



# Advanced State Management

**Steve Kinney**

A Frontend Masters Workshop

**Hi, I'm Steve.  
(@stevekinney)**





SendGrid Marketing Campaign X Steve

Secure | https://sendgrid.com/marketing\_campaigns/ui/campaigns/1917514/edit

DESIGN PREVIEW Save Draft Send Campaign

Settings Build Tags A/B Testing T Format Font Size B U Tx

From:  
Subject:  
Preheader:

**MODULE STYLES**

**BUTTON**

Button Color #333333 Border Color #333333 Font Color #FFFFFF

Width AUTO % Height 16 px

Padding ↑ 12 px → 18 px ↓ 12 px ← 18 px

Border Radius 6 px Border Width 1 px

Font Family Arial

Font Size 16 px Font Weight normal

This is my awesome button

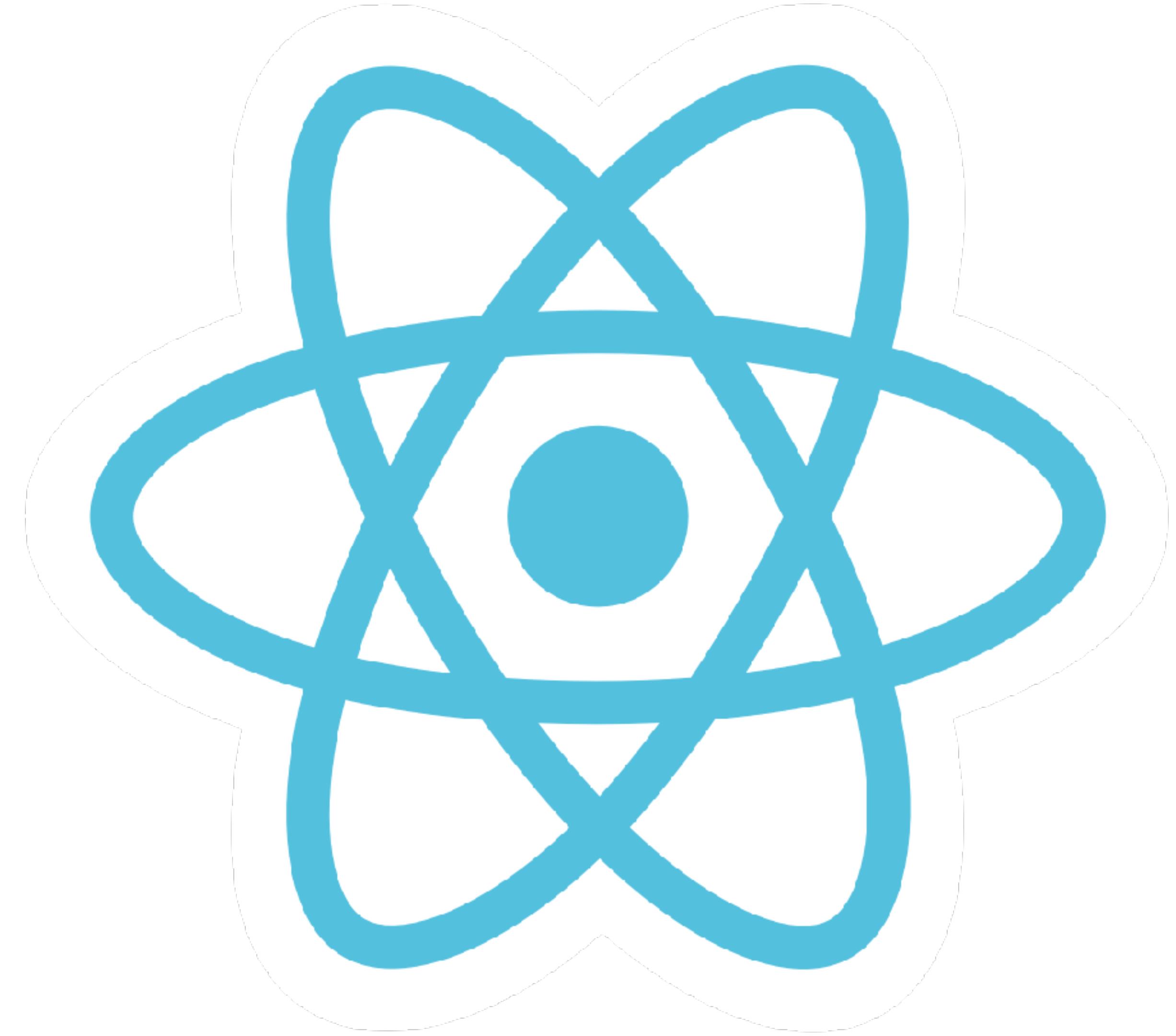
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam tincidunt elementum sem non luctus. Ut dolor nisl, facilisis non magna quis, elementum ultricies tortor. In mattis, purus ut tincidunt egestas, ligula nulla accumsan justo, vitae bibendum orci ligula id ipsum. Nunc elementum tincidunt libero, in ullamcorper magna volutpat a.

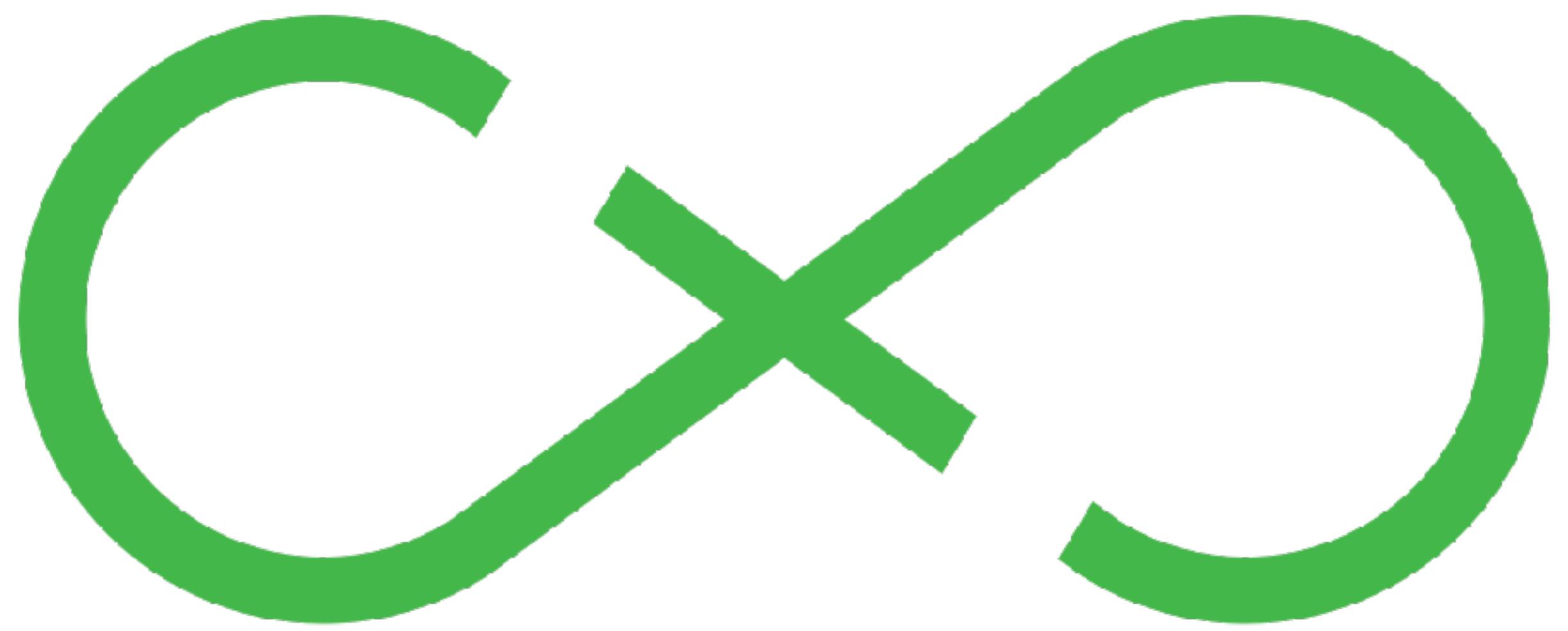
mobx (1).svg react.svg mobx.svg flux.svg Show All

**We're going to talk about  
state.**

To build our understanding of how to manage state, we're going to take a whirlwind tour of a number of approaches.

We're going to start from the very basics and work our way up.











JVL

But, this workshop is about  
*more* than just the libraries.

Libraries come and go.

**Patterns and approaches  
stick around.**

Managing UI state is not a solved problem. New ideas and implementations will come along.

My goal is to help you think about and apply these conceptual patterns, regardless of what library is the current flavor.

My goal is that if something new comes along, you're able to apply concepts you learned during this workshop.

# Part One

## Understanding State

The main job of React is to take  
your application state and turn it  
into DOM nodes.

reactjs.org/docs/thinking-in-react.html#step-4-identify-where-your-state-shou

React Docs Tutorial Community Blog Search docs v16.1.1 GitHub

# Thinking in React

React is, in our opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes you think about apps as you build them. In this document, we'll walk you through the thought process of building a searchable product data table using React.

## Start With A Mock

Imagine that we already have a JSON API and a mock from our designer. The mock looks like this:

The screenshot shows a search bar with placeholder text "Search...". Below it is a checkbox labeled "Only show products in stock". Underneath is a table with columns "Name" and "Price". The table has a header row "Sporting Goods" and two data rows: "Football \$49.99" and "Baseball \$9.99".

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99

QUICK START ^

- Installation
- Hello World
- Introducing JSX
- Rendering Elements
- Components and Props
- State and Lifecycle
- Handling Events
- Conditional Rendering
- Lists and Keys
- Forms
- Lifting State Up
- Composition vs Inheritance

Thinking In React

ADVANCED GUIDES ▾

REFERENCE ▾

CONTRIBUTING ▾

FAQ ▾

“  
The key here is **DRY: Don’t Repeat Yourself**. Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand.

”

“

1. *Is it passed in from a parent via props? If so, it probably isn't state.*
2. *Does it remain unchanged over time? If so, it probably isn't state.*
3. *Can you compute it based on any other state or props in your component? If so, it isn't state.*

”

“

*Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state. **This is often the most challenging part for newcomers to understand.***

”

props vs. state

State is created in the component  
and stays in the component. It can  
be passed to a children as its props.

```
class Counter extends Component {
  state = { count: 0 }
  render() {
    const { count } = this.state;
    return (
      <div>
        <h2>This is my state: { count }.</h2>
        <DoubleCount count={count} />
      </div>
    )
  }
}
```

```
class DoubleCount extends Component {
  render() {
    const { count } = this.props;
    return (
      <p>This is my prop: { count }.</p>
    )
  }
}
```

All state is not created equal.

# There are many kinds of state.

- **Model data:** The nouns in your application.
- **View/UI state:** Are those nouns sorted in ascending or descending order?
- **Session state:** Is the user even logged in?
- **Communication:** Are we in the process of fetching the nouns from the server?
- **Location:** Where are we in the application? Which nouns are we looking at?

Or, it might make sense to think about state relative to time.

- **Long-lasting state:** This is likely the data in your application.
- **Ephemeral state:** Stuff like the value of an input field that will be wiped away when you hit “enter.”

**Ask yourself:** Does a input field need  
the same kind of state management as  
your model data?

**Ask yourself: What about  
form validation?**

**Ask yourself:** Does it make sense to store all of your data in one place or compartmentalize it?

**Spoiler alert: There is no  
silver bullet.**

# Part Two

## An Uncomfortably Close Look at React Component State

Let's start with the world's  
simplest React component.

# Exercise

- Okay, this is going to be a quick one: We're going to implement a little counter.
- <https://github.com/stevekinney/basic-counter>
- It will have three buttons:
  - Increment
  - Decrement
  - Reset
- **Your job:** Get it working with React's built-in component state.
- Only touch `<Counter>` for now.

Oh, wow—it looks like it's  
time for a pop quiz, already.

```
class Counter extends Component {  
  constructor() {  
    this.state = {  
      counter: 0  
    }  
  }  
  
  render() { ... }  
}
```

```
this.setState({ count: this.state.count + 1 });
this.setState({ count: this.state.count + 1 });
this.setState({ count: this.state.count + 1 });

console.log(this.state.count);
```

Any guesses?

0

`this.setState()` is  
asynchronous.

But, why?

React is trying to avoid  
unnecessary re-renders.

```
export default class Counter extends Component {
  constructor() {
    super();
    this.state = { count: 0 };
    this.increment = this.increment.bind(this);
  }

  increment() {...}

  render() {
    return (
      <section>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
      </section>
    )
  }
}
```

```
export default class Counter extends Component {  
  constructor() { ... }  
  
  increment() {  
    this.setState({ count: this.state.count + 1 });  
    this.setState({ count: this.state.count + 1 });  
    this.setState({ count: this.state.count + 1 });  
  }  
  
  render() { ... }  
}
```

What will the count be after the user's clicks the “Increment” button?

1

Effectively, you're queuing up  
state changes.

React will batch them up, figure out  
the result and then efficiently make  
that change.

Alright, here is the second  
question...

```
Object.assign(  
  {},  
  yourFirstCallToSetState,  
  yourSecondCallToSetState,  
  yourThirdCallToSetState,  
);
```

There is actually a bit more  
to this.setState().

**Fun fact:** Did you know that you can also pass a function in as an argument?

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

export default class Counter extends Component {
  constructor() { ... }

  increment() {
    this.setState((state) => { return { count: state.count + 1 } });
    this.setState((state) => { return { count: state.count + 1 } });
    this.setState((state) => { return { count: state.count + 1 } });
  }

  render() { ... }
}
```

Any guesses here?

3

When you pass functions to  
this.setState(), it plays  
through each of them.

But, the more useful feature is that it gives you some programmatic control.

Twitter, Inc. [twitter.com/dan\\_abramov/status/824308413559668744](https://twitter.com/dan_abramov/status/824308413559668744)

Home Moments Notifications Messages  Tweet X

Dan Abramov... m-and-c-in-m... James K. Nels... Redux'ing wit... Introduction |... Lowest Comm... A Case for set... javascript - D... Finding `state...

 **Dan Abramov**  
@dan\_abramov Following

Best kept React secret: you can declare state changes separately from the component classes.

```
function increment(state, props) {
  return {
    value: state.value + props.step
  };
}

function decrement(state, props) {
  return {
    value: state.value - props.step
  };
}

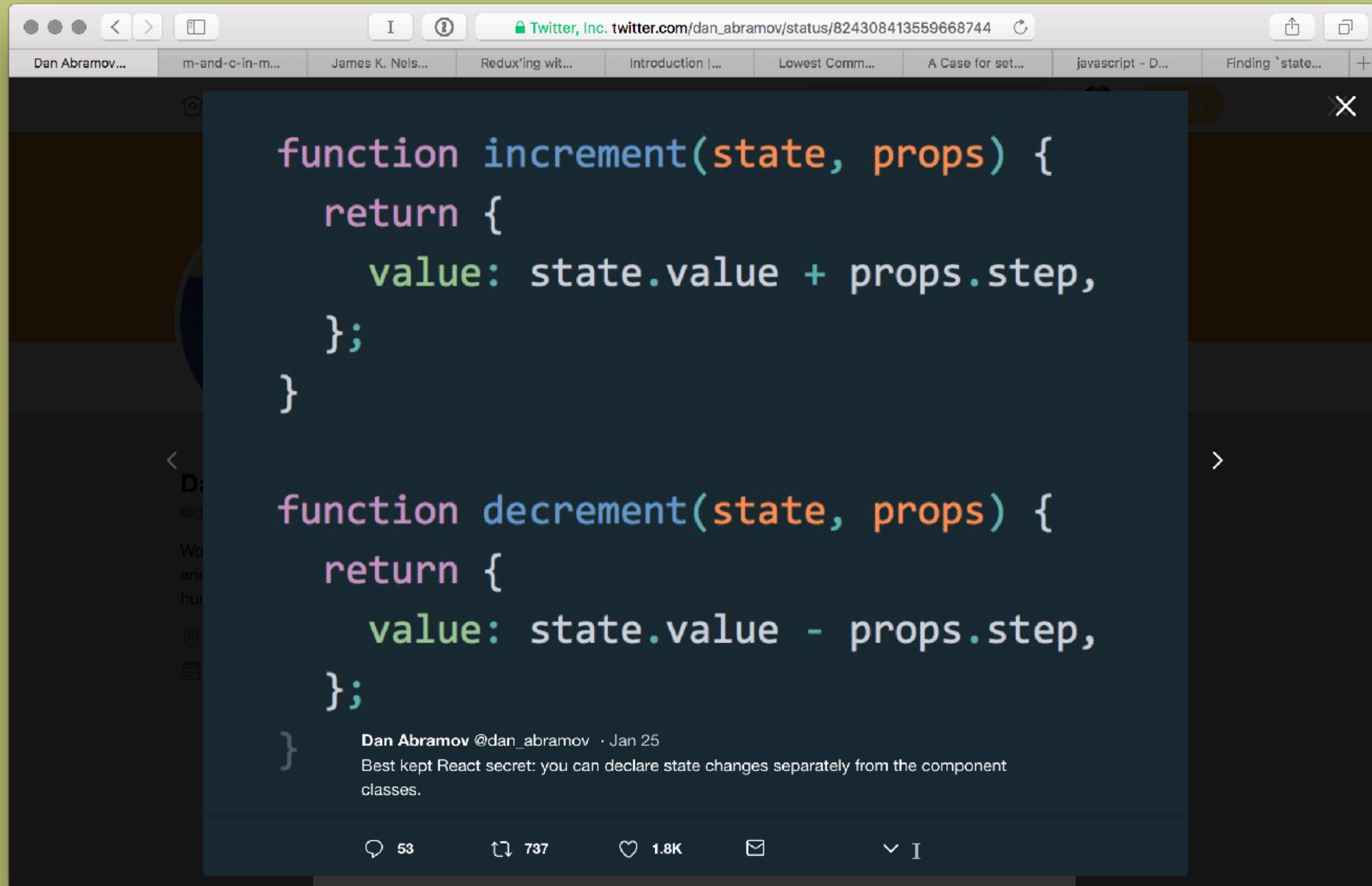
class Counter extends React.Component {
  state = { value: 0 };
  handleIncrement = () => {
    this.setState(increment);
  }
  handleDecrement = () => {
    this.setState(decrement);
  }
  render() {
    return (
      <div>
        <button onClick={this.handleIncrement}>+</button>
        <h1>{this.state.value}</h1>
        <button onClick={this.handleDecrement}>-</button>
      </div>
    );
  }
}

ReactDOM.render(
  <Counter step={5} />,
  document.getElementById('root')
);
```

10:30 AM - 25 Jan 2017

737 Retweets 1,806 Likes

 53  737  1.8K  



A screenshot of a Twitter post from Dan Abramov (@dan\_abramov) dated Jan 25. The post contains a block of code for a React component named `Counter`. The code uses class-based components, state, and props. It includes methods for incrementing and decrementing the counter value, and a render method that creates a UI with two buttons and a display. The code also demonstrates the use of `ReactDOM.render` to mount the component.

```
class Counter extends React.Component {
  state = { value: 0 };
  handleIncrement = () => {
    this.setState(increment);
  }
  handleDecrement = () => {
    this.setState(decrement);
  }
  render() {
    return (
      <div>
        <button onClick={this.handleIncrement}>+</button>
        <h1>{this.state.value}</h1>
        <button onClick={this.handleDecrement}>-</button>
      </div>
    )
  }
}

ReactDOM.render(
  <Counter step={5} />,
```

Dan Abramov @dan\_abramov · Jan 25  
document.getElementById('root')

Best kept React secret: you can declare state changes separately from the component classes.

53 737 1.8K I

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

export default class Counter extends Component {
  constructor() { ... }

  increment() {
    this.setState(state => {
      if (state.count ≥ 5) return;
      return { count: state.count + 1 };
    })
  }

  render() { ... }
}
```

this.setState also takes  
a callback.

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

export default class Counter extends Component {
  constructor() { ... }

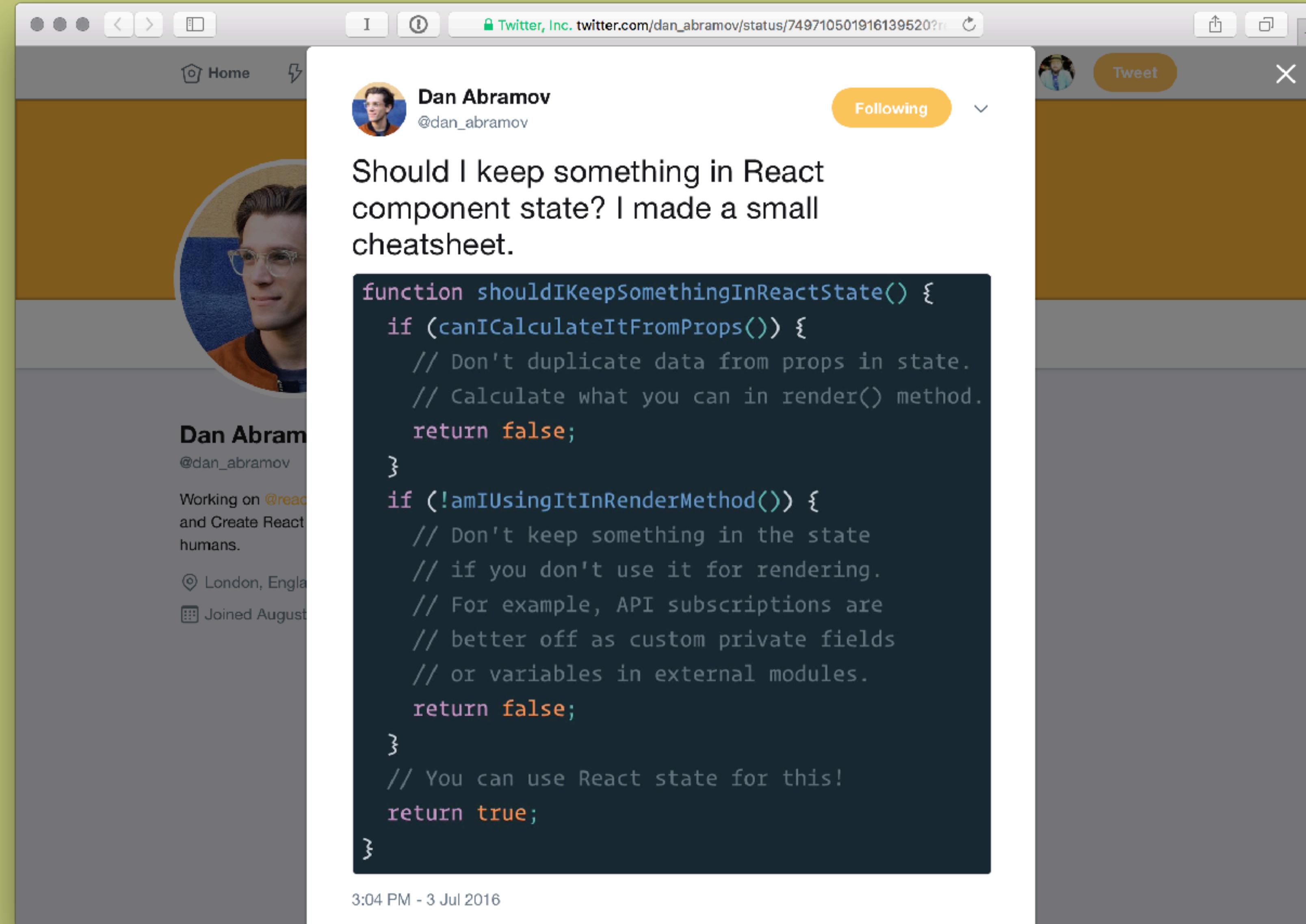
  increment() {
    this.setState(
      { count: this.state.count + 1 },
      () => { console.log(this.state); }
    )
  }

  render() { ... }
}
```

# **Patterns and anti-patterns**

State should be considered  
private data.

When we're working with props, we have PropTypes. That's not the case with state.

A screenshot of a Twitter browser window showing a tweet from Dan Abramov (@dan\_abramov). The tweet discusses whether to keep something in React component state, featuring a small cheatsheet in code block format. The tweet has a yellow 'Following' button and a 'Tweet' button. The background shows Dan Abramov's profile picture and bio.

**Dan Abramov**  
@dan\_abramov

Following

Tweet

X

Should I keep something in React component state? I made a small cheatsheet.

```
function shouldIKeepSomethingInReactState() {  
  if (canICalculateItFromProps()) {  
    // Don't duplicate data from props in state.  
    // Calculate what you can in render() method.  
    return false;  
  }  
  if (!amIUsingItInRenderMethod()) {  
    // Don't keep something in the state  
    // if you don't use it for rendering.  
    // For example, API subscriptions are  
    // better off as custom private fields  
    // or variables in external modules.  
    return false;  
  }  
  // You can use React state for this!  
  return true;  
}
```

3:04 PM - 3 Jul 2016

Don't use this.state for  
derivations of props.

```
class User extends Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      fullName: props.firstName + ' ' + props.lastName  
    };  
  }  
}
```

Don't do this. Instead, derive  
computed properties directly from  
the props themselves.

```
class User extends Component {  
  render() {  
    const { firstName, lastName } = this.props;  
    const fullName = firstName + ' ' + lastName;  
    return (  
      <h1>{fullName}</h1>  
    );  
  }  
}
```

```
// Alternatively...
class User extends Component {
  get fullName() {
    const { firstName, lastName } = this.props;
    return firstName + ' ' + lastName;
  }

  render() {
    return (
      <h1>{this.fullName}</h1>
    );
  }
}
```

You don't need to shove everything  
into your render method.

You can break things out into  
helper methods.

```
class UserList extends Component {
  render() {
    const { users } = this.props;
    return (
      <section>
        <VeryImportantUserControls >
          { users.map(user => (
            <UserProfile
              key={user.id}
              photograph={user.mugshot}
              onLayoff={handleLayoff}
            >
          )));
        <SomeSpecialFooter >
      </section>
    );
  }
}
```

```
class UserList extends Component {
  renderUserProfile(user) {
    return (
      <UserProfile
        key={user.id}
        photograph={user.mugshot}
        onLayoff={handleLayoff}
      />
    )
  }

  render() {
    const { users } = this.props;
    return (
      <section>
        <VeryImportantUserControls />
        { users.map(this.renderUserProfile) }
        <SomeSpecialFooter />
      </section>
    );
  }
}
```

Don't use state for things  
you're not going to render.

```
class TweetStream extends Component {
  constructor() {
    super();
    this.state = {
      tweets: [],
      tweetChecker: setInterval(() => {
        Api.getAll('/api/tweets').then(newTweets => {
          const { tweets } = this.state;
          this.setState({ tweets: [ ...tweets, newTweets ] });
        });
      }, 1000)
    }
  }

  componentWillUnmount() {
    clearInterval(this.state.tweetChecker);
  }

  render() { // Do stuff with tweets }
}
```

```
class TweetStream extends Component {  
  constructor() {  
    super();  
    this.state = {  
      tweets: [],  
    }  
  }  
  
  componentWillMount() {  
    this.tweetChecker = setInterval( ... );  
  }  
  
  componentWillUnmount() {  
    clearInterval(this.tweetChecker);  
  }  
  
  render() { // Do stuff with tweets }  
}
```

Use sensible defaults.

```
class Items extends Component {  
  constructor() {  
    super();  
  }  
  
  componentDidMount() {  
    Api.getAll('/api/items').then(items => {  
      this.setState({ items });  
    });  
  }  
  
  render() { // Do stuff with items }  
}
```

```
class Items extends Component {  
  constructor() {  
    super();  
    this.state = {  
      items: []  
    }  
  }  
  
  componentDidMount() {  
    Api.getAll('/api/items').then(items => {  
      this.setState({ items });  
    });  
  }  
  
  render() { // Do stuff with items }  
}
```

# **Example Application:**

# **Jetsetter**

Jetsetter x Steve

Secure | https://alert-tiger.glitch.me

New Item  Submit

**Unpacked Items (8)**

Pants [Remove](#)  
 Jacket [Remove](#)  
 iPhone Charger [Remove](#)  
 MacBook [Remove](#)  
 Underwear [Remove](#)  
 Hat [Remove](#)  
 T-Shirts [Remove](#)  
 Belt [Remove](#)

**Packed Items (3)**

Sleeping Pills [Remove](#)  
 Passport [Remove](#)  
 Sandwich [Remove](#)

# Exercise

- If you checkout the **master** branch, you'll find the components are there, but nothing is wired up just yet.
- **Your job:** get it working using React's built-in component state.



# Part Three

## State Architecture Patterns

React state is stored in a component and passed down as props to its children.

**“Data down. Events up.”**

reactjs.org/docs/thinking-in-react.html#step-4-identify-where-your-state-shou

React Docs Tutorial Community Blog Search docs v16.1.1 GitHub

# Thinking in React

React is, in our opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes you think about apps as you build them. In this document, we'll walk you through the thought process of building a searchable product data table using React.

---

## Start With A Mock

Imagine that we already have a JSON API and a mock from our designer. The mock looks like this:



The screenshot shows a search bar with placeholder text "Search...". Below it is a checkbox labeled "Only show products in stock". Underneath is a table with columns "Name" and "Price". The table has a single row labeled "Sporting Goods" with two items: "Football" and "\$49.99", and "Baseball" and "\$9.99".

QUICK START ^

- Installation
- Hello World
- Introducing JSX
- Rendering Elements
- Components and Props
- State and Lifecycle
- Handling Events
- Conditional Rendering
- Lists and Keys
- Forms
- Lifting State Up
- Composition vs Inheritance

Thinking In React

ADVANCED GUIDES ▾

REFERENCE ▾

CONTRIBUTING ▾

FAQ ▾

“  
*Identify every component that renders something based on that state.*

”

*Find a common owner component (a single component above all the components that need the state in the hierarchy).*

*Either the common owner or another component higher up in the hierarchy should own the state.*

*If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.*

*Though this sounds complex, it's really just a few lines of code. And it's really explicit how your data is flowing throughout the app.*

“  
While it may be a little more typing than you’re used to,  
remember that code is read far more than it’s written,  
and it’s extremely easy to read this modular, explicit  
code.

”

“  
As you start to build large libraries of components, you’ll appreciate this explicitness and modularity, and with code reuse, your lines of code will start to shrink. 😊

”

Maybe.

(Easier said than done.)

This effectively means that all state needs to live in the topmost common component that needs access.

# **The Lowest Common Ancestor**

Secure | https://blog.embermap.com/lowest-common-ancestor-fbf5d5313a1

Sign in Get started

Sam Selikoff [Follow](#)  
Software developer, Ember.js  
Dec 8, 2015 · 7 min read

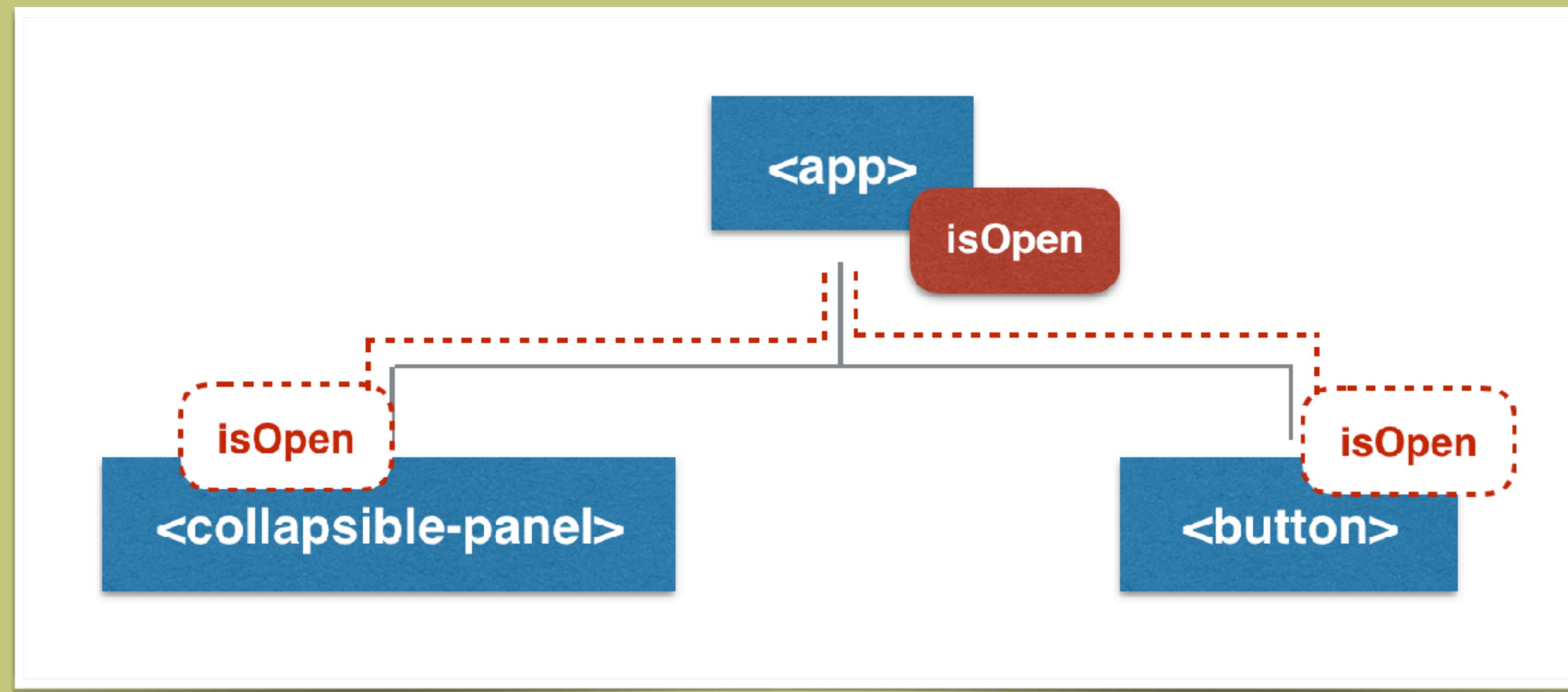
```
graph TD; app["<app>"] --> sidebar["<sidebar>"]; app --> main["<main>"]; main --> button["<button>"]; main -.-> panel["<collapsible-panel>"]; main -.-> button; panel -.-> isOpen1["isOpen"]; main -.-> isOpen2["isOpen"]; button -.-> isOpen3["isOpen"]
```

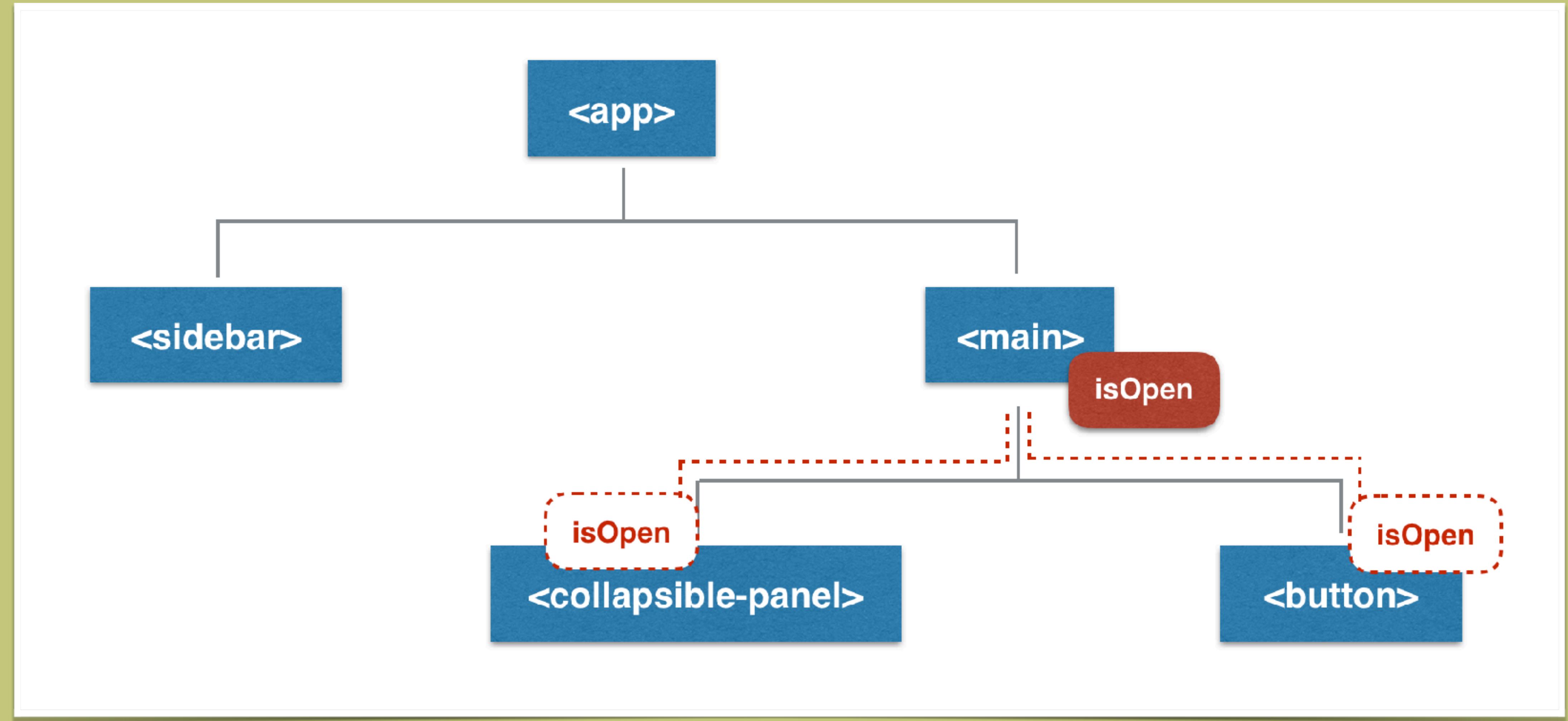
Never miss a story from **EmberMap**, when you sign up for Medium. [Learn more](#)

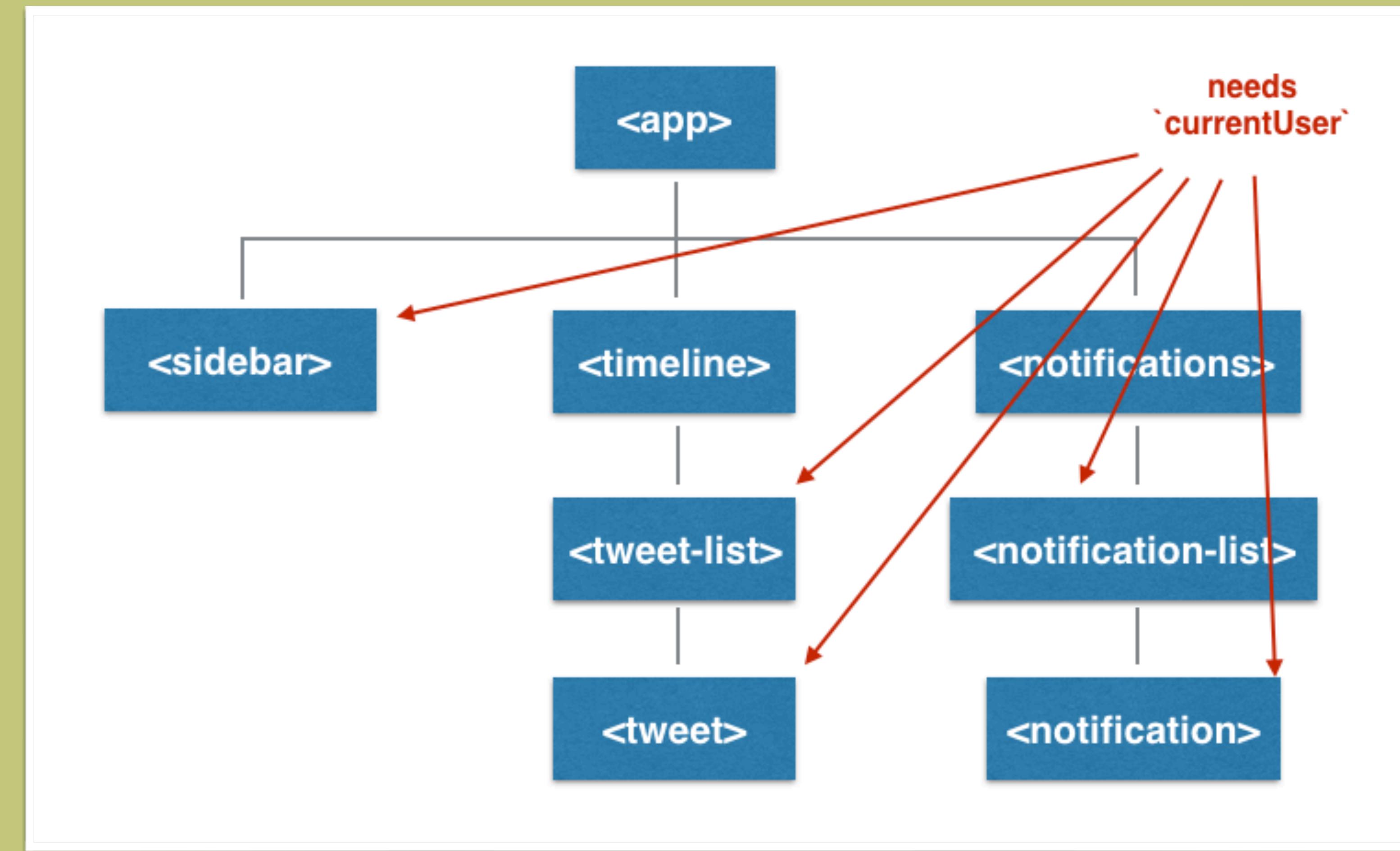
GET UPDATES

### Panel Title

Anim pariatur cliche reprehenderit, enim eiusmod high life accusamus terry richardson ad squid. 3 wolf moon officia aute, non cupidatat skateboard dolor brunch. Food truck quinoa nesciunt laborum eiusmod. Brunch 3 wolf moon tempor, sunt aliqua put a bird on it squid single-origin coffee nulla assumenda shoreditch et. Nihil anim keffiyeh helvetica, craft beer labore wes anderson cred nesciunt sapiente ea proident. Ad vegan excepteur butcher vice lomo. Leggings occaecat craft beer farm-to-table, raw denim aesthetic synth nesciunt you probably haven't heard of them accusamus labore sustainable VHS.

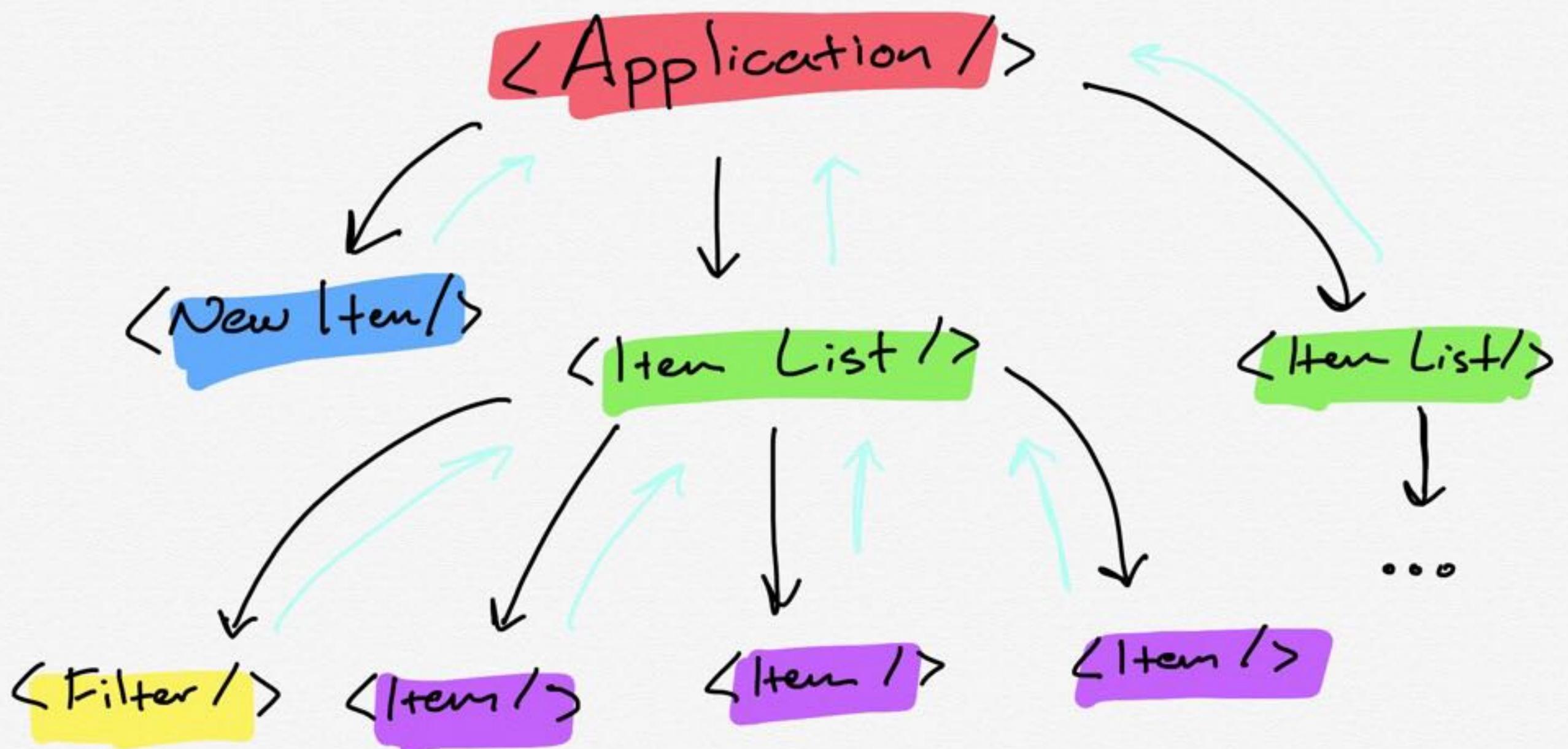


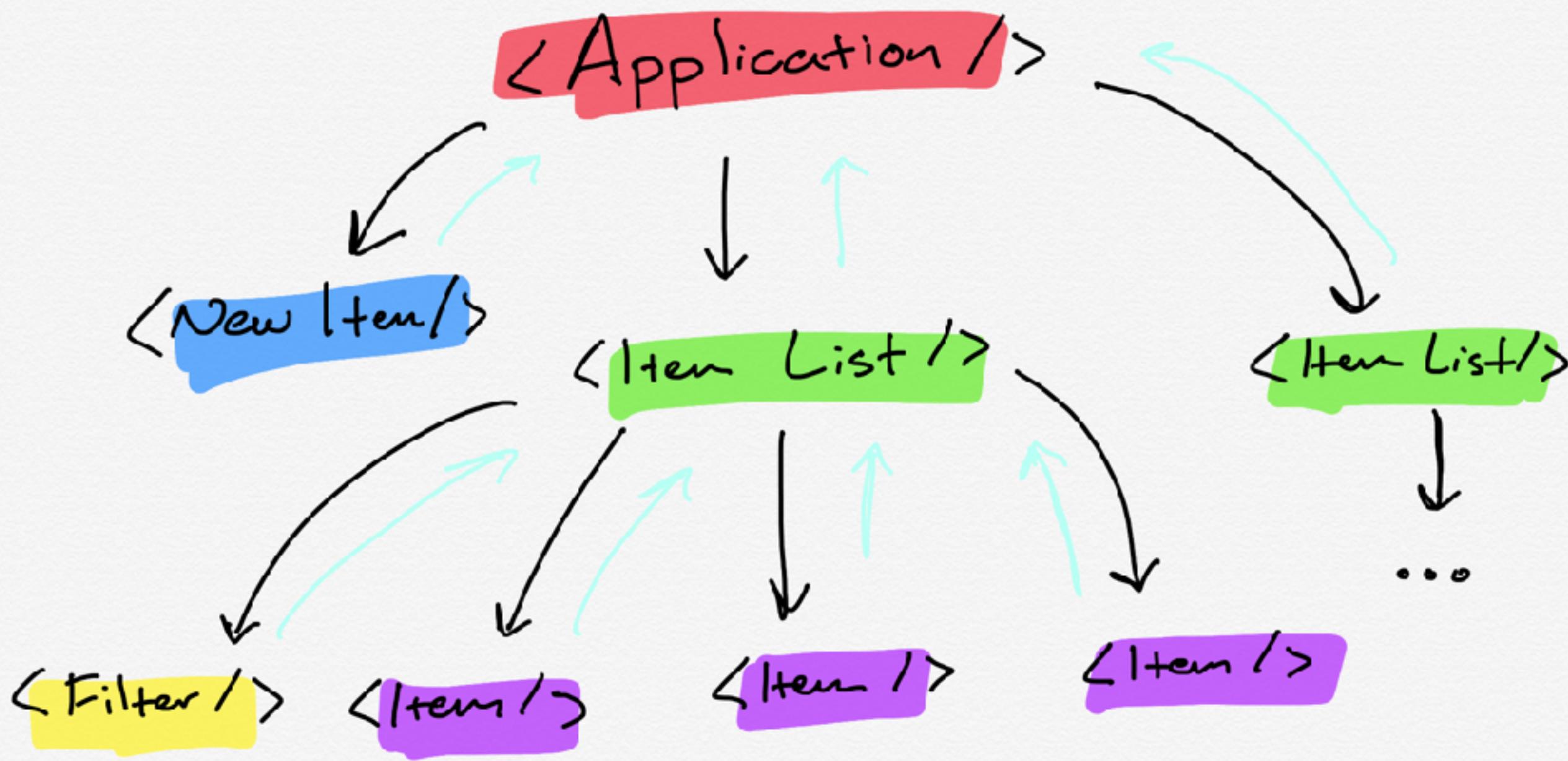




**We have some hard  
problems that need solving...**

**Deep component trees.**





What if this application  
grew and there  
were even  
more intermediary steps?

**Welcome to the Lightning  
Round**

Here's the deal...

# We're going to look at three different patterns.

- I'm going to talk about it.
- I'll do it with an absurdly simple example.
- You'll do with the a slightly-less simple example.
- We'll do it together.
- We'll revisit and discuss these patterns as we get deeper in using Flux, Redux, and MobX.

# **Lifting State with the Container Pattern**

Draw a line between state  
and presentation.

Container components manage state and pass it to presentational components.

Presentational components  
receive props and render UI.

They also receive actions and pass them back to the container.

They either only have a render()  
method or they are stateless  
functional components.

```
const Counter = ({ count, onIncrement }) => {
  return (
    <section>
      <h1>Count: {count}</h1>
      <button onClick={onIncrement}>Increment</button>
    </section>
  );
};
```

```
import React, { Component } from 'react';
import Counter from './Counter';

export default class CounterContainer extends Component {
  state = { count: 0 }
  increment = () => {
    this.setState(state => ({ count: state.count + 1 }));
  }

  render() {
    const { count } = this.state;
    return (
      <div>
        <Counter count={count} onIncrement={this.increment} />
        <Counter count={count} onIncrement={this.increment} />
      </div>
    );
  }
}
```

# Live Coding

# Exercise

- <https://github.com/stevekinney/pizza-calculator>
- You joke, but this is an application that got a lot of use at my old gig.
- Right now, the state and the UI are jammed together.
- Can you separate them out?
- <PizzaCalculator /> and <PizzaCalculatorContainer />

# **Higher Order Components**

A container factory.

“Hey, I’m going to need this state all over the place—and I don’t want to pass it around.”

```
const WithCount = WrappedComponent => class extends Component {
  state = { count: 0 };
  increment = () => {
    this.setState(state => ({ count: state.count + 1 }));
  };

  render() {
    return (
      <WrappedComponent
        count={this.state.count}
        onIncrement={this.increment}
        {...this.props}
      />
    );
  }
};
```

```
const CounterWithCount = WithCount(Counter);
```

# Live Coding

# Exercise

- So, I'm ordering pizzas for two different events apparently and I can't be bothered to open two tabs.
- Can you use the Higher Order Component pattern to create two separately managed calculators?

I love the Higher Order  
Component pattern, but...

Sometimes, it's hard to see  
what's going on inside.

I also don't want to make two of every component: a presentational and a presentation wrapped in a container.

# **Render Properties**

```
export default class WithCount extends Component {
  state = { count: 0 };
  increment = () => {
    this.setState(state => ({ count: state.count + 1 }));
  };

  render() {
    return (
      <div className="WithCount">
        {
          this.props.render(
            this.state.count,
            this.increment,
          )
        }
      </div>
    );
  }
}
```

```
<WithCount render={  
  (count, increment) => (  
    <Counter count={count} onIncrement={increment} />  
  )  
}  
>
```

# Live Coding

# Exercise

- I don't have a good story for why I'm going to make you do this.
- In fairness, my previous excuses weren't that great either.
- So, just go ahead and refactor those higher order components to use render props instead?

# Part Four

## The Flux Pattern

● ● ● / G flux p × Back × Twitter, Inc. [US] https://twitter.com/ryanflorence/status/861641842038681600 Steve

Ryan Florence .@ryanflorence May 8  
We totally screwed this up with React Router pre v4 too, we hosed the component model! Not mad, no accusation just hindsight and reflection.

Ryan Florence .@ryanflorence Following

Flux, alt, redux, are solid ways to build an app. I just wonder what hidden trade offs we've made as a community going whole hog into them.

11:59 AM - 8 May 2017

2 Retweets 22 Likes

Ryan Florence .@ryanflorence Co-Author Bear

Tweet your reply

# We're going to try this one backwards.

- I'm going to implement the Flux pattern in a simple example.
- Then I'll go into all of the conceptual mumbo-jumbo once we've seen it in action.
- Then we'll implement it again in **Jetsetter**.

# Live Coding

flux NPM, Inc. [US] https://www.npmjs.com/package/flux Steve

Narnia's Psychedelic Mushrooms npm Enterprise features pricing documentation support

npm find packages Greetings, stevekinney

Loosely couple your services. Use Orgs to version and reuse your code. [Create a free org »](#)

★ **flux** public

An application architecture for React utilizing a unidirectional data flow.

```
graph TD; UI[User Interactions] --> RV[React Views]; RV --> AC[Action Creators]; AC --> D[Dispatcher]; D --> S[Store]; S --> RV; S --> CE[Change Events + Store Queries]; AC --> WebAPI[Web API]; AC --> WebAPIUtils[Web API Utils]; WebAPIUtils --> WebAPI; WebAPI --> AC; WebAPI --> D; WebAPIUtils --> D; WebAPIUtils --> CE; D --> C[Callbacks]; C --> S; S --> C;
```

[npm install flux](#)  
how? learn more

[kyldvs](#) published 4 months ago  
3.1.3 is the latest of 13 releases  
[github.com/facebook/flux](#)  
[facebook.github.io/flux](#)  
BSD-3-Clause

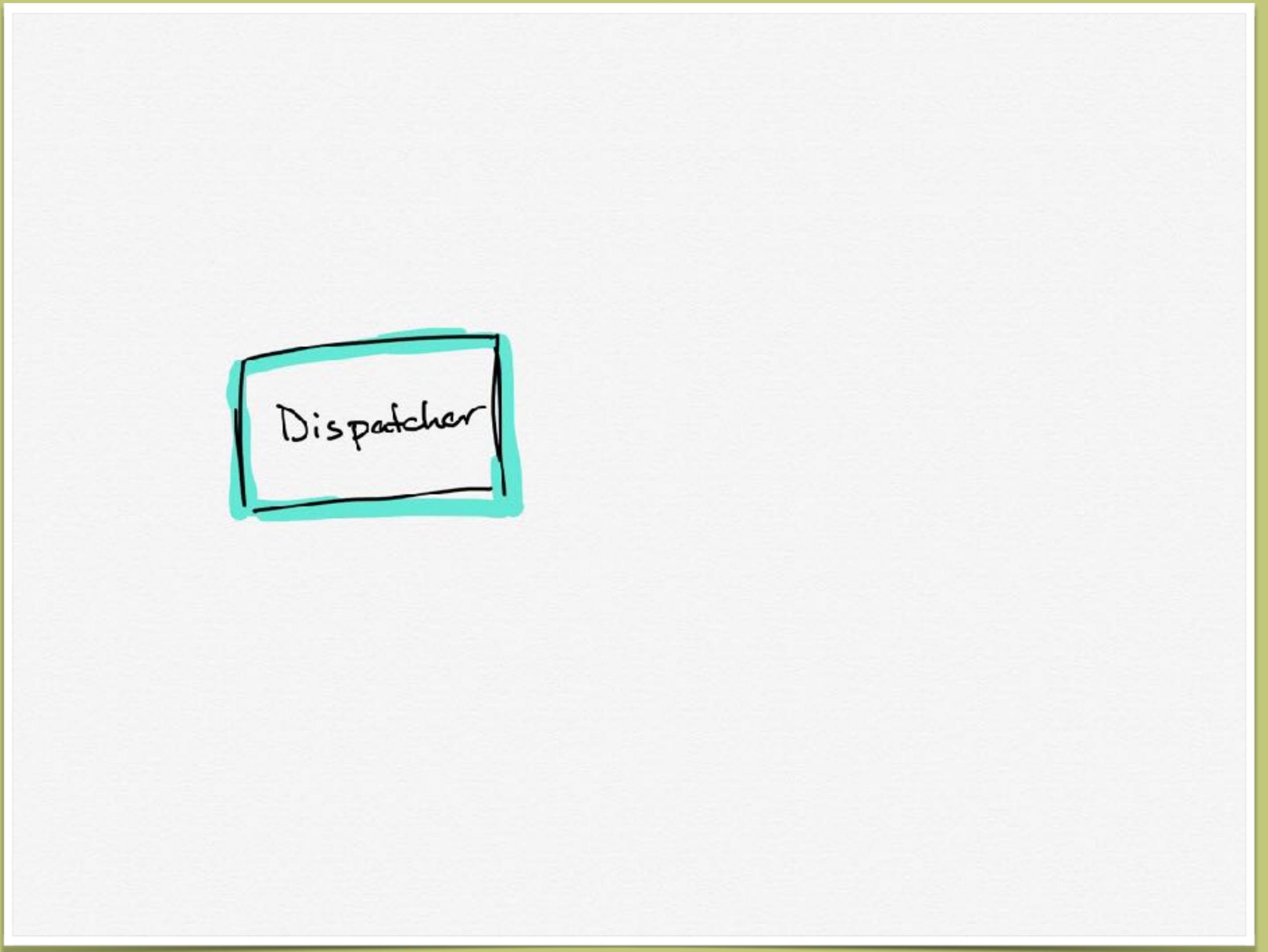
Collaborators list

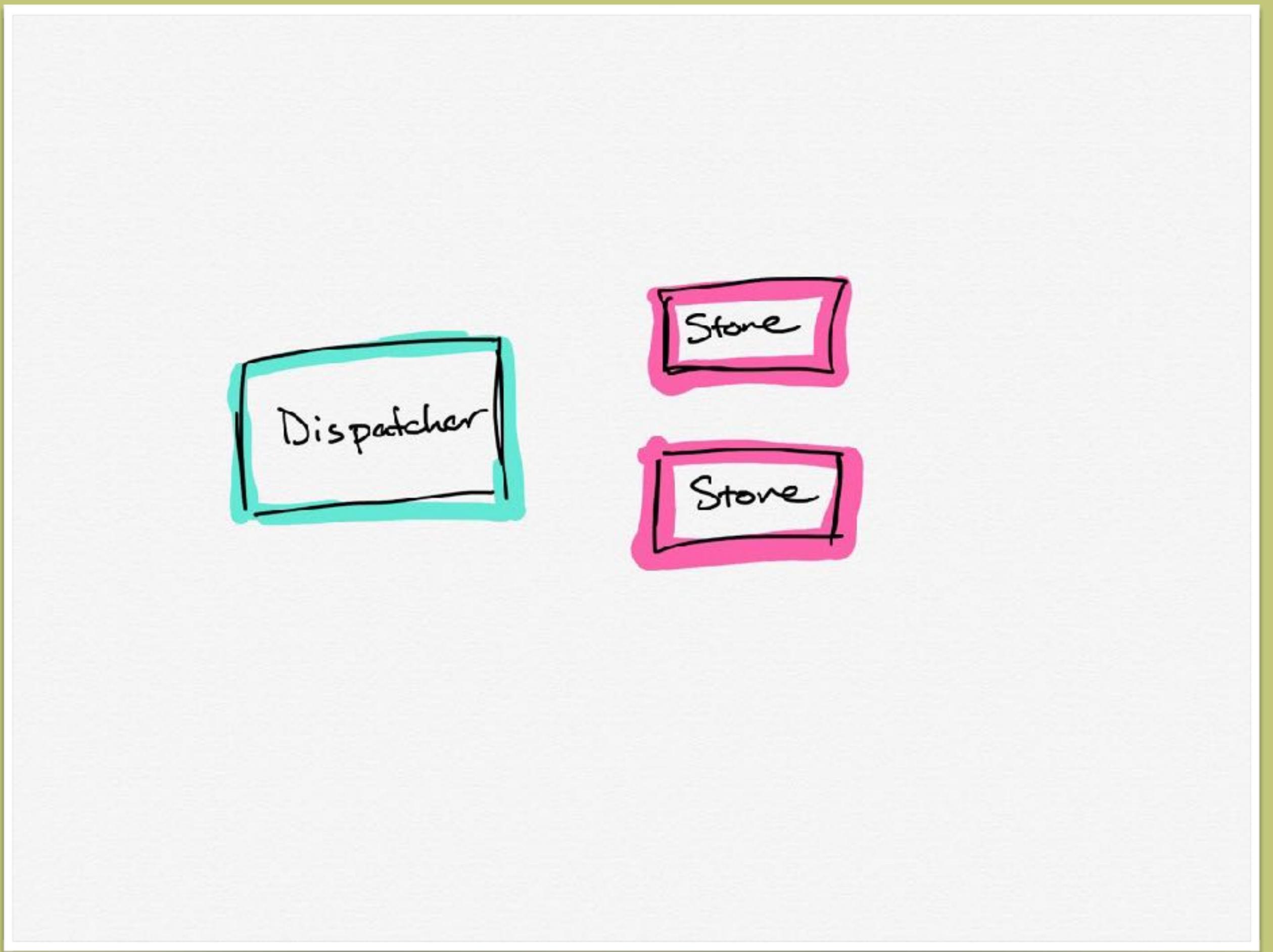
Stats  
13,210 downloads in the last day  
61,392 downloads in the last week

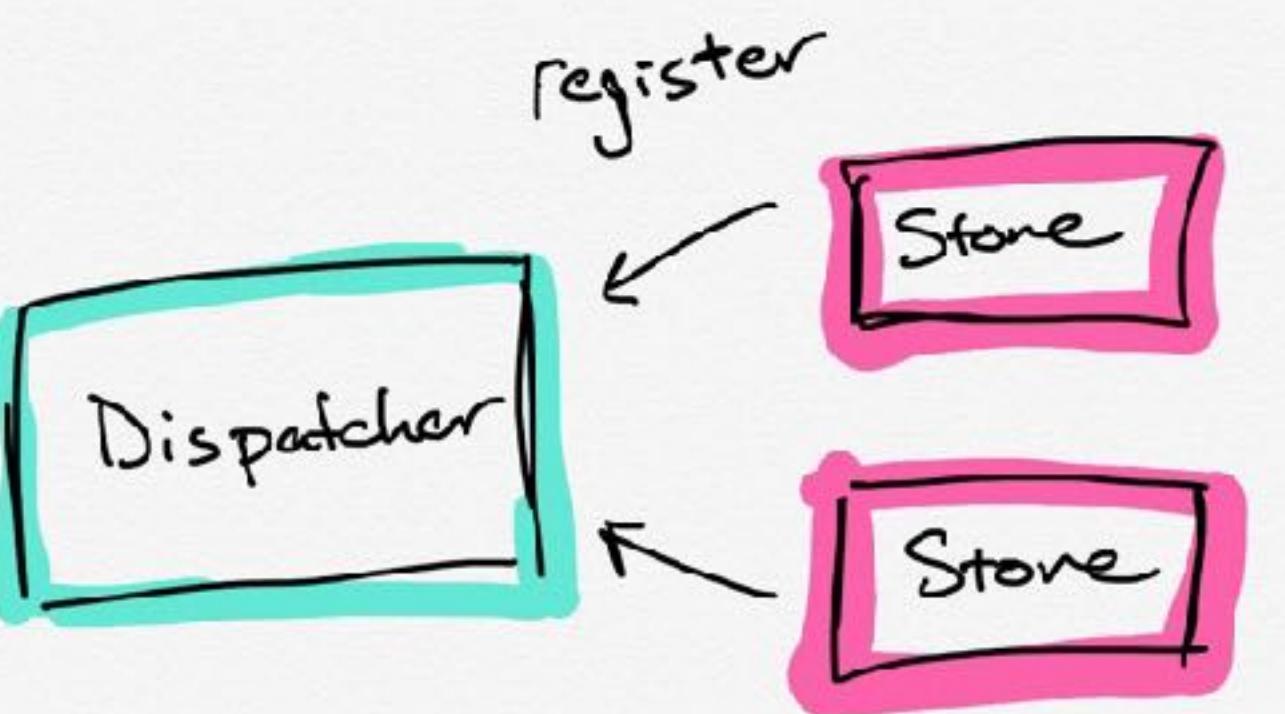
**Getting Started**

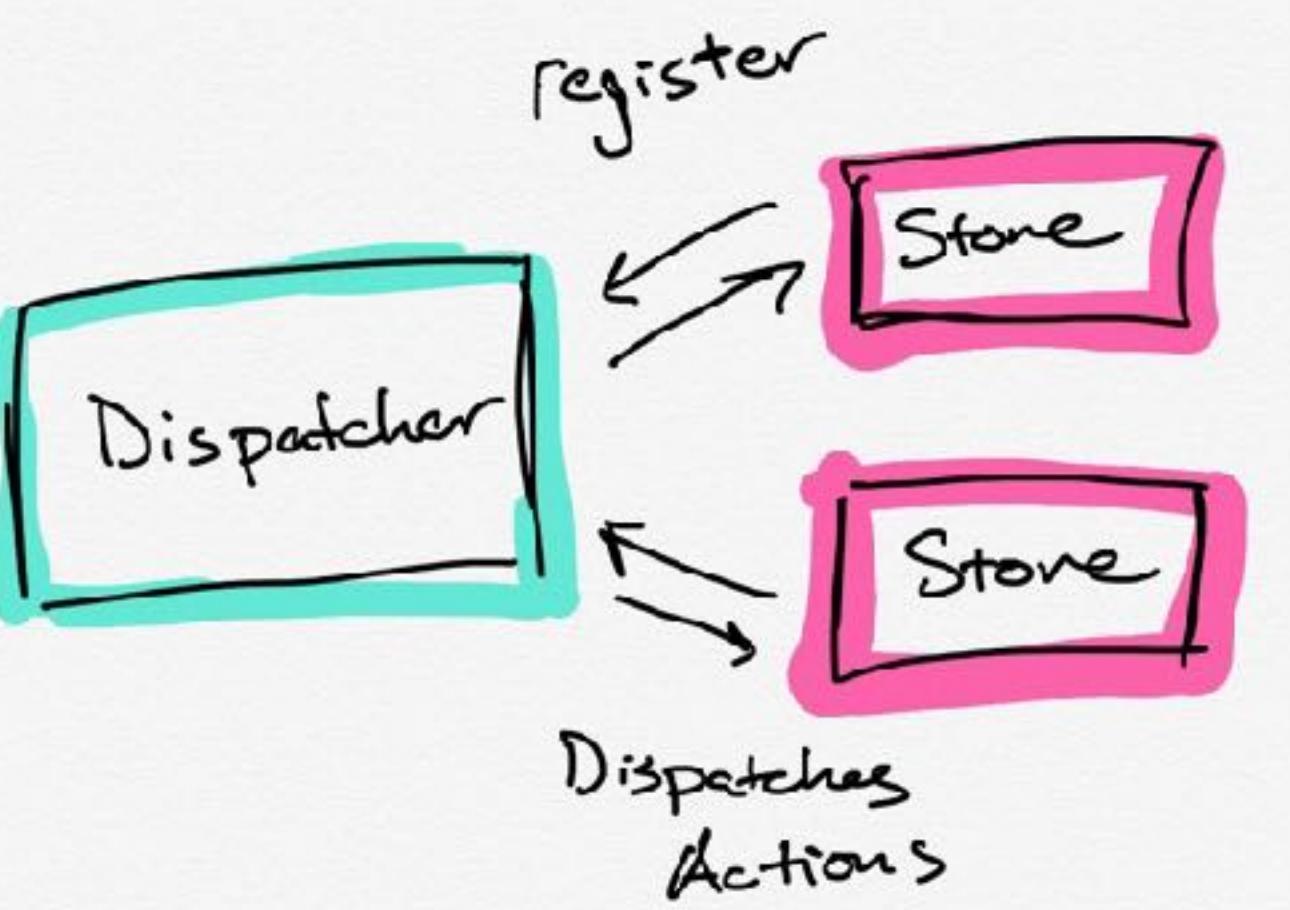
Start by looking through the [guides and examples](#) on Github. For more resources and API docs check

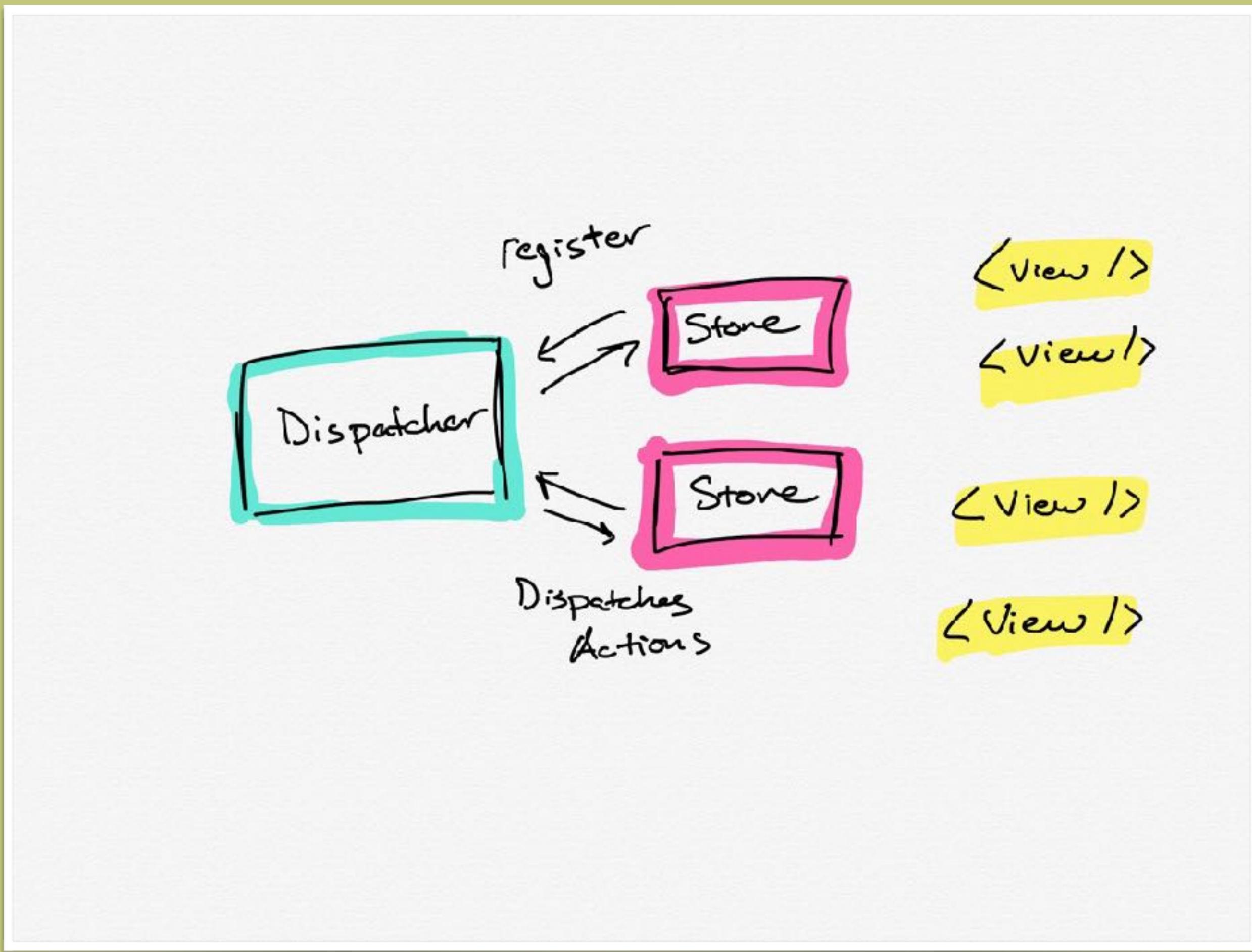
**And now: A very scientific  
chart.**

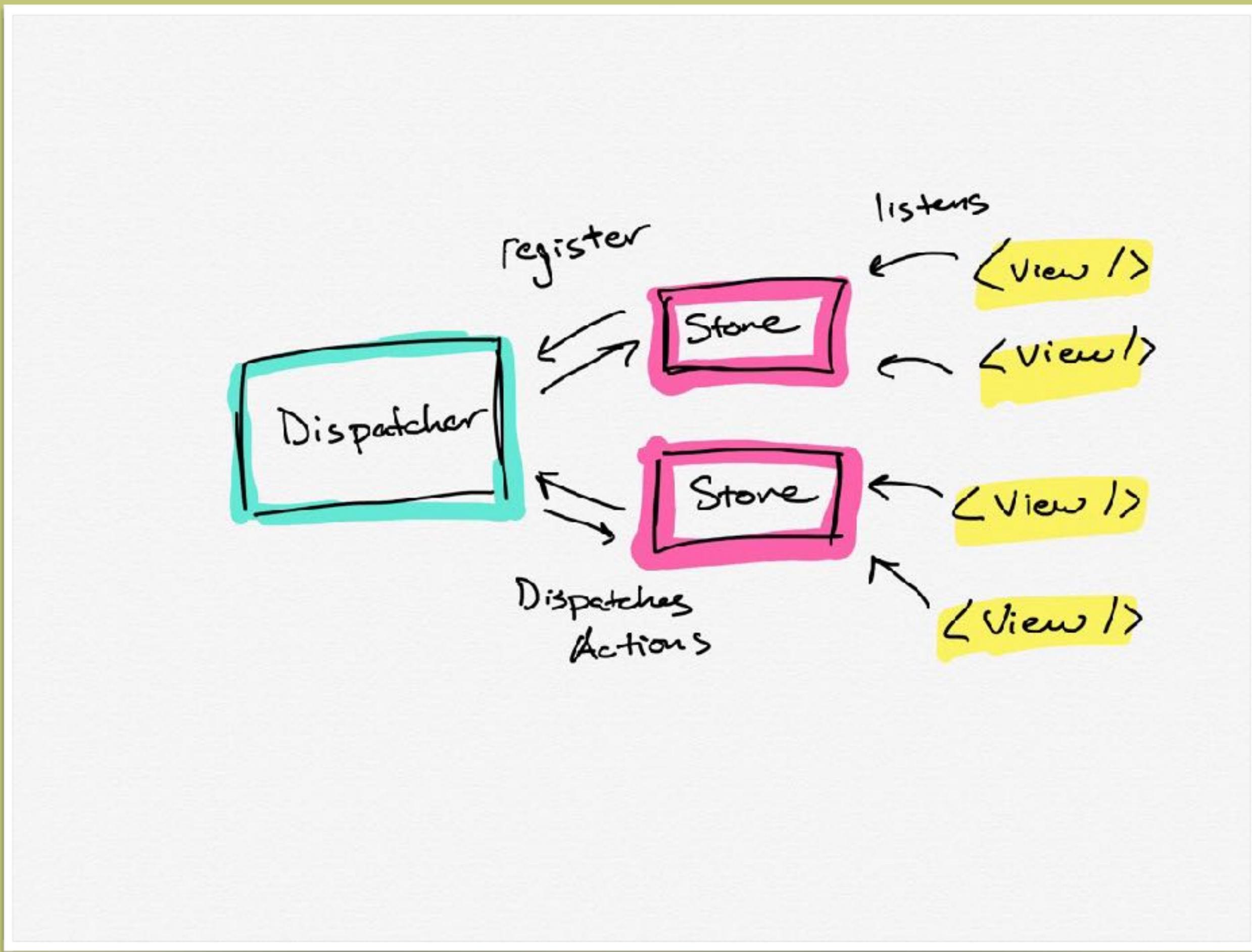


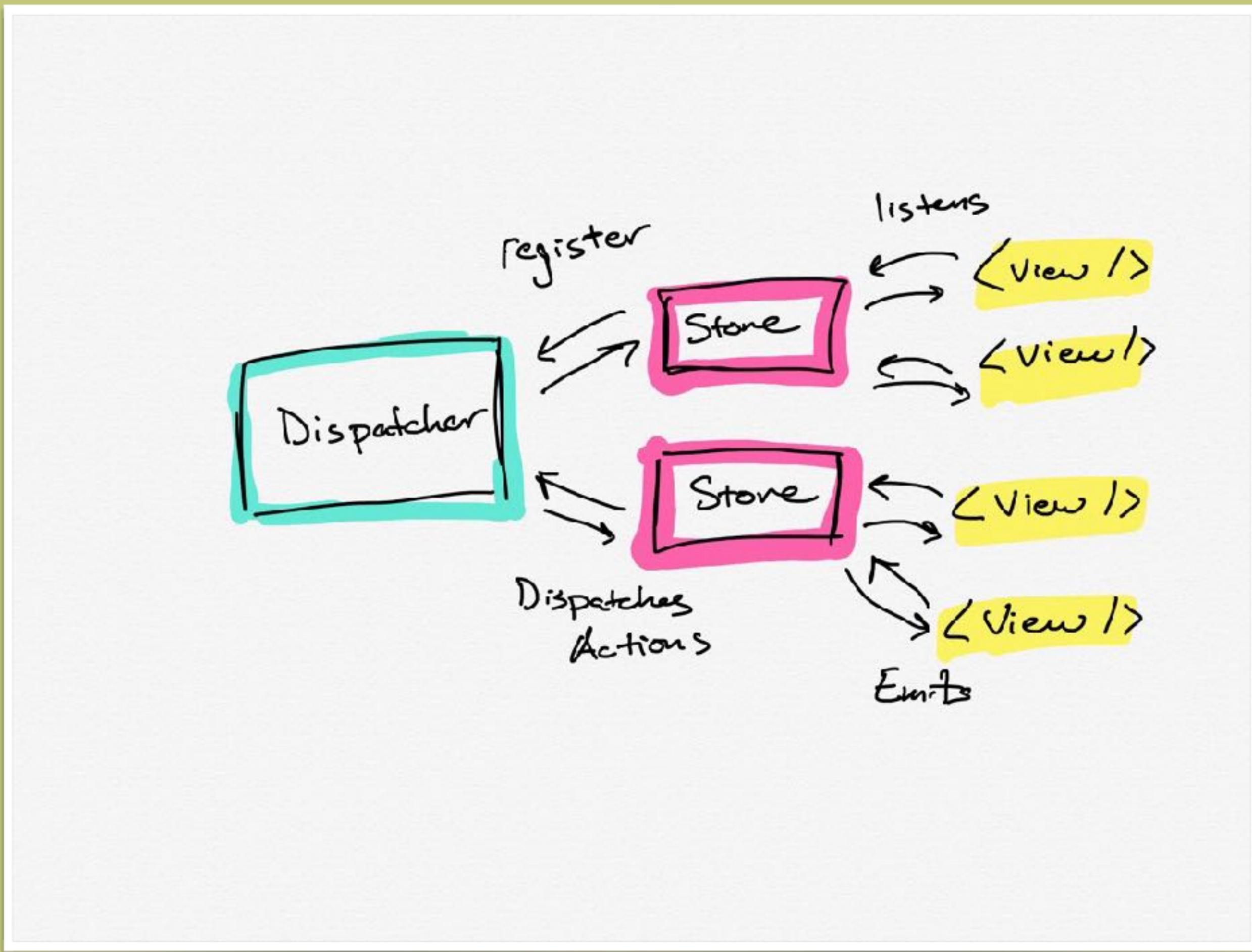


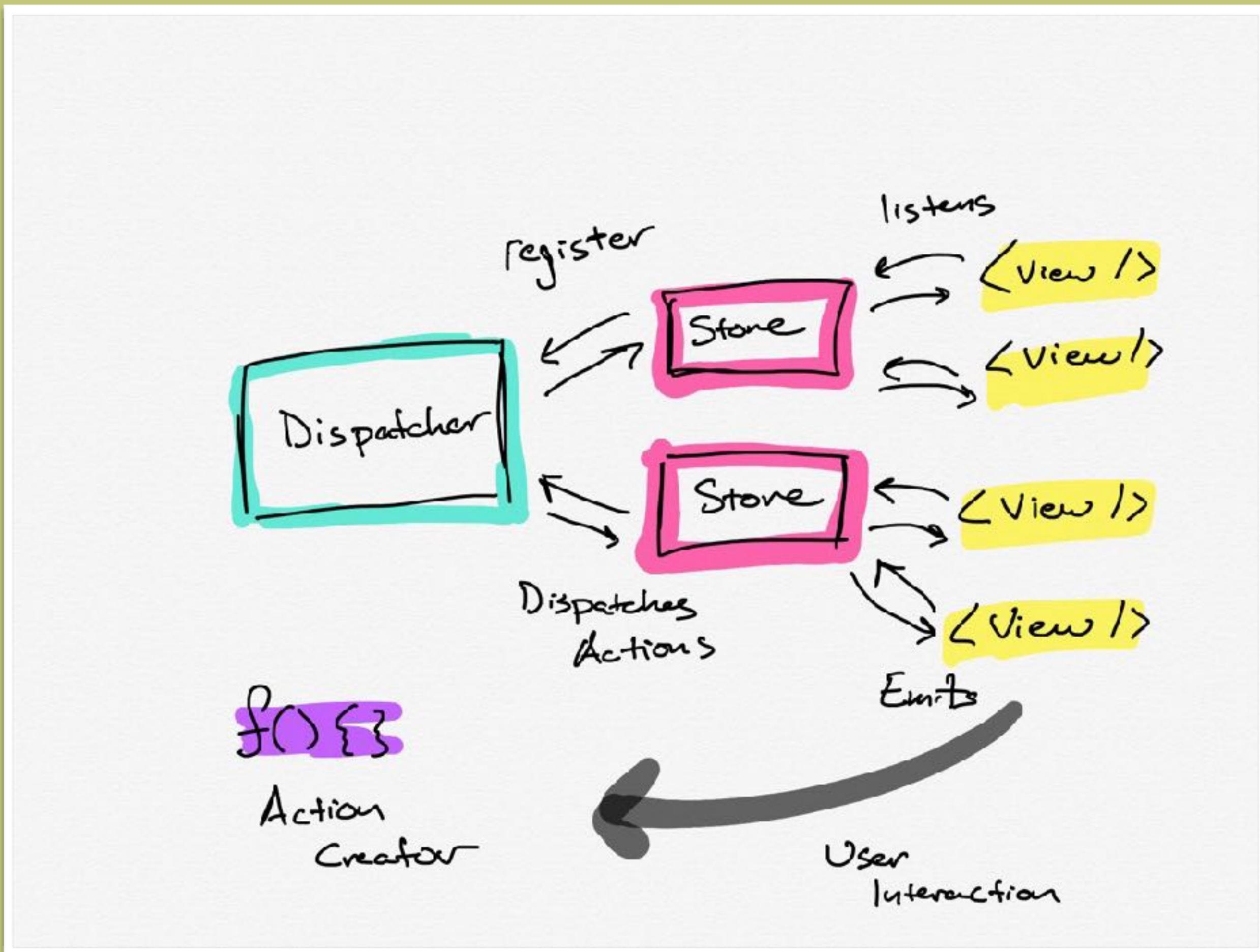


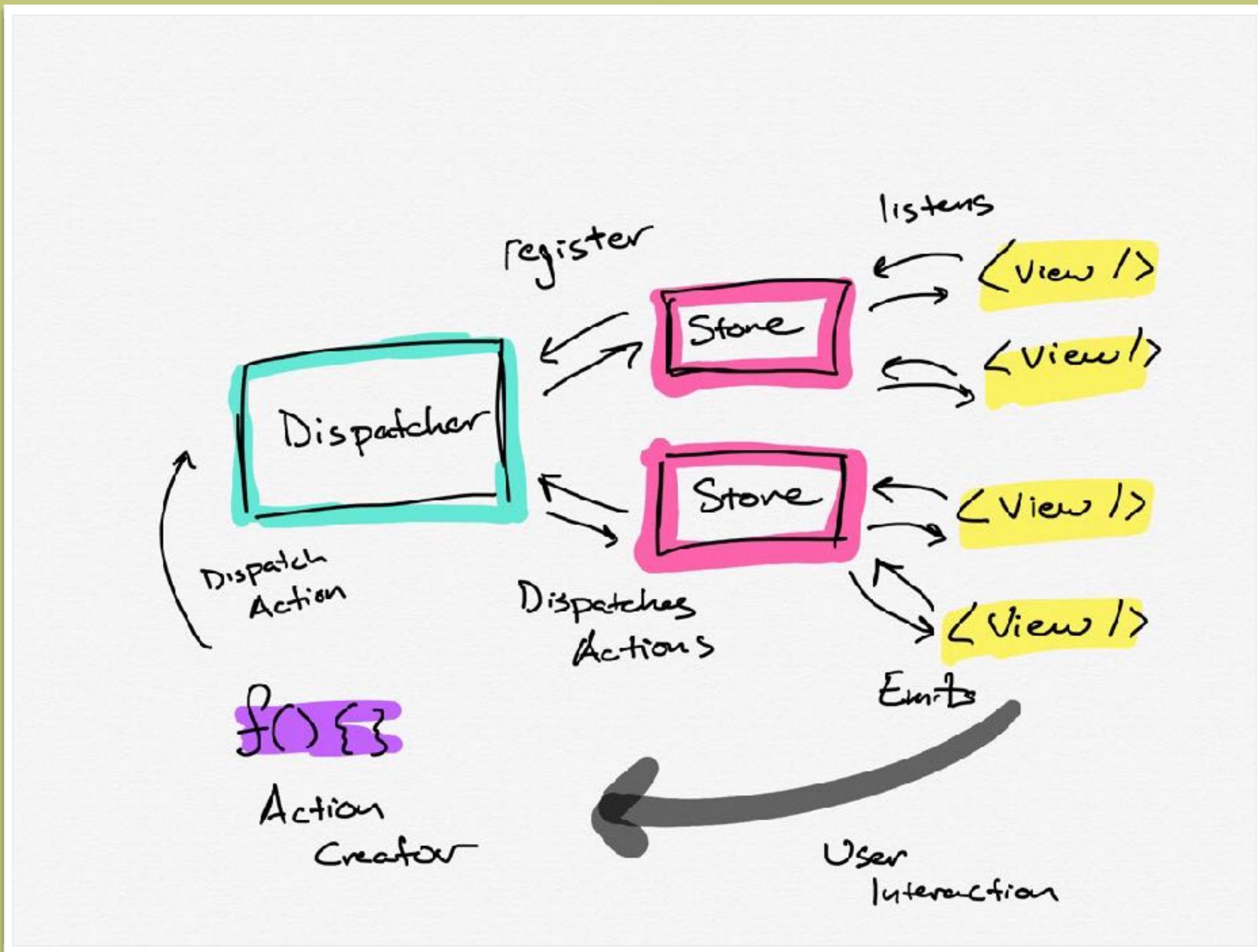


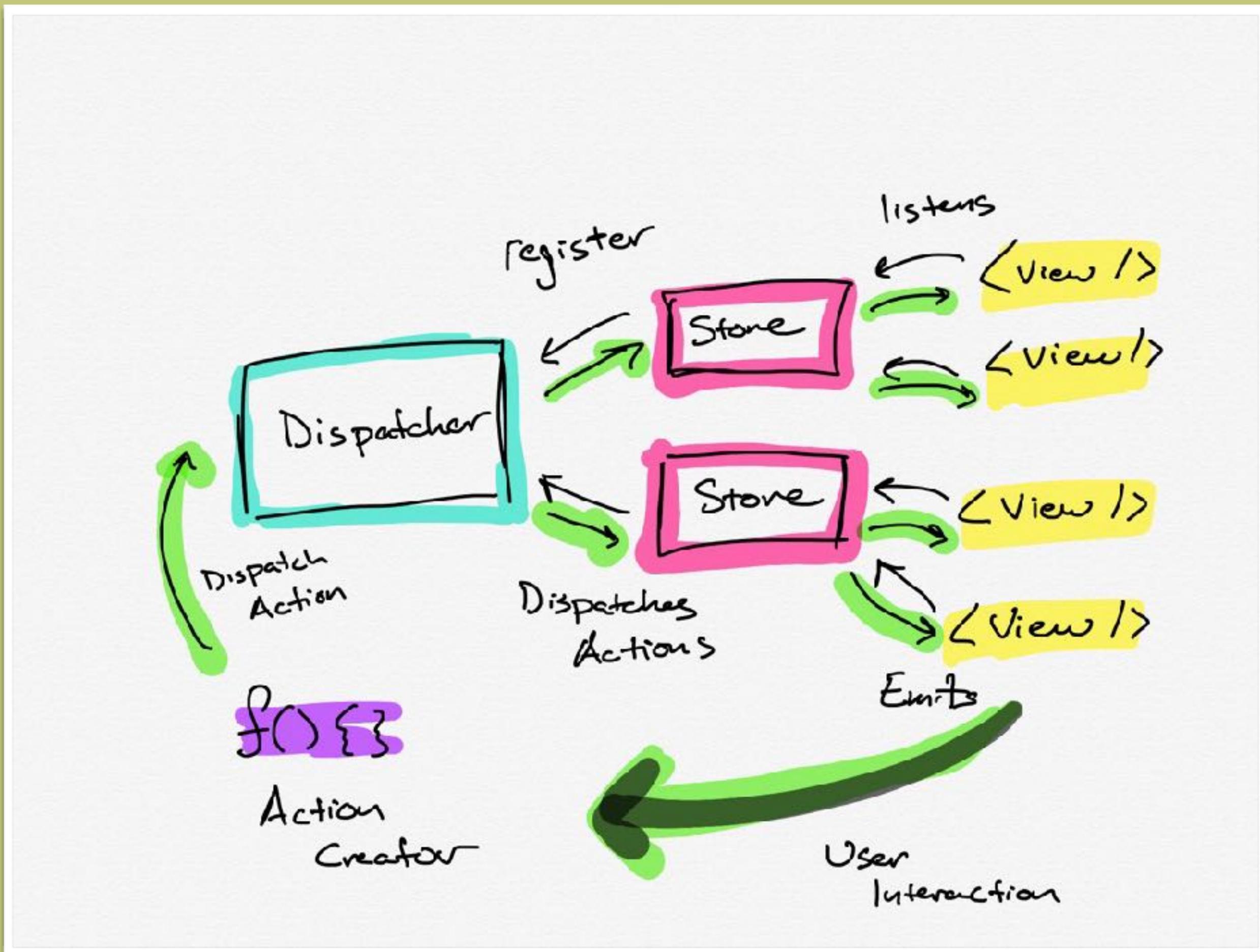












```
const AppDispatcher = new Dispatcher();
```

```
export const addItem = value => {
  AppDispatcher.dispatch({
    type: 'ADD_NEW_ITEM',
    item: {
      id: uniqueId(),
      packed: false,
      value,
    },
  });
};
```

```
class ItemStore extends EventEmitter {  
  constructor() {  
    super();  
    AppDispatcher.register(action => {  
      if (action.type === 'ADD_NEW_ITEM') { ... }  
      if (action.type === 'UPDATE_ITEM') { ... }  
      if (action.type === 'REMOVE_ITEM') { ... }  
    });  
  }  
}
```

The store contains your  
models.

It registers itself with the dispatcher and receives actions.

***action* (noun):** The minimal amount of information necessary to represent the change that should occur.

Actions are the *only* way to initiate a change to the state inside a store.

{ }

```
{ type: 'INCREMENT' }
```

*Actions can contain additional information, but they don't need to.*

{

    type: 'INCREMENT',  
    amount: 5

}

Whenever it does something based  
on an action, it emits a change  
event.

```
let items = [];  
  
class ItemStore extends EventEmitter {  
  constructor() {  
    super();  
  
    AppDispatcher.register(action => {  
      // ...  
    });  
  }  
  
  getItems() {  
    return items;  
  }  
  
  addItem(item) {  
    items = [...items, item];  
    this.emit('change');  
  }  
}
```

Your views listen for these  
change events.

```
class Application extends Component {
  state = {
    items: ItemStore.getItems(),
  };

  updateItems = () => {
    const items = ItemStore.getItems();
    this.setState({ items });
  };

  componentDidMount() {
    ItemStore.on('change', this.updateItems);
  }

  componentWillUnmount() {
    ItemStore.off('change', this.updateItems);
  }

  render() { ... }
}
```

Your views can trigger actions  
based on user interaction.

```
import { addItem } from './actions';

class NewItem extends Component {
  constructor() { ... }

  handleChange(event) { ... }

  handleSubmit(event) {
    const { value } = this.state;
    event.preventDefault();

    addItem(value); // Look here! It's an action creator!
    this.setState({ value: '' });
  }

  render() { ... }
}
```

Which then goes to the dispatcher  
and then to the store, which triggers  
another change—updating the view.

# Exercise

- I told you this was coming.
- It's your turn to implement the Flux pattern.
- This time in **Jetsetter**.
- Start on the flux-base branch.

# Live Coding



# Part Five

## Redux

# **What is Redux?**

Redux is considered to be an implementation of the Flux pattern.

We're going to start by explaining  
Redux outside of the context of  
React.

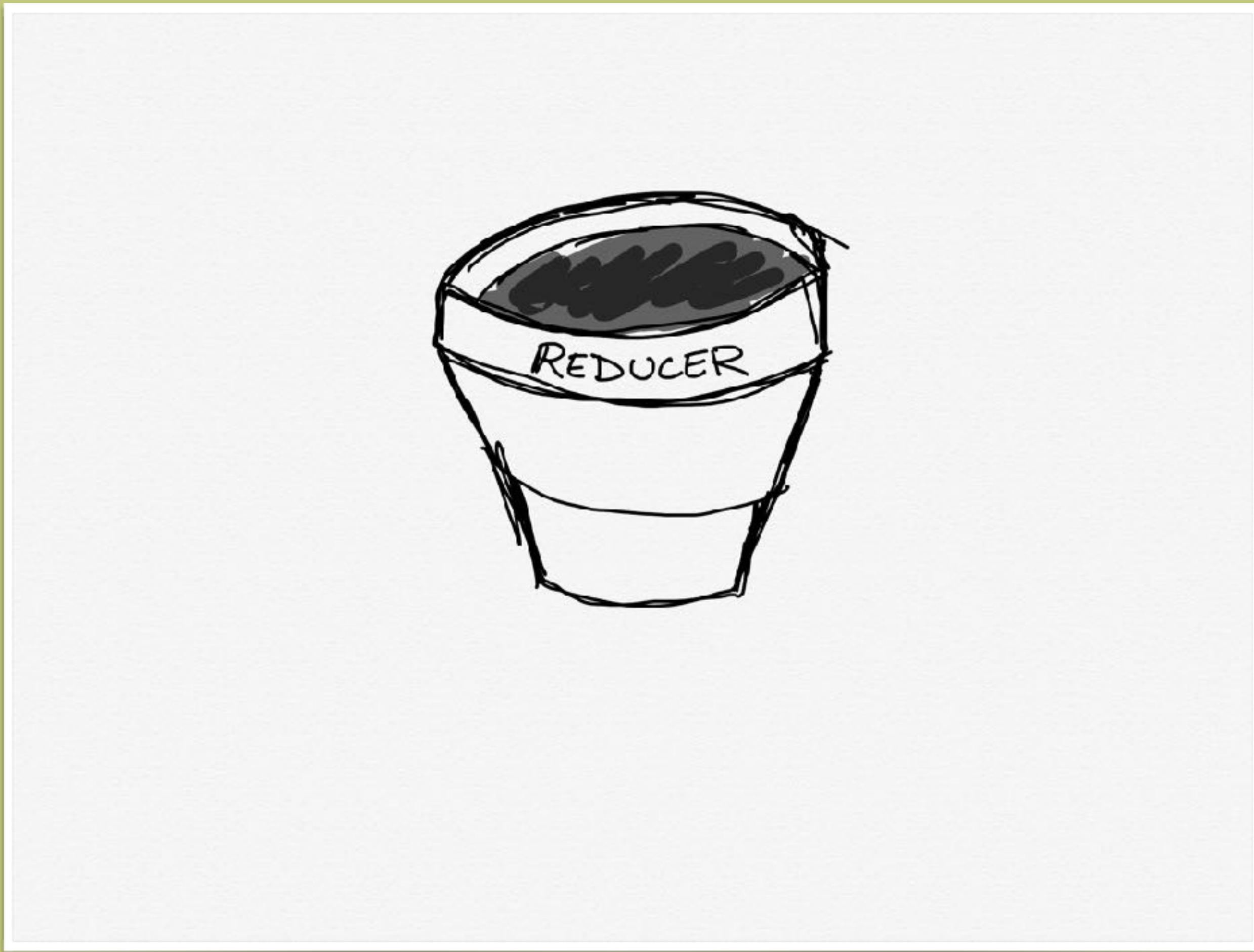
With Flux, you'd probably end up  
with a different store for every  
noun/model in your application.

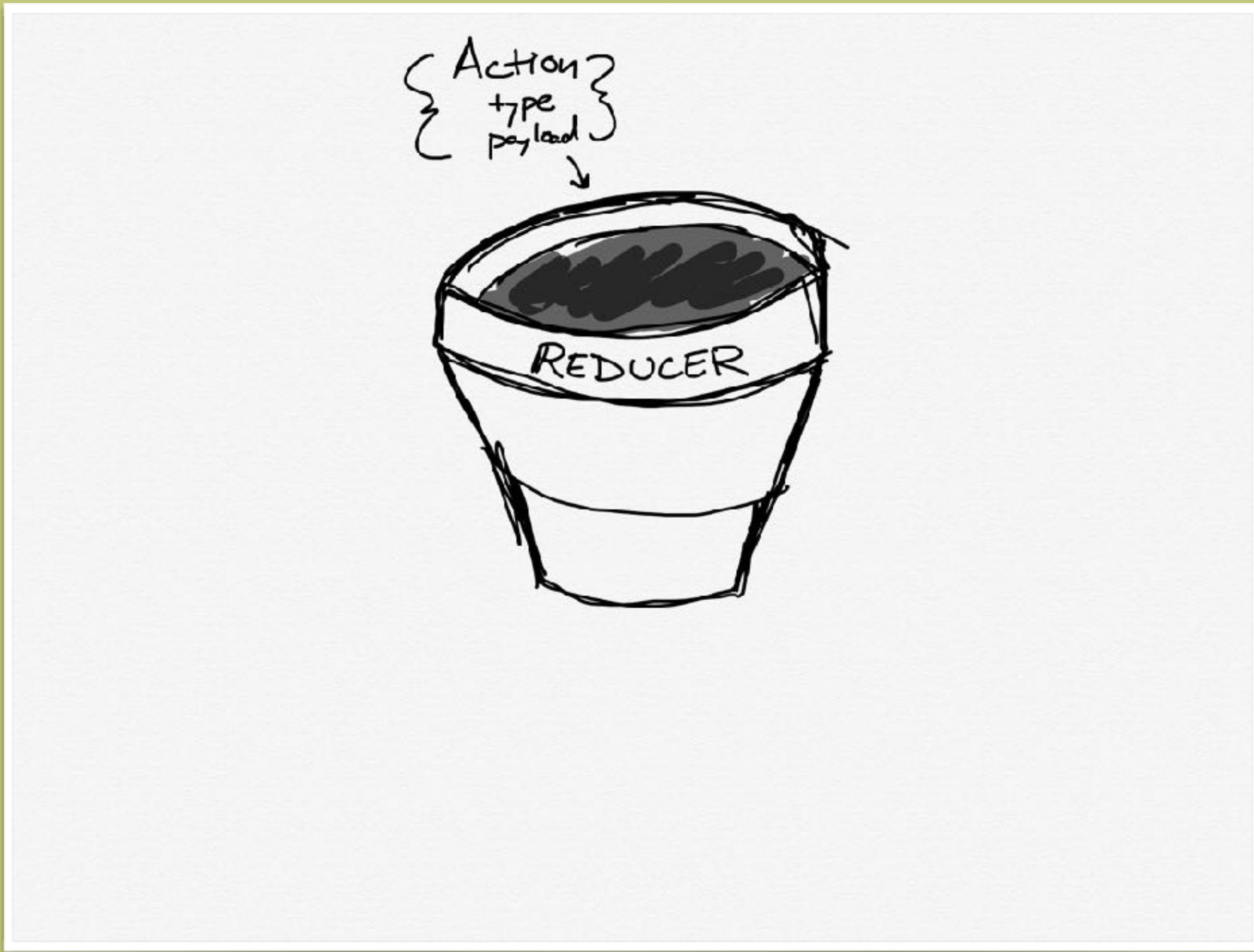
The whole state tree of your application is kept in one store.

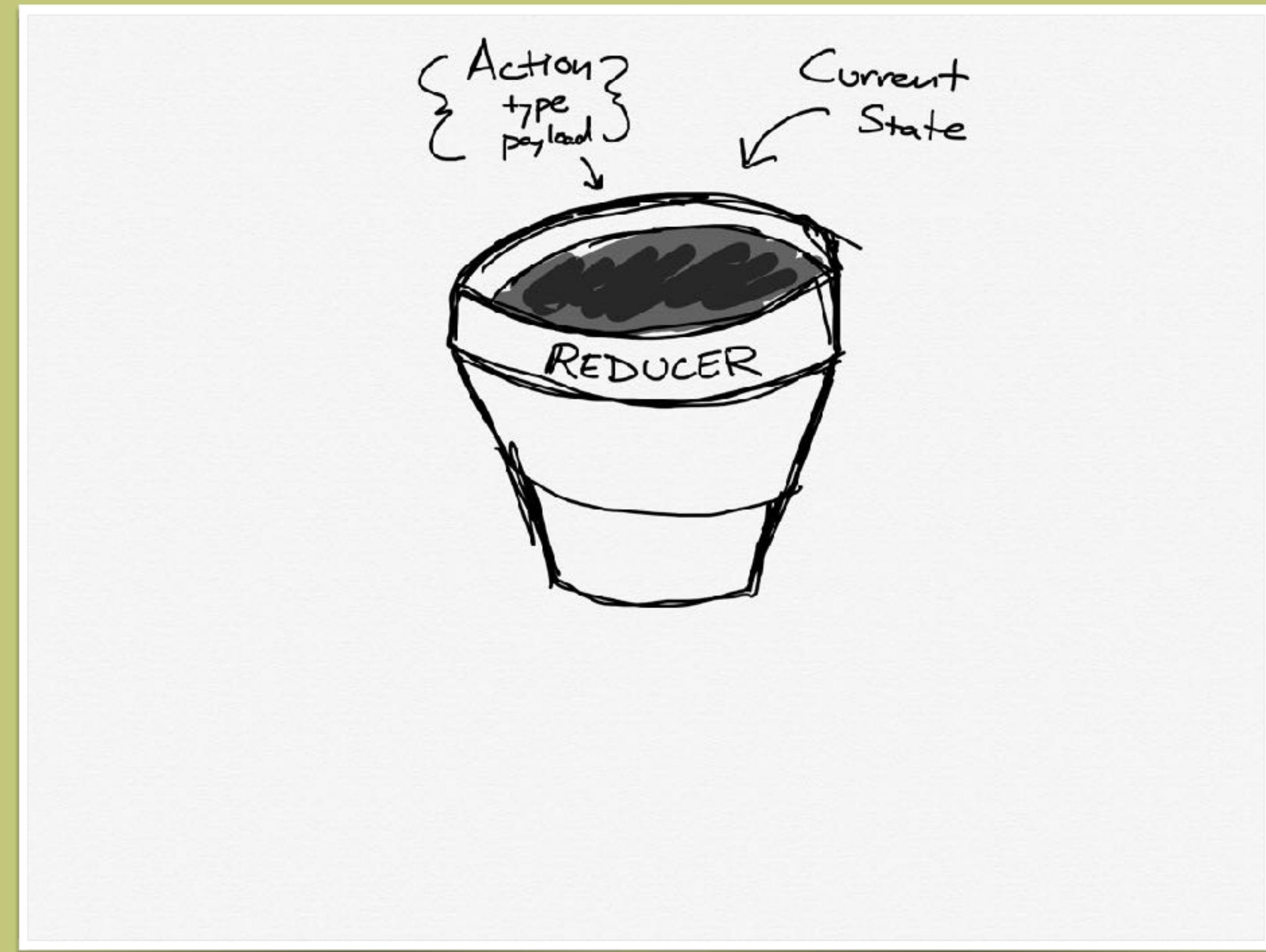
Just one plain old JavaScript  
object. 😊

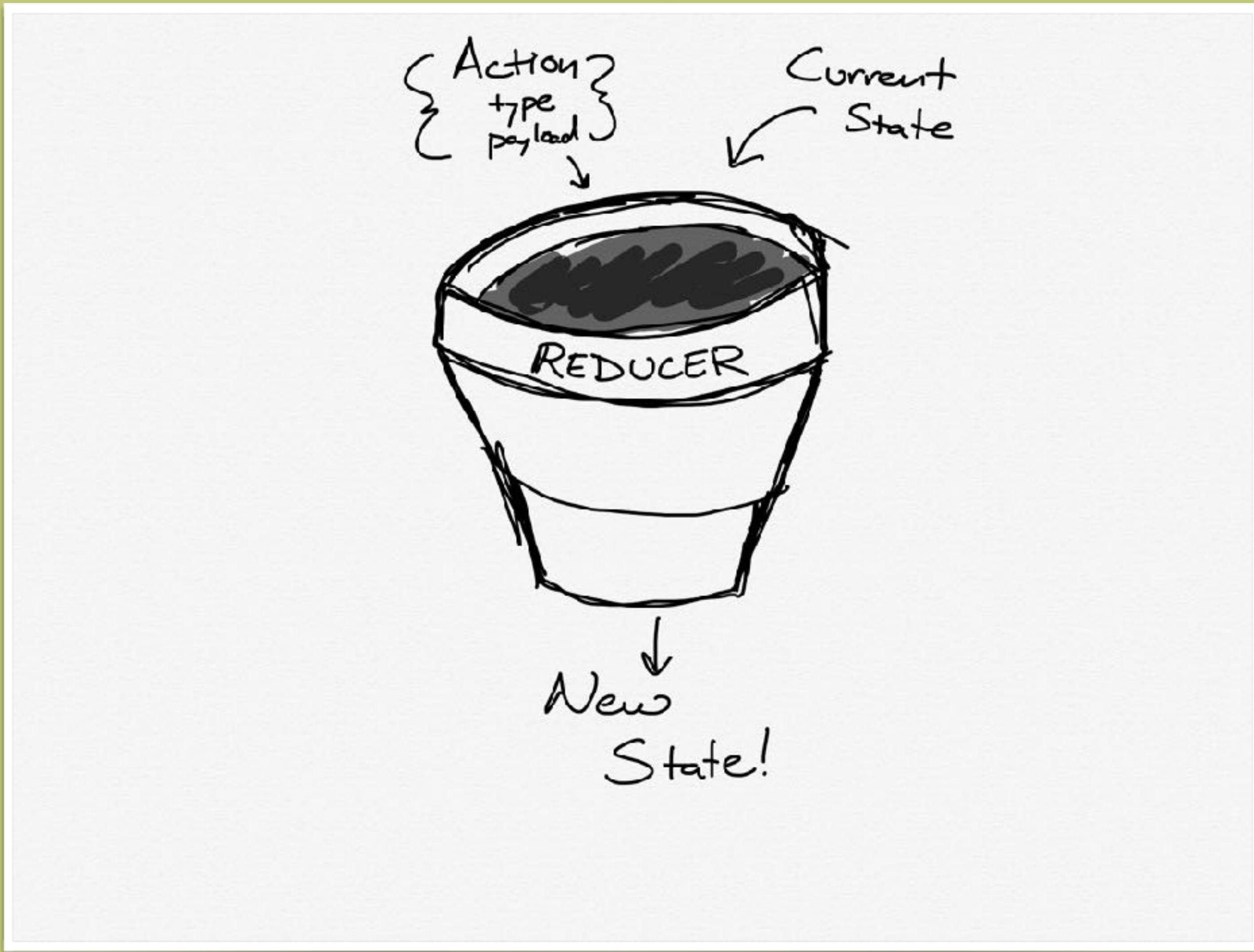
Like Flux, “one does not simply modify the state tree.”

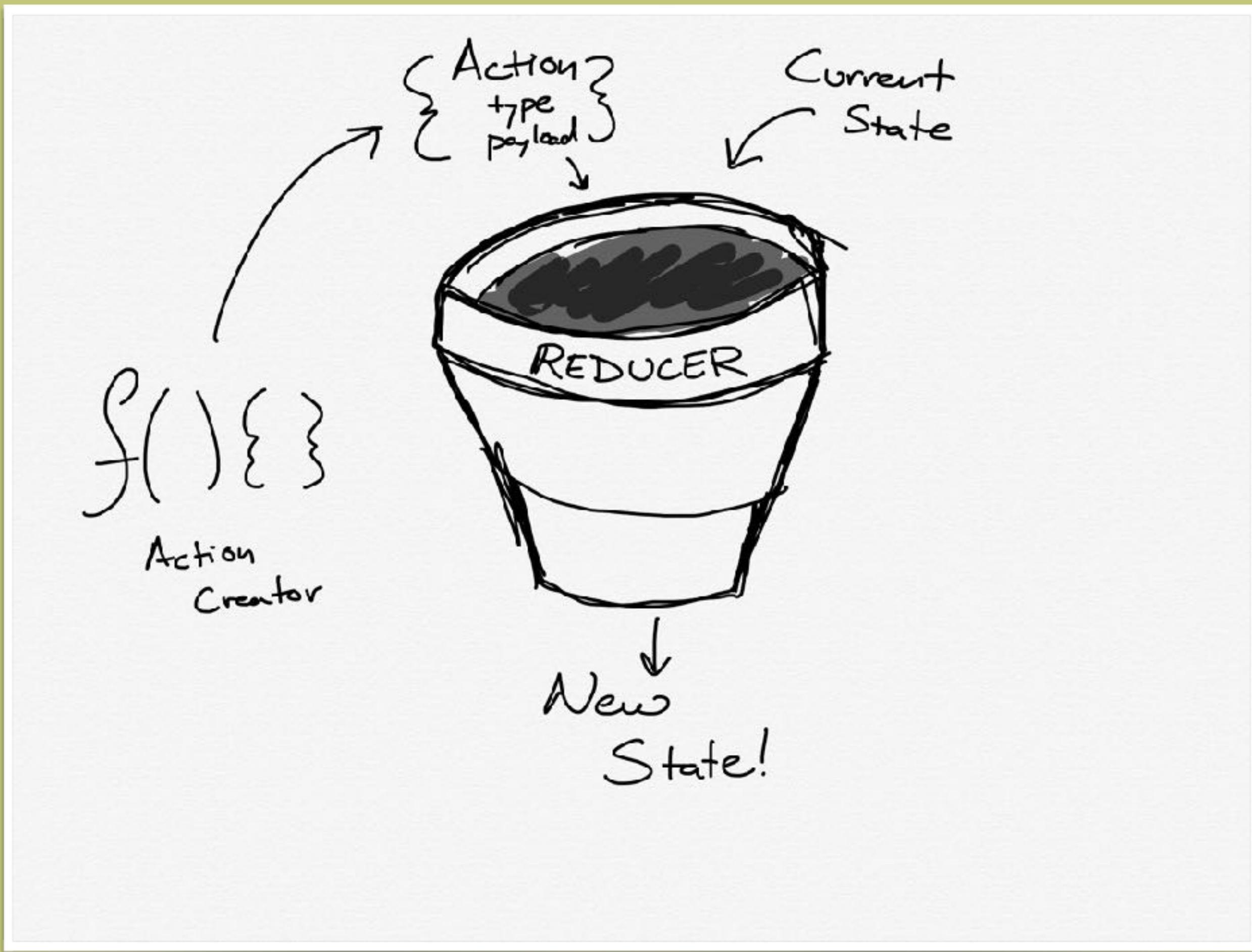
**And now: Another very  
scientific chart.**

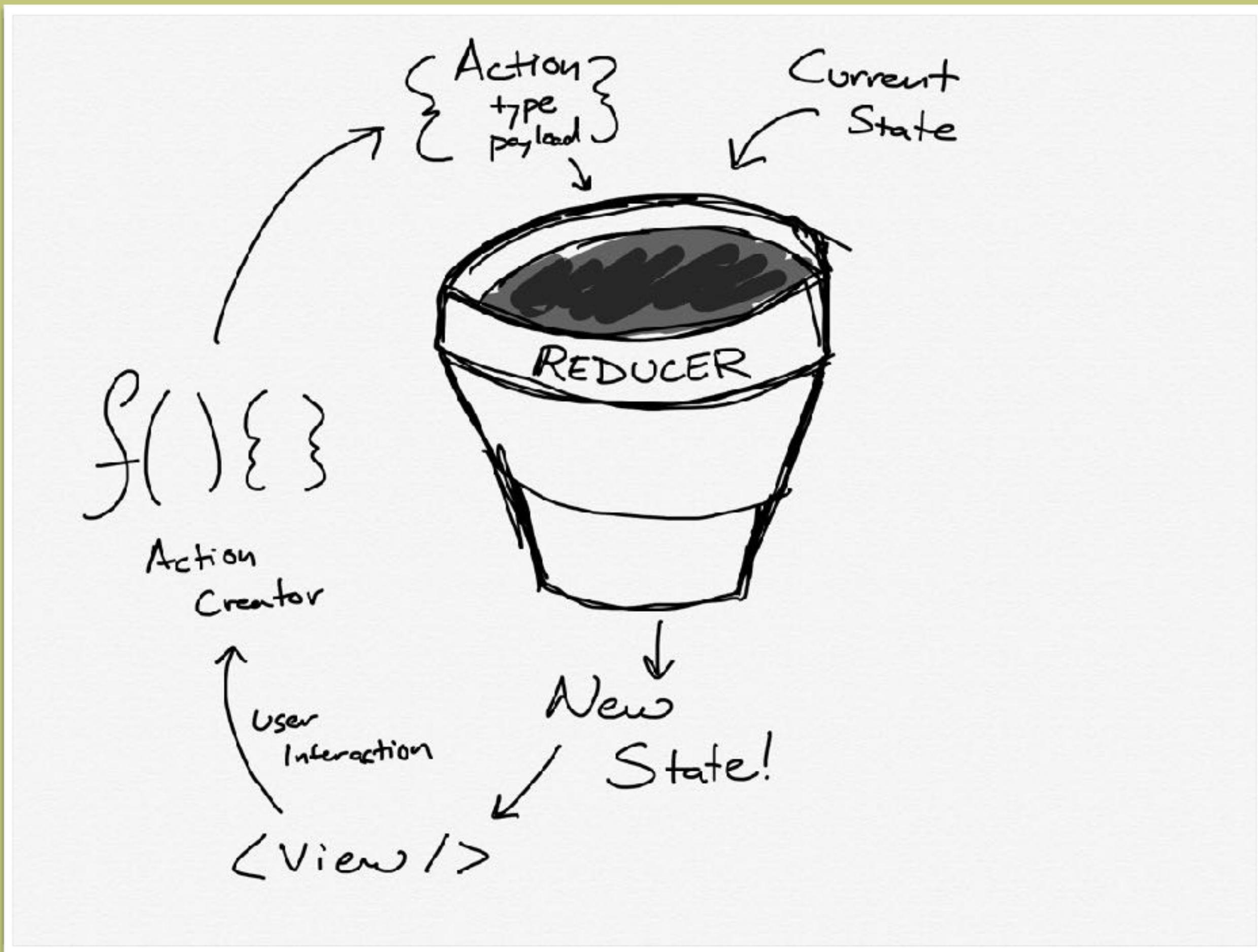












**Redux is *small*.**

```
applyMiddleware: function()  
bindActionCreators: function()  
combineReducers: function()  
compose: function()  
createStore: function()
```

<http://bit.ly/redux-codealong>

# Exercise

- On the redux-testing branch for **Jetsetter**, there are a number of tests ready to go.
- The problem is that there is no implementation.
- Can you implement the reducers and actions in order to make the tests pass?
- `npm test` or `yarn test` to run the tests.
- I'll get you started.

**Hey Steve, I use Flux at work, but I  
dig this reducer pattern! Can I use it?**

Yea! 😊

```
import { ReduceStore } from 'flux/utils';

class ItemStore extends ReduceStore {
  constructor() { super(AppDispatcher); }
  getInitialState() { return []; }

  reduce(state, action) {
    if (action.type === 'ADD_NEW_ITEM') {
      return [ ...state.items, action.item ]
    }

    return state;
  }
}
```

# Part Six

## Redux and React

# We're going to do that thing again.

- I'm going to code up a quick example using Redux and React.
- Then I'm going to explain the moving pieces once you've seen it in action.

# **React State vs. Redux State**

You Might Not Need Redux - X

A Medium Corporation [US] | https://medium.com/@dan\_abramov/you-might-not-need-redux-be46360cf367

Upgrade

Medium

Applause from Jay Phelps, Drew Reynolds, and 3,618 others

Dan Abramov [Follow](#)  
Working on @reactjs. Co-author of Redux and Create React App. Building tools for humans.  
Sep 19, 2016 · 3 min read

## You Might Not Need Redux

People often choose Redux before they need it. “What if our app doesn’t scale without it?” Later, developers frown at the indirection Redux introduced to their code. “Why do I have to touch three files to get a simple feature working?” Why indeed!

People blame Redux, React, functional programming, immutability, and many other things for their woes, and I understand them. It is natural to compare Redux to an approach that doesn’t require “boilerplate” code to update the state, and to conclude that Redux is just complicated. In a way it is, and by design so.

Redux offers a tradeoff. It asks you to:

- Describe application state as plain objects and arrays.

8.5K

69

Next story  
ES8 was Released and here are...

You Might Not Need Redux - X Steve

A Medium Corporation [US] | https://medium.com/@dan\_abramov/you-might-not-need-redux-be46360cf367

# Medium

Applause from Jay Phelps, Drew Reynolds, and 3,618 others



Dan Abramov  
Working on @reactjs. Co-author of Redux and Create React App.  
Building tools for humans.  
Sep 19, 2016 · 3 min read

## You Might Not Need Redux

8.5K

Q 69

Upvote icon

Bookmark icon

Should I put stuff in the  
Redux store?

 It depends. 

Like most things with  
computers—it's a trade off.

The cost? Indirection.

```
class NewItem extends Component {
  state = { value: '' };

  handleChange = event => {
    const value = event.target.value;
    this.setState({ value });
  };

  handleSubmit = event => {
    const { onSubmit } = this.props;
    const { value } = this.state;

    event.preventDefault();

    onSubmit({ value, packed: false, id: uniqId() });
    this.setState({ value: '' });
  };

  render() { ... }
}
```

# Now, it will be in four files!

- `NewItem.js`
- `NewItemContainer.js`
- `new-item-actions.js`
- `items-reducer.js`

`this.setState()` is inherently  
simpler to reason about than  
actions, reducers, and stores.

Think about short-term view state  
versus long-term model state.

# **react-redux**

Good news! The react-redux  
library is also *super* small.

**Even *smaller* than Redux!**

```
<Provider>
  connect()
```

(Yea, that's it.)

`connect();`

# It's the higher-order component pattern!

Pick which things you want from the store.  
(Maybe transform the data if you need to.)

```
connect(mapStateToProps, mapDispatchToProps)(WrappedComponent);
```

Pick which actions this component needs.

Mix these two together and pass them as props to a presentational component.

This is a function that you make that takes the entire state tree and boils it down to just what your components needs.

```
const mapStateToProps = ( state ) => {  
  return {  
    item: state.items  
  }  
};
```

This would be the entire state tree.

```
const mapStateToProps = (state) => {  
  return state;  
};
```

This would be just the packed items.

```
const mapStateToProps = (state) => {
  return {
    items: items.filter(item => item.packed)
  };
};
```

A function that receives `store.dispatch` and then returns an object with methods that will call `dispatch`.

```
const mapDispatchToProps = (dispatch) => ({  
  onCheckOff(id) {  
    dispatch(toggleItem(id))  
  },  
  onRemove(id) {  
    dispatch(removeItem(id))  
  }  
});
```

# Remember bindActionCreators?

```
const mapDispatchToProps = (dispatch) => {
  return bindActionCreators({
    updateNewItemValue,
    addNewItem
  }, dispatch)
};
```

**This is all very cool, but where is the store  
and how does connect() know about it?**

```
<Provider store={store}>
  <Application />
</Provider>
```

The provider is a component that sets the context of all of its children?

```
this.props;  
this.state;  
this.context;
```

Context - React

Secure | https://reactjs.org/docs/context.html

Steve

React Docs Tutorial Community Blog Search docs v16.1.1 GitHub

# Why Not To Use Context

The vast majority of applications do not need to use context.

If you want your application to be stable, don't use context. It is an experimental API and it is likely to break in future releases of React.

If you aren't familiar with state management libraries like [Redux](#) or [MobX](#), don't use context. For many practical applications, these libraries and their React bindings are a good choice for managing state that is relevant to many components. It is far more likely that Redux is the right solution to your problem than that context is the right solution.

If you aren't an experienced React developer, don't use context. There is usually a better way to implement functionality just using props and state.

If you insist on using context despite these warnings, try to isolate your use of context to a small area and avoid using the context API directly when possible so that it's easier to upgrade when the API changes.

---

## How To Use Context

Suppose you have a structure like:

QUICK START ▾

ADVANCED GUIDES ▾

- JSX In Depth
- Typechecking With PropTypes
- Static Type Checking
- Refs and the DOM
- Uncontrolled Components
- Optimizing Performance
- React Without ES6
- React Without JSX
- Reconciliation

**Context**

- Portals
- Error Boundaries
- Web Components
- Higher-Order Components
- Integrating with Other Libraries
- Accessibility

REFERENCE ▾

CONTRIBUTING ▾

```
class Provider extends Component {
  getChildContext() {
    return { store: this.props.store };
  }
  render() {
    return Children.only(this.props.children)
  }
}
```

```
class Provider extends Component {...}

Provider.childContextTypes = {
  store: PropTypes.shape({
    subscribe: PropTypes.func.isRequired,
    dispatch: PropTypes.func.isRequired,
    getState: PropTypes.func.isRequired,
  }),
  storeSubscription: PropTypes.shape({
    trySubscribe: PropTypes.func.isRequired,
    tryUnsubscribe: PropTypes.func.isRequired,
    notifyNestedSubs: PropTypes.func.isRequired,
    isSubscribed: PropTypes.func.isRequired,
  })
};
```

```
class SomePresentationalComponent extends Component {  
  render() { ... }  
}
```

```
SomePresentationalComponent.contextTypes = {  
  store: PropTypes.shape({  
    subscribe: PropTypes.func.isRequired,  
    dispatch: PropTypes.func.isRequired,  
    getState: PropTypes.func.isRequired,  
  })  
};
```

# Exercise

- Previously, we wired up our actions, reducers, and the store.
- Now, it's on us to actually connect Redux to our React application.
- I'll get us started, and then we'll review together.
- You may not get it all done. Start with either the packed or unpacked list. Put some items in the initial state if that helps.

# Part Seven

## Redux Thunk

Thunk?

***thunk*** (noun): a function  
returned from another function.

```
function definitelyNotAThunk() {  
    return function aThunk() {  
        console.log('Hello, I am a thunk.')  
    }  
}
```

**But, why is this useful?**

The major idea behind a thunk is  
that it's code to be executed later.

We've been a bit quiet about  
asynchronous code.

Here is the thing with Redux—it only accepts objects as actions.

But, we might not have the data ready yet.

We might be waiting for it.

# **redux-thunk**

**redux-thunk** is a middleware that allows us to dispatch a function (thunk) now that will dispatch a legit action later.

```
export const getAllItems = () => ({  
  type: UPDATE_ALL_ITEMS,  
  items,  
});
```

```
export const getAllItems = () => {
  return dispatch => {
    Api.getAll().then(items => {
      dispatch({
        type: UPDATE_ALL_ITEMS,
        items,
      });
    });
  };
};
```

# Exercise

- I'll implement fetching all of the items and adding a new item from the “server.”
- You'll implement toggling, removing, and marking all as unpacked.
- Then we'll go over it together.

# Part Eight

## Redux Saga

The action creators in **redux-thunk**  
aren't pure and this can make testing  
tricky.

```
it('fetches items from the database', () => {
  const itemsInDatabase = {
    items: [{ id: 1, value: 'Cheese', packed: false }],
  };

  fetchMock.getOnce('/items', {
    body: itemsInDatabase,
    headers: { 'content-type': 'application/json' },
  });

  const store = mockStore({ items: [] });

  return store.dispatch(actions.getItems()).then(() => {
    expect(store.getItems()).toEqual({
      type: GET_ALL_ITEMS,
      items: itemsInDatabase
    });
  });
});
```

It would be *great* if we could separate out the dispatch of actions from the talking to the database.

The tricky part is that we need the information to dispatch the action that's going to the store.

# An aside: Generator functions

Generators are a way to  
delay execution.

```
function* generateThreeNumbers() {  
    yield 1;  
    yield 2;  
    yield 3;  
}
```

Line: 5:2 | JavaScript | Soft Tabs: 2 | generateThreeNumbers

```
function* generateThreeNumbers() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
var numberIterator = generateThreeNumbers();
```

```
function* generateThreeNumbers() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
var numberIterator = generateThreeNumbers();  
  
numberIterator.next() // { value: 1, done: false }  
numberIterator.next() // { value: 2, done: false }  
numberIterator.next() // { value: 3, done: false }  
numberIterator.next() // { value: undefined, done: true }
```

13

We can also pass new data into them after they've been started.

```
function* addNumbersEventually(a) {  
    const b = yield a;  
    return a + b;  
}
```

Line: 5 | JavaScript | Soft Tabs: 2 | addNumbersEventually

```
function* addNumbersEventually(a) {  
  const b = yield a;  
  return a + b;  
}
```

```
function* addNumbersEventually(a) {  
    const b = yield a;  
    return a + b;  
}
```

```
function* addNumbersEventually(a) {  
    const b = yield a;  
    return a + b;  
}
```

```
function* addNumbersEventually(a) {  
    const b = yield a;  
    return a + b;  
}
```

Line: 2:9-2:10 | JavaScript | Soft Tabs: 2 | addNumbersEventually

```
function* addNumbersEventually(a) {  
  const b = yield a;  
  return a + b;  
}  
  
const it = addNumbersEventually(1);  
  
it.next(2) // Yields 1, takes in 2 and sets "b"  
it.next() // Yields 3 and is done.
```

So, sometimes we need to  
wait a bit for information.

And, we now have the ability to pause functions and come back to them later with more information.

It seems like we could do  
something clever here.

```
const request = (url) => {
  ajax('GET', url).then(response => it.next(JSON.parse(response)));
}

function* main() {
  const steve = yield request('https://api.github.com/users/stevekinney');
  const lon = yield request('https://api.github.com/users/lawnsea');

  console.log(steve, lon);
}

const it = main();
it.next();
```

```
const request = (url) => {
  ajax('GET', url).then(response => it.next(JSON.parse(response)));
}

function* main() {
  const steve = yield request('https://api.github.com/users/stevekinney');
  const lon = yield request('https://api.github.com/users/lawnsea');

  console.log(steve, lon);
}

const it = main();
it.next();
```

```
const request = (url) => {
  ajax('GET', url).then(response => it.next(JSON.parse(response)));
}

function* main() {
  const steve = yield request('https://api.github.com/users/stevekinney');
  const lon = yield request('https://api.github.com/users/lawnsea');

  console.log(steve, lon);
}

const it = main();
it.next();
```

```
const request = (url) => {
  ajax('GET', url).then(response => it.next(JSON.parse(response)));
}

function* main() {
  const steve = yield request('https://api.github.com/users/stevekinney');
  const lon = yield request('https://api.github.com/users/lawnsea');

  console.log(steve, lon);
}

const it = main();
it.next();
```

```
const request = (url) => {
  ajax('GET', url).then(response => it.next(JSON.parse(response)));
}

function* main() {
  const steve = yield request('https://api.github.com/users/stevekinney');
  const lon = yield request('https://api.github.com/users/lawnsea');

  console.log(steve, lon);
}

const it = main();
it.next();
```

**Okay, so back to sagas...**

Thunks let us cheat by dispatching a function that eventually dispatches an object.

With a saga, we set up a generator function to intercept the dispatched action object...

...pause until we have have the data,  
and then dispatch the another action  
with the data to modify the UI.

Instead of immediately adding a new item to the page, let's create an action for the POST request.

```
export const requestNewFriend = () => {
  return {
    type: REQUEST_NEW_FRIEND,
  };
};

export const addFriendToList = friend => ({
  type: ADD_FRIEND_TO_LIST,
  friend,
});
```

Elements Console Memory Sources Audits Network Performance Redux >

Inspector Jetsetter

filter...

@@INIT 3:02:51.01

Diff Action State Diff Test

Tree Raw

(states are equal)

Pause Lock

⋮ X

Request New Friend

# Steps

- User clicks on “Request New Friend” on the page.
- This fires a REQUEST\_NEW\_FRIEND action.
- The saga is listening for this action and intercepts.
- It makes the API request and then dispatches an ADD\_NEW\_FRIEND event, which updates the store and subsequently the view.

```
import createSagaMiddleware from 'redux-saga';

// Import a file with saga that you made (or, will make).
import saga from './saga';

// Instantiate the middleware
const sagaMiddleware = createSagaMiddleware();

const middleware = [sagaMiddleware];

const store = createStore(
  reducers,
  initialState,
  composeEnhancers(applyMiddleware(...middleware), ...enhancers),
);

// Start up the generator
sagaMiddleware.run(saga);
```

# Register, intercept, and put

```
export default function* rootSaga() {
  yield all([fetchUserFromApi()]);
}

export function* fetchUserFromApi() {
  yield takeEvery(REQUEST_NEW_FRIEND, makeApiRequest);
}

export function* makeApiRequest() {
  const friend = yield call(Api.requestNewFriend);
  yield put(addFriendToList(friend));
}
```

# Live Coding

# Exercise

- Check master branch out the **Offices and Dragons** repository.
  - Do an extra git pull to make sure you have the freshest version.
  - Effectively, we want to implement the steps we saw a few slides back: user clicks button → dispatch request action → make an API call → dispatch an action with the data.

**Sagas are pretty cool.**

```
takeEvery();  
takeLatest();
```

```
import { select, takeEvery } from 'redux-saga/effects'

function* logActionsAndState() {
  yield takeEvery('*', function* logger(action) {
    const state = yield select();

    console.log('action', action);
    console.log('state after', state);
  });
}
```

```
function* cancellableRequest() {
  while (true) {
    yield take('REQUEST_LYFT');
    yield race({
      task: call(Api.makeRequest),
      cancel: take('CANCEL_LYFT')
    });
  }
}
```

# Part Nine

## MobX

# An Aside: Computed Properties

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    fullName() {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

```
const person = new Person('Grace', 'Hopper');

person.firstName; // 'Grace'
person.lastName; // 'Hopper'
person.fullName; // function fullName() {...}
```

```
const person = new Person('Grace', 'Hopper');

person.firstName; // 'Grace'
person.lastName; // 'Hopper'
person.fullName(); // 'Grace Hopper'
```

Ugh. 😞

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    get fullName() {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

```
const person = new Person('Grace', 'Hopper');

person.firstName; // 'Grace'
person.lastName; // 'Hopper'
person.fullName; // 'Grace Hopper'
```

Much Better! 😎

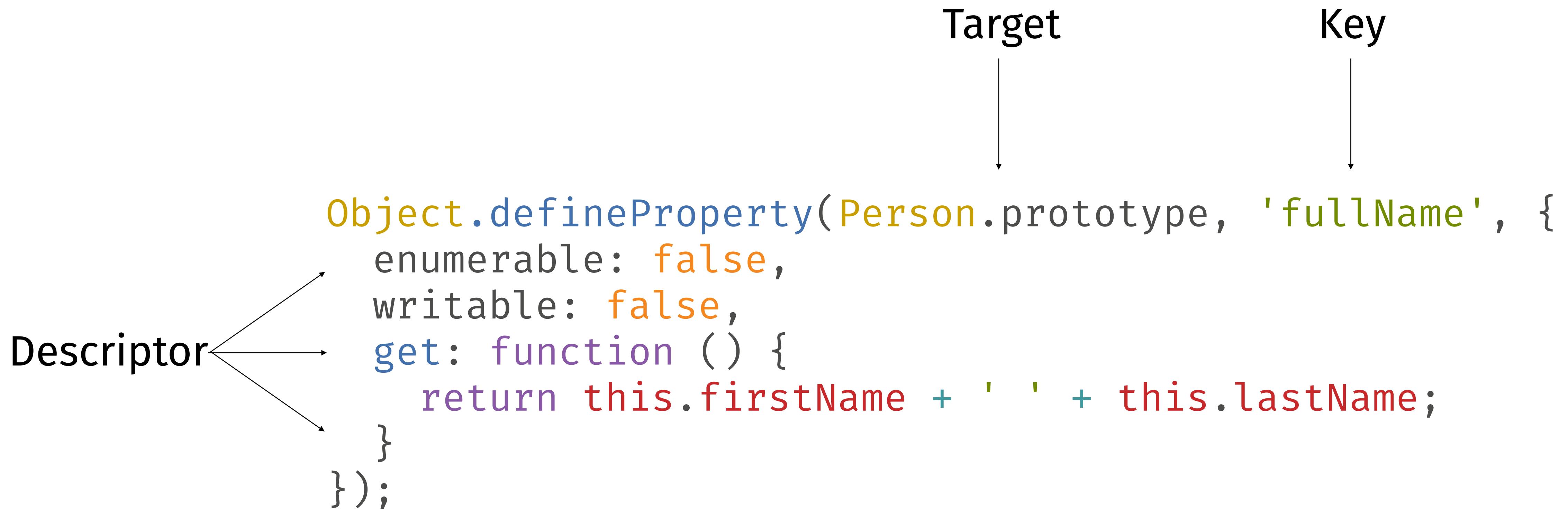
Getters and setters may seem like some fancy new magic, but they've been around since ES5.

# Not as elegant, but it'll do.

```
function Person(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
}  
  
Object.defineProperty(Person.prototype, 'fullName', {  
    get: function () {  
        return this.firstName + ' ' + this.lastName;  
    }  
});
```

# An Aside: Decorators

Effectively decorators provide a syntactic sugar for higher-order functions.



```
function decoratorName(target, key, descriptor) {  
    // ...  
}
```

```
function readonly(target, key, descriptor) {  
    descriptor.writable = false;  
    return descriptor;  
}
```

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    @readonly get fullName() {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

# npm install core-decorators

@autobind  
@deprecate  
@readonly  
@memoize  
@debounce  
@profile

Jets x Stor x Rel x Rec x reac x Pizz x mob x style x Expl x obje x Obje x Obje x Add x Inbo x core x loda x Steve

NPM, Inc. [US] | https://www.npmjs.com/package/lodash-decorators

New Programs Makers

npm Enterprise features pricing documentation support

**npm** find packages

Painless code sharing. npm Orgs help your team discover, share, and reuse code. [Create a free org »](#)

**★ lodash-decorators** public

Decorators using lodash functions. View the [API docs](#) for more in depth documentation.

build passing npm version

- [Install](#)
  - [Polyfills](#)
- [Usage](#)
  - [Decorators](#)
  - [Example](#)
  - [Partials](#)
  - [Example](#)
  - [Composition](#)
  - [Example](#)
- [Instance Decorators](#)
- [Mixin](#)
  - [Example](#)
- [Attempt](#)
  - [Example](#)
- [Bind](#)
  - [Example](#)
  - [Example](#)

[npm i lodash-decorators](#)

[how? learn more](#)

[steelsojka](#) published a week ago

**4.5.0** is the latest of 64 releases

[github.com/steelsojka/lodash-decorators](https://github.com/steelsojka/lodash-decorators)

MIT

[Collaborators list](#)

**Stats**

806 downloads in the last day

10,919 downloads in the last week

26,092 downloads in the last month

**Okay, so... MobX**

Imagine if you could simply  
change the state tree.

Bear with me here.

A primary tenet of using MobX is that you can store state in a simple data structure and allow the library to care of keeping everything up to date.

<http://bit.ly/super-basic-mobx>

# Ridiculously simplified, not real code™

```
const onChange = (oldValue, newValue) => {
  // Tell MobX that this value has changed.
}

const observable = (value) => {
  return {
    get() { return value; },
    set(newValue) {
      onChange(this.get(), newValue);
      value = newValue;
    }
  }
}
```

# This code...

```
class Person {  
    @observable firstName;  
    @observable lastName;  
  
    constructor(firstName, lastName) {  
        this.firstName;  
        this.lastName;  
    }  
}
```

# ...is effectively equivalent.

```
function Person (firstName, lastName) {  
    this.firstName;  
    this.lastName;  
  
    extendObservable(this, {  
        firstName: firstName,  
        lastName: lastName  
    });  
}
```

```
const extendObservable = (target, source) => {
  source.keys().forEach(key => {
    const wrappedInObservable = observable(source[key]);
    Object.defineProperty(target, key, {
      set: value.set.
        get: value.get
    });
  });
};
```

```
// This is the @observable decorator
const observable = (object) => {
  return extendObservable(object, object);
};
```

# Four-ish major concepts

- Observable state
- Actions
- Derivations
  - Computed properties
  - Reactions

Computed properties update their value based on observable data.

Reactions produce side effects.

```
class PizzaCalculator {
    numberOfPeople = 0;
    slicesPerPerson = 2;
    slicesPerPie = 8;

    get slicesNeeded() {
        return this.numberOfPeople * this.slicesPerPerson;
    }

    get piesNeeded() {
        return Math.ceil(this.slicesNeeded / this.slicesPerPie);
    }

    addGuest() { this.numberOfPeople++; }
}
```

```
import { action, observable, computed } from 'mobx';

class PizzaCalculator {
  @observable numberOfPeople = 0;
  @observable slicesPerPerson = 2;
  @observable slicesPerPie = 8;

  @computed get slicesNeeded() {
    console.log('Getting slices needed');
    return this.numberOfPeople * this.slicesPerPerson;
  }

  @computed get piesNeeded() {
    console.log('Getting pies needed');
    return Math.ceil(this.slicesNeeded / this.slicesPerPie);
  }

  @action addGuest() {
    this.numberOfPeople++;
  }
}
```

You may notice that you can sometimes omit the `@action` annotation.

Originally, this was just for documenting, but we'll see that it has some special implications now-a-days.

You can also pass most common data structures to MobX.

- Objects – `observable({})`
- Arrays – `observable([])`
- Maps – `observable(new Map())`

**Caution:** If you add properties to an object after you pass it to observable(), those new properties will not be observed.

Use a Map() if you're going  
to be adding keys later on.

# MobX with React

```
@observer class Counter extends Component {  
  render() {  
    const { counter } = this.props;  
    return (  
      <section>  
        <h1>Count: {counter.count}</h1>  
        <button onClick={counter.increment}>Increment</button>  
        <button onClick={counter.decrement}>Decrement</button>  
        <button onClick={counter.reset}>Reset</button>  
      </section>  
    );  
  }  
}
```

```
const Counter = observer(({ counter }) => (
  <section>
    <h1>Count: {counter.count}</h1>
    <button onClick={counter.increment}>Increment</button>
    <button onClick={counter.decrement}>Decrement</button>
    <button onClick={counter.reset}>Reset</button>
  </section>
));
```

```
class ContainerComponent extends Component () {
  componentDidMount() {
    this.stopListening = autorun(() => this.render());
  }

  componentWillUnmount() {
    this.stopListening();
  }

  render() { ... }
}
```

```
import { Provider } from 'mobx-react';

import ItemStore from './store/ItemStore';
import Application from './components/Application';

const itemStore = new ItemStore();

ReactDOM.render(
  <Provider itemStore={itemStore}>
    <Application />
  </Provider>,
  document.getElementById('root'),
);
```

```
@inject('itemStore')
class NewItem extends Component {
  state = { ... };

  handleChange = (event) => { ... }

  handleSubmit = (event) => { ... }

  render() { ... }

}
```

```
const UnpackedItems = inject('itemStore')(
  observer(({ itemStore }) => (
    <Items
      title="Unpacked Items"
      items={itemStore.filteredUnpackedItems}
      total={itemStore.unpackedItemsLength}
    >
      <Filter
        value={itemStore.unpackedItemsFilter}
        onChange={itemStore.updateUnpackedItemsFilter}
      />
      </Items>
    )),
);
```

# Exercise

- I'll implement the basic functionality for adding and removing items.
- Then you'll implement toggling.
- Then I'll implement filtering.
- Then you'll implement marking all as unpacked.

# Exercise

- Whoa, it's another exercise!
- This time it will be the same flow as last time, but we're going to add asynchronous calls to the server into the mix.

# Conclusion

## Closing Thoughts

# **MobX versus Redux**

# **MobX ~~versus~~ Redux**

## **Dependency Graphs ~~versus~~ Immutable State Trees**

# Advantages of Dependency Graphs

- Easy to update
- There is a graph structure: nodes can refer to each other
- Actions are simpler and co-located with the data
- Reference by identity

# Advantages of Immutable State Trees

- Snapshots are cheap and easy
- It's a simple tree structure
- You can serialize the entire tree
- Reference by state

What does it mean to have to  
normalize your data?

```
state = {
  items: [
    { id: 1, value: "Storm Trooper action figure", owner: 2 },
    { id: 2, value: "Yoga mat", owner: 1 },
    { id: 4, value: "MacBook", owner: 3 },
    { id: 5, value: "iPhone", owner: 1 },
    { id: 7, value: "Melatonin", owner: 3 }
  ],
  owners: [
    { id: 1, name: "Logan", items: [2, 5] },
    { id: 2, name: "Wes", items: [1] },
    { id: 3, name: "Steve", items: [4, 7] }
  ]
}
```

```
state = {
  items: {
    1: { id: 1, value: "Storm Trooper action figure", owner: 2 },
    2: { id: 2, value: "Yoga mat", owner: 1 },
    4: { id: 4, value: "MacBook", owner: 3 },
    5: { id: 5, value: "iPhone", owner: 1 },
    7: { id: 7, value: "Melatonin", owner: 3 }
  },
  owners: {
    1: { id: 1, name: "Logan", items: [2, 5] },
    2: { id: 2, name: "Wes", items: [1] },
    3: { id: 3, name: "Steve", items: [4, 7] }
  }
}
```

Steve

GitHub, Inc. [US] | <https://github.com/mobxjs/mobx-state-tree>

## README.md



### mobx-state-tree

*Opinionated, transactional, MobX powered state container combining the best features of the immutable and mutable world for an optimal DX*

[npm package 1.1.0](#) [build passing](#) [coverage 95%](#) [chat on gitter](#)

Mobx and MST are amazing pieces of software, for me it is the missing brick when you build React based apps.  
Thanks for the great work!

Nicolas Galle [full post](#)

Introduction blog post [The curious case of MobX state tree](#)

## Contents

- [Installation](#)
- [Getting Started](#)
- [Talks & blogs](#)
- [Philosophy & Overview](#)
- [Examples](#)
- [Concepts](#)
  - [Trees, types and state](#)
  - [Creating models](#)

**Where can you take this from  
here?**

Could you implement the undo/  
redo pattern outside of Redux?

Would an action/reducer  
pattern be helpful in MobX?

Would `async/await` make a  
suitable replacement for thunks or  
sagas?

Can you implement undo  
with API requests?

**You now have a good sense  
of the lay of the land.**

But, even people who have some experience are still figuring this stuff out.



Dan Abramov @dan\_abramov · 26 Aug 2016

I think Redux is popular partly because of deficiencies in React state model. I also think local component state is ultimately better.



2



19



43



Dan Abramov @dan\_abramov · 26 Aug 2016

I also hoped that Redux would encourage innovation around React state model, not create a wall of “best practices” and “right ways”.



5



13



31



Dan Abramov

@dan\_abramov

Following

Think: what does Redux give you? What properties are at play? What properties does React state model lack? Can we fix this?

6:11 PM - 26 Aug 2016

● ● ● / G flux p × Back × Twitter, Inc. [US] https://twitter.com/ryanflorence/status/861603318824550400?ref\_src=tw... Steve

Ryan Thinki Ryan Matt Matt Which Steve

Twitter, Inc. [US] https://twitter.com/ryanflorence/status/861603318824550400?ref\_src=tw... ☆ ⓘ ⚡ :

Search Twitter

Ryan Florence .@ryanflorence Following

I'm gonna say it. Flux hampered React innovation: led everybody right back to events and "templates" and intimately knowing data changes.

9:26 AM - 8 May 2017

29 Retweets 111 Likes

23 29 111

Ryan Florence .@ryanflorence @ryanflorence Co-Author Read

Ryan Florence .@ryanflorence @ryanflorence · May 8

Tweet your reply

This is not a solved problem.

Welcome to the team.

