React is Not Composable!

Alternate title:

Components can be useful
but let's not get carried away.

ReactJS Utah 2018

by Seth House

@whiteinge
seth@eseth.com

React: A (Short) History.

The 'V' in MVC.

(2013)

Declarative Rendering

brought us out of the dark ages of imperative DOM juggling.

Had to convince the industry of several new ideas.

Had to convince the industry of several new ideas.

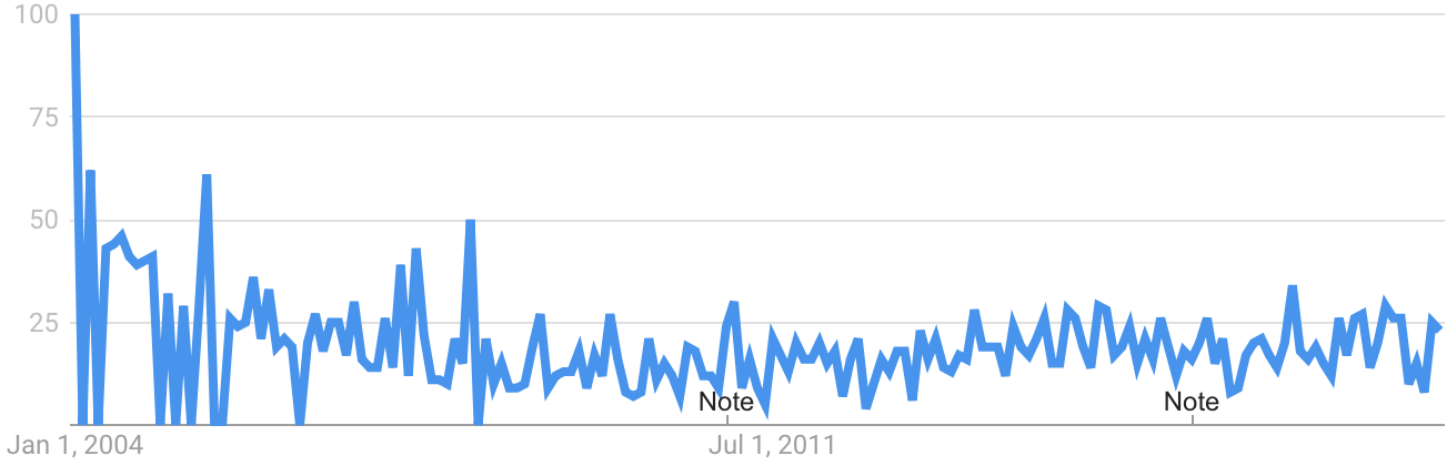
- How on earth is this performant?

Had to convince the industry of several new ideas.

- How on earth is this performant?
- JSX is both HTML and JavaScript, but it's just another view language. (Templating i.e., Handlebars, Mustache, etc.)

Had to convince the industry of several new ideas.

- How on earth is this performant?
- JSX is both HTML and JavaScript, but it's just another view language. (Templating i.e., Handlebars, Mustache, etc.)
- Why doesn't this violate separation of concerns?



Separation of Concerns 2004-2018

One traditional view is:

- HTML - Model
- CSS - View
- JavaScript - Controller

But what's the difference with:

- JavaScript - Model
- JavaScript - View
- JavaScript - Controller

Separating HTML and JS is separating technologies.
It is not separation of concerns.

– Rob Eisenberg (Aurelia)

Continued mainstreaming functional ideas in JS.

```
const app = view(state)
```

React's true strengths: composition, unidirectional data flow, freedom from DSLs, explicit mutation and static mental model.

– Dan Abramov (2015)

But!

- Routing.
- Controllers.
- Data stores.
- Data fetching.
- Code organiztion.
- App organiztion.
- Development environment.

Alt, Barracks, Delorean, disto, fluce, fluctuations, Flummox, Flux, Flux This, Fluxette, Fluxible, Fluxxor, Fluxy, Lux, Marty.js, Material Flux, McFly, microcosm, microflux, mmox, Nuclear.js, NuclearJS, OmniscientJS, Reducer, Redux, Reflux.

(Flux variants circa 2015)

Late 2015: The community starts to settle around a few projects.

So long, V in the MVC!

– Dan Abramov (2016)

The future?

- create-react-app.
- React Router.
- Redux.
- Context.
- Suspense.

Composition.

Object-oriented composition.

- Mixins.
- Managing instances of other classes from within a class method.

Ad-hoc and manual.

Functional composition.

```
const compose = (f, g) ⇒ (...args) ⇒ f(g(...args))
```

The application of one function to the result from another function.

```
SomeContainer
    .map(fnFoo)
    .map(fnBar)
    .map(fnBaz)
```

Informal composition.

"Using different things together."

Is React OOP or functional?

OOP?

```
class MyComponent extends React.Component {
    someMethod() {...}
    render() { return <div>...</div> }
}
```

Is React OOP or functional?

OOP?

```
class MyComponent extends React.Component {
    someMethod() {...}
    render() { return <div>...</div> }
}
```

Functional?

```
const MyComponent = (props) => <div>...</div>
```

What composition does React use?

Higher-order components.

```
function wrapInHOC(OldComponent) {
    const newComponent = (moreProps) => <div>
        <OldComponent {...moreProps}/>
    </div>
}

const EnhancedComponent = wrapInHOC(WrappedComponent);
```

> [A] higher-order component is a function that takes a component and returns a new component.
>
> – https://reactjs.org/docs/higher-order-components.html

Higher-order components.

```
function wrapInHOC(OldComponent) {
    const newComponent = (moreProps) => <div>
        <OldComponent {...moreProps}/>
    </div>
}

const EnhancedComponent = wrapInHOC(WrappedComponent);
```

Higher-order components.

```
function wrapInHOC(OldComponent) {
    const newComponent = (moreProps) => <div>
        <OldComponent {...moreProps}/>
    </div>
}

const EnhancedComponent = wrapInHOC(WrappedComponent);
```

```
function wrapInFn(oldFunction) {
    return function newFunction(...oldArgs) {
        return oldFunction(newArgs, ...oldArgs)
    };
}
```

Higher-order components.

```
function wrapInHOC(OldComponent) {
    const newComponent = (moreProps) => <div>
        <OldComponent {...moreProps}/>
    </div>
}

const EnhancedComponent = wrapInHOC(WrappedComponent);
```

```
function wrapInFn(oldFunction) {
    return function newFunction(...oldArgs) {
        return oldFunction(newArgs, ...oldArgs)
    };
}
```

- Function decorator pattern?

Higher-order components.

```
function wrapInHOC(OldComponent) {
    const newComponent = (moreProps) => <div>
        <OldComponent {...moreProps}/>
    </div>
}

const EnhancedComponent = wrapInHOC(WrappedComponent);
```

```
function wrapInFn(oldFunction) {
    return function newFunction(...oldArgs) {
        return oldFunction(newArgs, ...oldArgs)
    };
}
```

- Function decorator pattern?
- Perhaps currying?

  `addBazProp(addBarProp(addFooProp(WrappedComponent)))`

Higher-order components.

Toss it in a closure.

Benefits:

- Easy to implement

Drawbacks:

- Everything has access to settings everywhere whether it needs it or not.
- Implicitly stateful
- Need data up front

– Tail of three envs

Render props.

```
class LogicComponent extends React.Component {
    ...stuff...
    render() {
        return this.props.someRenderFunction(this.state)
    }
}
```

> A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.
>
> – https://reactjs.org/docs/render-props.html

Render props.

Benefits:

- Separate business logic or utility logic from rendering.
- Better testability.

Drawbacks:

- Data must be pre-processed (formatted, grouped, culled, etc).
- Must be fast and efficient.
- Render function has no control over when or how often it is called.
- Author of render function has no control over where it is placed.
- Subject to props tunneling.
- Exist solely to work around the drawbacks of classes.

Render reducers (😱).

```
<SomeComponent
    reducerFoo={someFunction)
    renderFoo={otherFunction} />
```

Render reducers (😱).

```
<SomeComponent
    reducerFoo={someFunction)
    renderFoo={otherFunction} />
```

Benefits:

- Separate state management from rendering.
- Better testability.

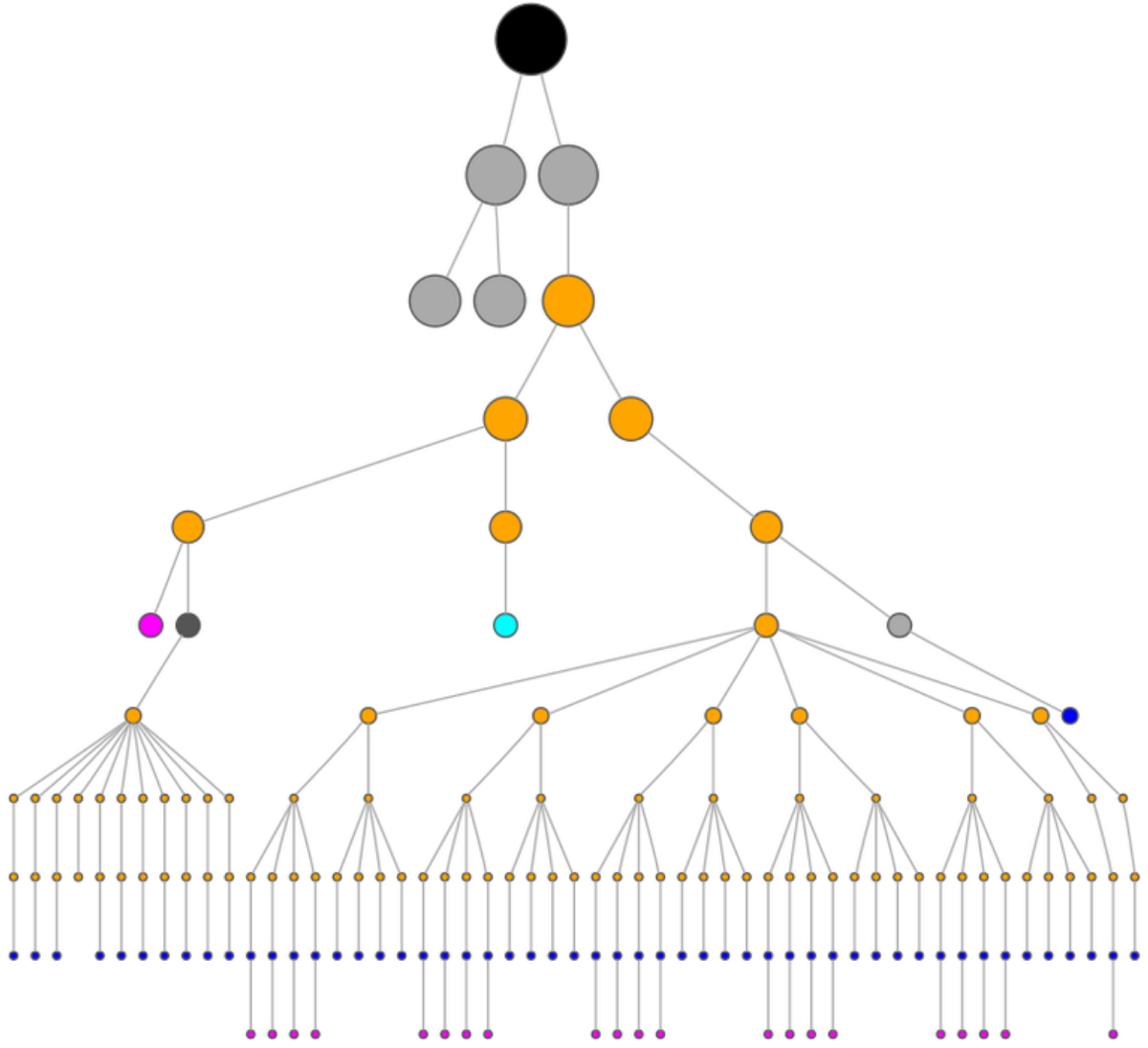
Drawbacks:

- User of a reducer function must be very aware of the performance characteristics and intended usage of the function.
- Reducer function composition inside render function.
- No effects management (without reinventing Flux).
- Subject to props tunneling.
- Exist solely to work around the Drawbacks of classes.

Interlude.

Trees.

(The composite pattern.)



Message passing.

(And state management.)

Message passing.

(And state management.)

- Parent-to-child: rebuild a subset of the tree.

Message passing.

(And state management.)

- Parent-to-child: rebuild a subset of the tree.
- Child-to-parent: event bubbling
  (chain of responsibility pattern).

Message passing.

(And state management.)

- Parent-to-child: rebuild a subset of the tree.
- Child-to-parent: event bubbling
  (chain of responsibility pattern).
- Sideload data external to the DOM tree
  (flyweight pattern + pub/sub or observer patterns).

Message passing.

(And state management.)

- Parent-to-child: rebuild a subset of the tree.
- Child-to-parent: event bubbling
  (chain of responsibility pattern).
- Sideload data external to the DOM tree
  (flyweight pattern + pub/sub or observer patterns).

  The flyweight pattern shows how to rework a design without storing parents altogether. It works in cases where children can avoid sending parent requests by externalizing some or all of their state.

  – Gang of Four book pg. 167

React component tree.

- Components are *not* the DOM tree.

React component tree.

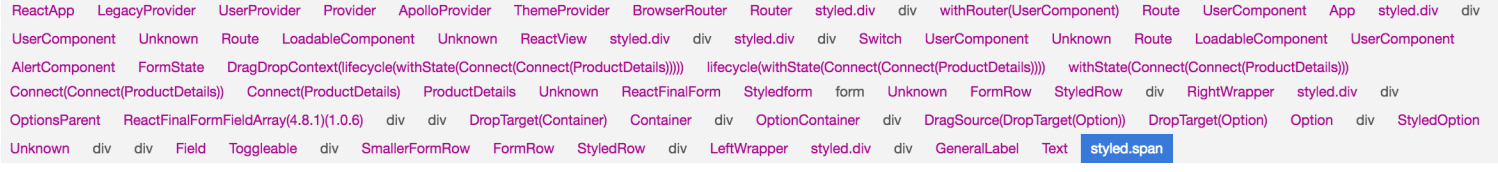
- Components are *not* the DOM tree.
- But they can be used to construct one.

React component tree.

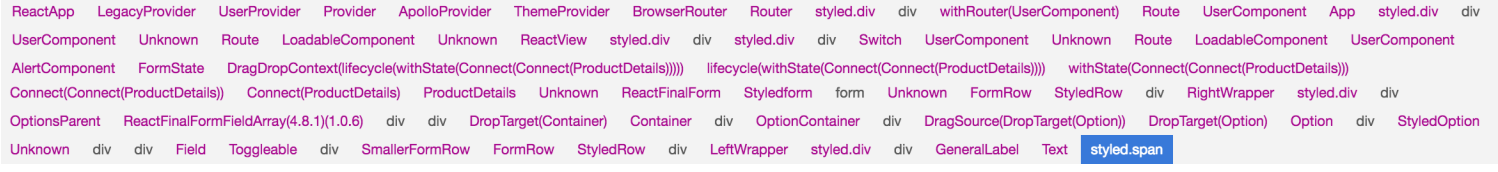
- Components are *not* the DOM tree.
- But they can be used to construct one.
- Encapsulate logic or behavior.

React component tree.

- Components are *not* the DOM tree.
- But they can be used to construct one.
- Encapsulate logic or behavior.
- A tree can store state and arbitrary other things like behavior and logic.



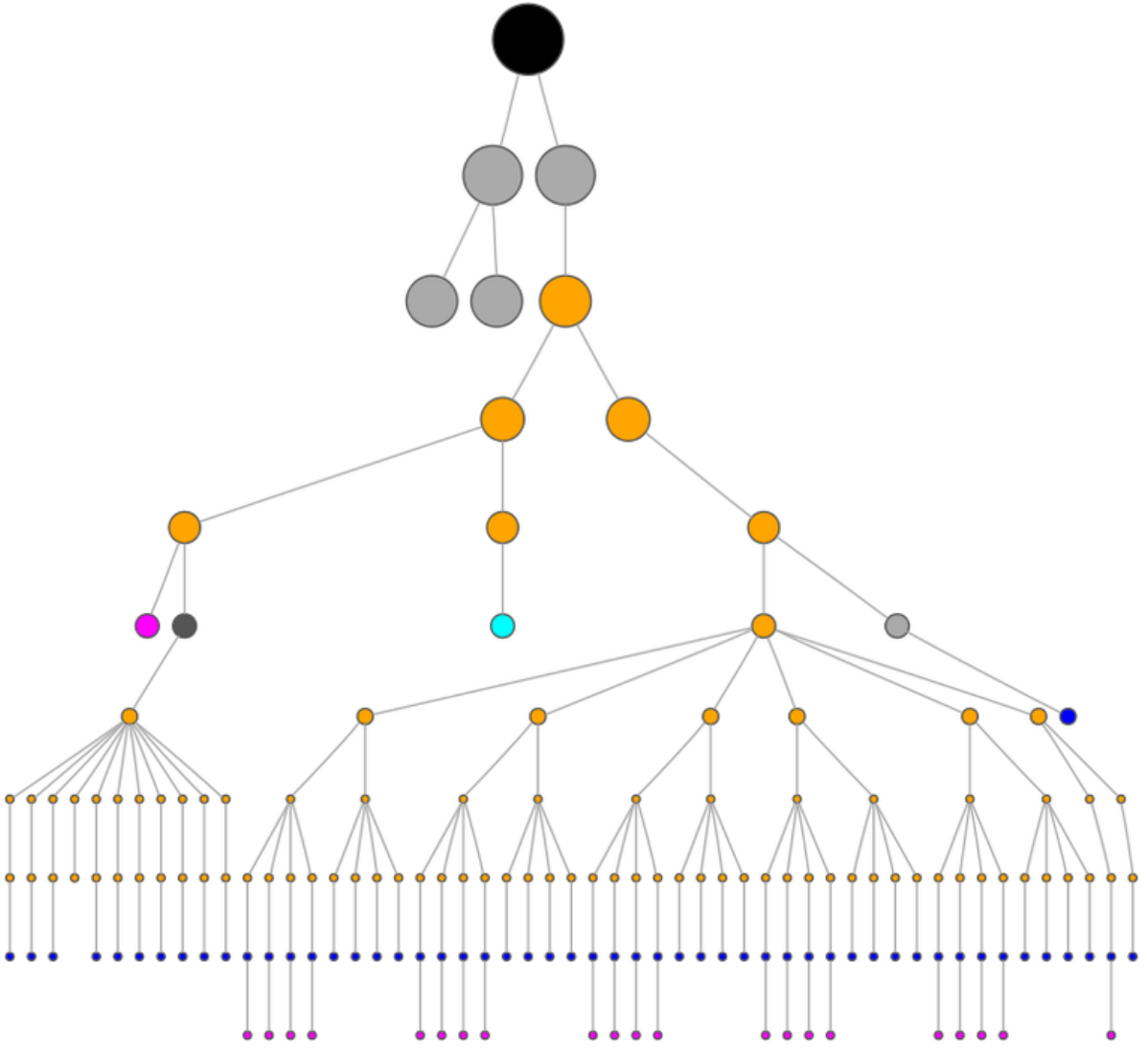
Component tree (81 components deep).



Component tree (81 components deep).

Corresponding DOM tree (21 elements deep).

A component-local-state scenario.



Context is not state management.

Context is not state management.

> Maybe you don’t care about the easy debugging, the customization, or the automatic performance improvements – all you want to do is pass data around easily.
>
> – https://daveceddia.com/context-api-vs-redux/

The jury is still out on Suspense.

- Convenient API.
- But we can do some (all?) of Suspense with better async primitives.
  - http://talks.eseth.com#js-adts

End of interlude.

(Back to) Components.

Props are...

Props are...

- Component attributes (configuration).

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.
- Child-component styles.

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.
- Child-component styles.
- Other components (HoC).

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.
- Child-component styles.
- Other components (HoC).
- Sub-render functions.

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.
- Child-component styles.
- Other components (HoC).
- Sub-render functions.
- State management functions.

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.
- Child-component styles.
- Other components (HoC).
- Sub-render functions.
- State management functions.
- Tunneled through wrapper components to distant children.

Props are...

- Component attributes (configuration).
- Message passing (parent-to-child via changing props).
- Message passing (child-to-parent via passing a function the child must call).
- Function arguments (data).
- Component styles.
- Child-component styles.
- Other components (HoC).
- Sub-render functions.
- State management functions.
- Tunneled through wrapper components to distant children.

...not composable.

Components...

Components...

- Are often stateful classes.

Components…

- Are often stateful classes.
- Have a very stateful API:
  - Impure `render()`.
  - Missing current value in `setState()` callback.

Components…

- Are often stateful classes.
- Have a very stateful API:
  - Impure `render()`.
  - Missing current value in `setState()` callback.
- Require lifecycle methods for day-to-day performance tuning.

Components…

- Are often stateful classes.
- Have a very stateful API:
  - Impure `render()`.
  - Missing current value in `setState()` callback.
- Require lifecycle methods for day-to-day performance tuning.
- Highly coupled to where they live in the component tree.

Components…

- Are often stateful classes.
- Have a very stateful API:
  - Impure `render()`.
  - Missing current value in `setState()` callback.
- Require lifecycle methods for day-to-day performance tuning.
- Highly coupled to where they live in the component tree.
- Encapsulate state, business logic, data fetching, effects, and view.

Components…

- Are often stateful classes.
- Have a very stateful API:
  - Impure `render()`.
  - Missing current value in `setState()` callback.
- Require lifecycle methods for day-to-day performance tuning.
- Highly coupled to where they live in the component tree.
- Encapsulate state, business logic, data fetching, effects, and view.
  - All of which do compose!
  - But they compose completely differently.

Components...

- Are often stateful classes.
- Have a very stateful API:
  - Impure `render()`.
  - Missing current value in `setState()` callback.
- Require lifecycle methods for day-to-day performance tuning.
- Highly coupled to where they live in the component tree.
- Encapsulate state, business logic, data fetching, effects, and view.
  - All of which do compose!
  - But they compose completely differently.

...are not composable.

Closing thoughts.

We're still figuring out how to make UIs!

We're still figuring out how to make UIs!

- The software industry is crazy young.

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?
- How many times are we going to write confusing, unmaintainable behemoths?

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?
- How many times are we going to write confusing, unmaintainable behemoths?
- We need people who care to keep discussing patterns.

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?
- How many times are we going to write confusing, unmaintainable behemoths?
- We need people who care to keep discussing patterns.
- Focus on programming fundamentals and techniques.

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?
- How many times are we going to write confusing, unmaintainable behemoths?
- We need people who care to keep discussing patterns.
- Focus on programming fundamentals and techniques.
- Read books.

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?
- How many times are we going to write confusing, unmaintainable behemoths?
- We need people who care to keep discussing patterns.
- Focus on programming fundamentals and techniques.
- Read books.
- Really *learn* JavaScript.

We're still figuring out how to make UIs!

- The software industry is crazy young.
- How many times are we going to rewrite our apps over and over for each new framework?
- How many times are we going to write confusing, unmaintainable behemoths?
- We need people who care to keep discussing patterns.
- Focus on programming fundamentals and techniques.
- Read books.
- Really *learn* JavaScript.
- Be wary of the new framework du jour.

I think developers have been sold this idea of a competitive landscape by authors of these frameworks because it helps sell the framework. You can build and strengthen a community by leaning into the tribalism that can surround the usage of a tool.

The older I've gotten as a person who was deeply tribalistic about Ruby on Rails when I got into it and Ember when I got into it, because I love tribes, I think tribes are awesome and it's a way to make friends but when you really lean into that, the costs are too high and experimenting with other technologies and noticing flaws in your own technology is not only not a betrayal, it's actually critical to your growth as a developer.

– The Frontside Podcast https://frontsidethepodcast.simplecast.fm/25857c8d

People are interested in exploring composition!

The compile-to-JS functional communities are a wealth of ideas.

Brian Lonsdorf - Oh Composable World! (React Rally 2016)

We need composable, async-first
state management ideas.

We need good message passing primitives.

Evaluating a new framework or library.

- Does the library espouse a new idea or technique?

Evaluating a new framework or library.

- Does the library espouse a new idea or technique?
- What are the author's motivations?

Evaluating a new framework or library.

- Does the library espouse a new idea or technique?
- What are the author's motivations?
- Do they use React in their day-job?

Evaluating a new framework or library.

- Does the library espouse a new idea or technique?
- What are the author's motivations?
- Do they use React in their day-job?
- Watch out for hero-worship (especially from the Big Tech Companies).

Evaluating a new framework or library.

- Does the library espouse a new idea or technique?
- What are the author's motivations?
- Do they use React in their day-job?
- Watch out for hero-worship (especially from the Big Tech Companies).
- Watch out for bike-shedding and personal preference. Stick to demonstrable improvements in *maintainability* over time.

Software Engineering is still an aspiration...
...because Computer Science is not yet a science.

– Ruth Ravenel, U of Co 1995

Not everything should be a component.

- A component is indirection! Only use as many as you need.

Not everything should be a component.

- A component is indirection! Only use as many as you need.
- Write regular JavaScript where you can.

Not everything should be a component.

- A component is indirection! Only use as many as you need.
- Write regular JavaScript where you can.
- Useful but in moderation.