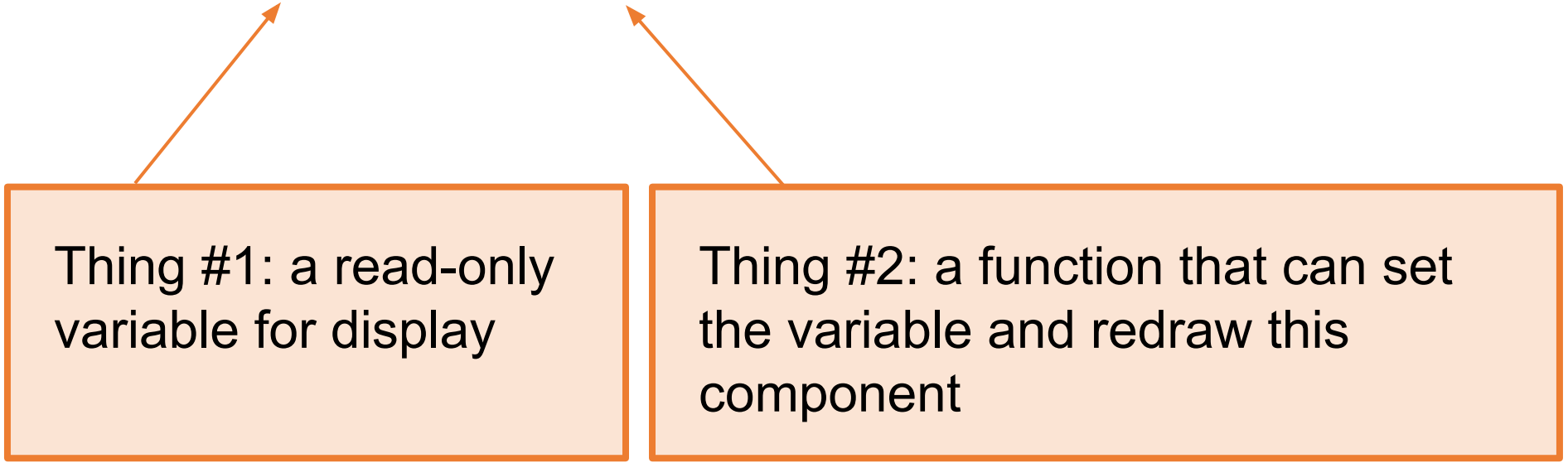
Thing #1: a read-only variable for display

Thing #2: a function that can set the variable and redraw this component

The best way to handle state is with a useState hook

... or more concisely ...

```
const [state, setState] = useState(initialState);
```



How to use state

```
import React, { useState } from "react";

export const FirstNameInput = ({firstName}) => {
 const [fName, setFName]= useState(firstName);

 return (
  <input
    onChange={e => setFName(e.target.value)}
    value={fName}
  />
 );
};
```

Lazy initialState

- When re-rendering, the state persists.
- React saves the last value and uses it again when re-drawing. It only gets the initialState on the first render.

State is clobbered, not appended!

- When you call setState, it overwrites the existing state.
- So if you want to add to state, you must use object spread:

```
setFoo({
  ...oldFoo,
  key: newValue,
  key2:newValue2
});
```

No!

```
// { first: "Jo", last: "Kim" }
setUser({ last: "Lee" });
// { last: "Lee"}
```

Use spread to append

Yes

```
// { first: "Jo", last: "Kim" }
setUser({ ...user, last: "Lee" });
// { first: "Jo", last: "Lee" }
```

What if state is a complex object?

- Two options

1. Have multiple state variables
   - const [first, setFirst] = useState("");
   - const [last, setLast] = useState("");
2. Have one big state object varaible
   - const [state, setState] = useState({});

- Confirmed - setState() is async. A console.log() before and after are exactly the same.
- Can you make them sync? No! But you can register a useEffect that fires when a value is updated. Using that, you can respond when a change occurs.
- When using >1 useState for >1 variables, and you update them both ... the updates are independent. They are not updated at the same moment. If you want them to be, wrap them in a Promise.

tl;dr

- State is data that changes through the life of your React component.
- The useState() hook allows us to manage state values

Introduction to React

How to create a React app

Creating a React component

Props

Styling

Events

State