Redux and Mobx

Steve Kinney

A Frontend Masters Workshop

Hi, I’m **Steve.**
(@stevekinney)

TWILIO SendGrid

We’re going to talk about state—using Redux and MobX.

To build our understanding of how to manage state in a large application, 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.

What are we going to learn in this course?

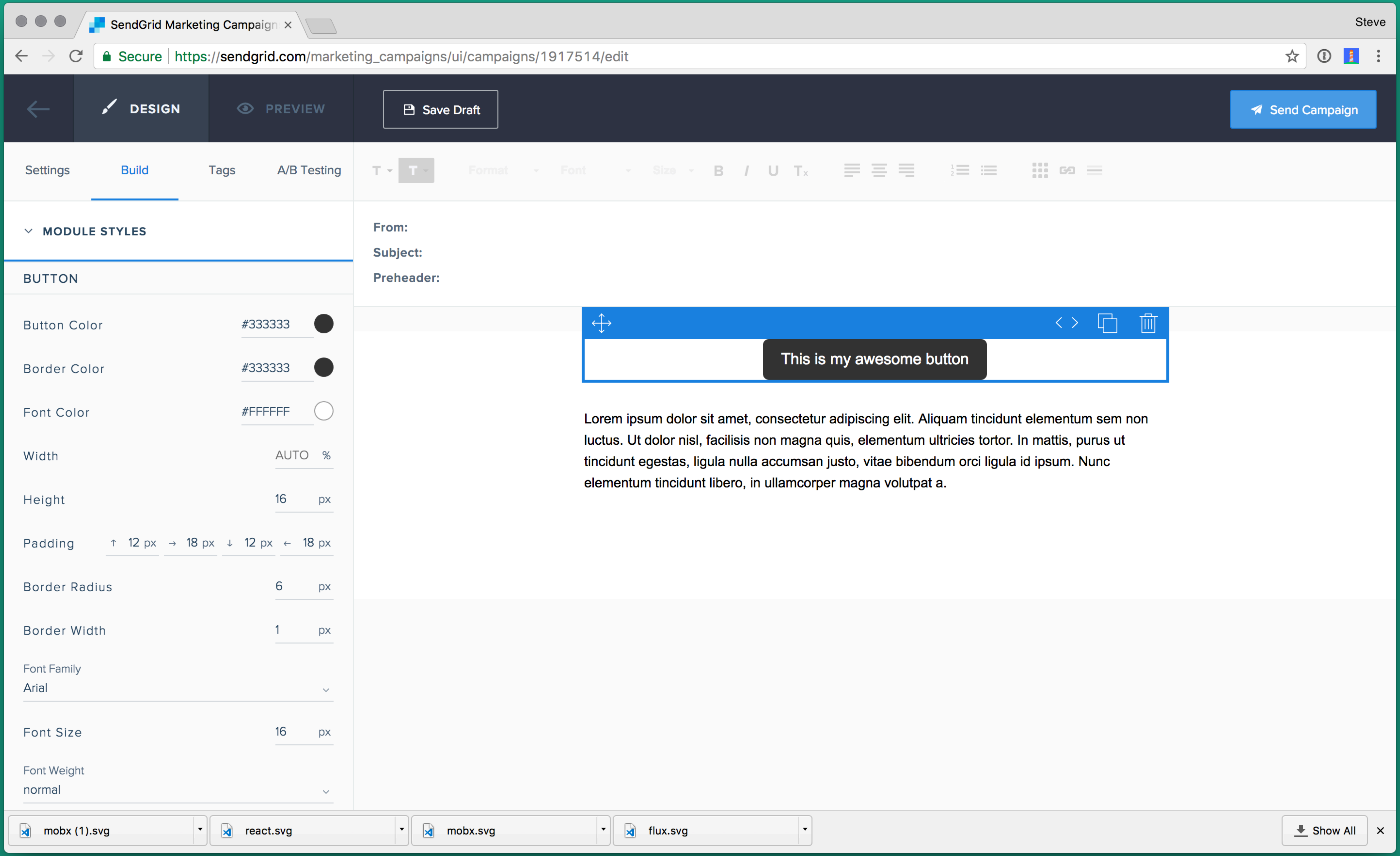
- The fundamentals of Redux—outside of React.
- Hooking Redux up to React.
- Normalizing the structure of your state.
- Using selectors to prevent needless re-renders.

What are we going to learn in this course?

- How middleware works with Redux.
- Making asynchronous API calls with Redux Thunk.
- Cracking open the doors to the wild world or Redux Observable.
- Mixing reactive and object-oriented state management with MobX.

Why is this important?

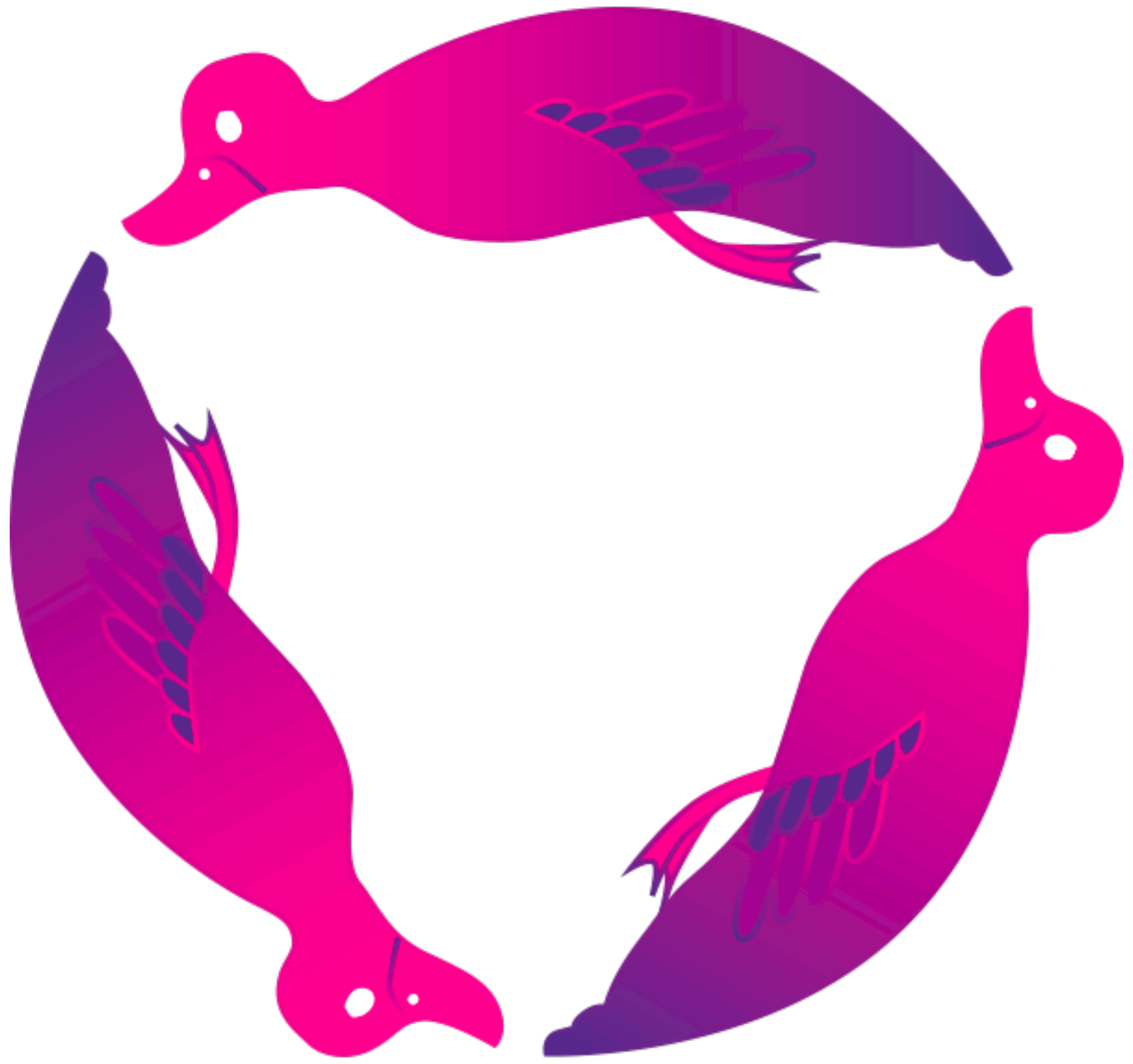
- Doing a massive refactor of your state later is fraught with peril.
- Having really great state management inspires joy.
- (The first point is *probably* more important.)



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

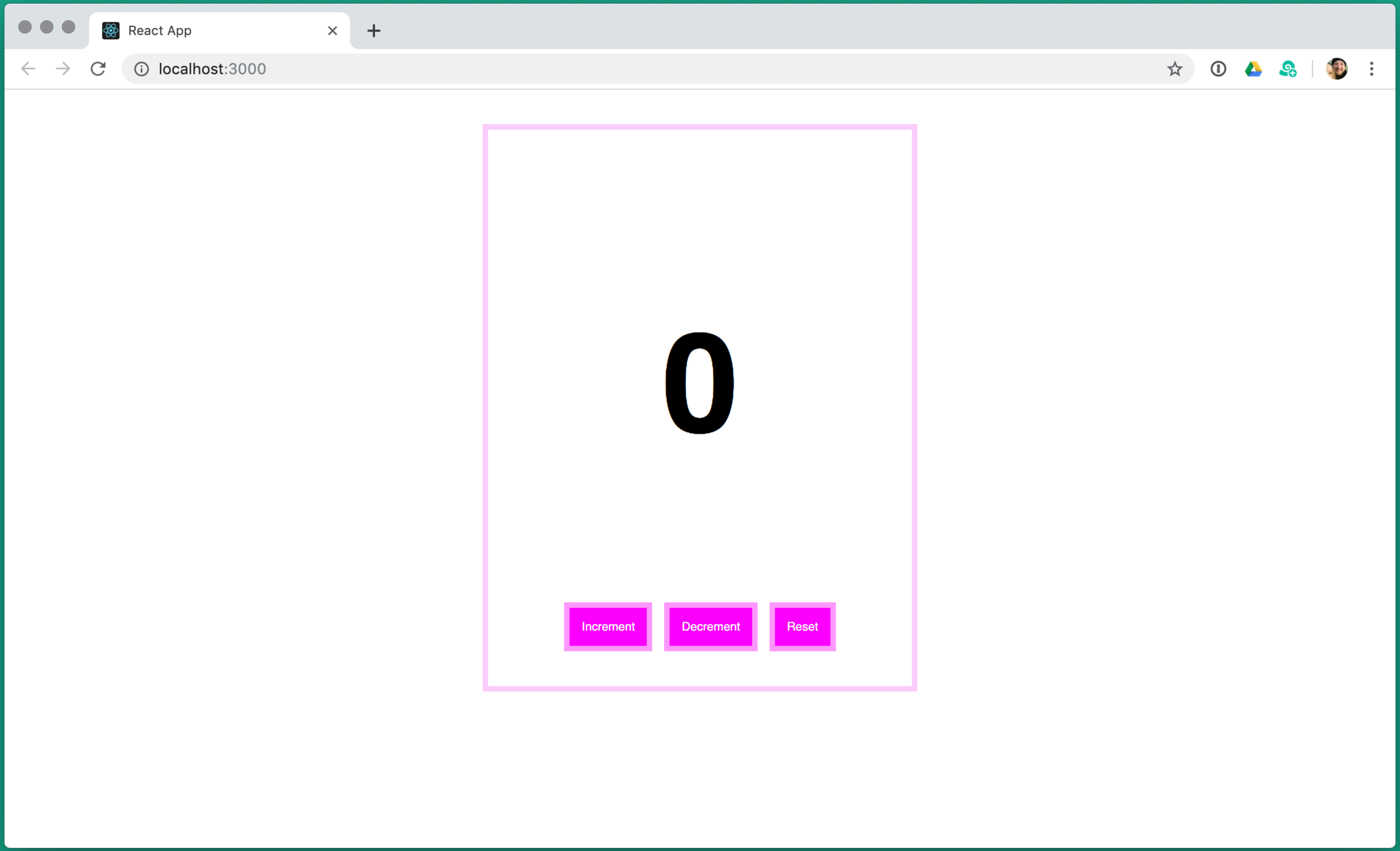
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam tincidunt elementum sem non luctus. Ut dolor nisl, facilisis non magna quis, elementum ultricies tortor. In mattis, purus ut tincidunt egestas, ligula nulla accumsan justo, vitae bibendum orci ligula id ipsum. Nunc elementum tincidunt libero,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

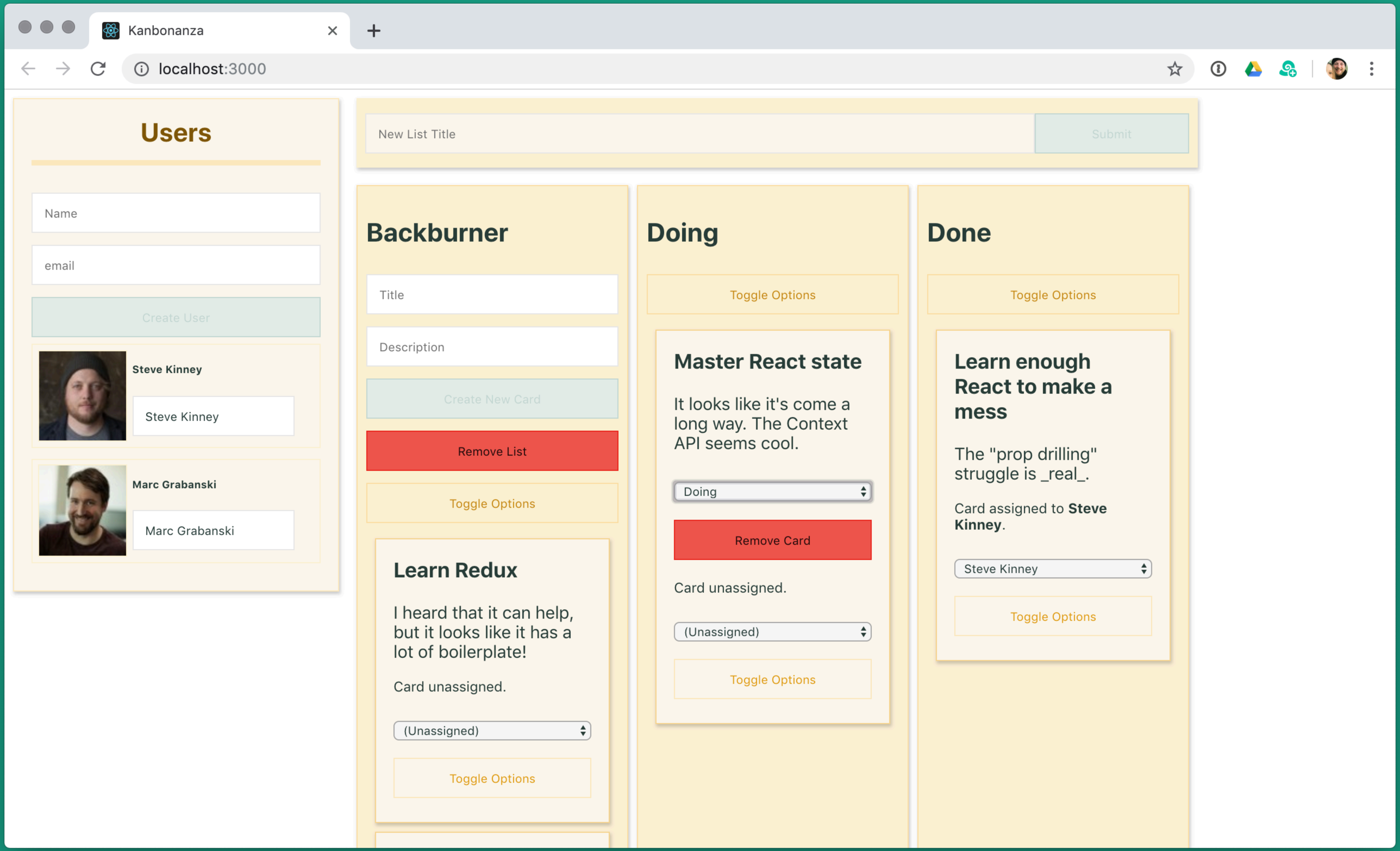
-thunk



]v[

What kind of applications are we going to build today?


React App
localhost:3000
0
Increment
Decrement
Reset



Kanbonanza

localhost:3000

Users

Name

email

Create User

Steve Kinney

Steve Kinney

Marc Grabanski

Marc Grabanski

New List Title

Submit

Backburner

Title

Description

Create New Card

Remove List

Toggle Options

Learn Redux

I heard that it can help, but it looks like it has a lot of boilerplate!

Card unassigned.

(Unassigned)

Toggle Options

Doing

Toggle Options

Master React state

It looks like it's come a long way. The Context API seems cool.

Doing

Remove Card

Card unassigned.

(Unassigned)

Toggle Options

Done

Toggle Options

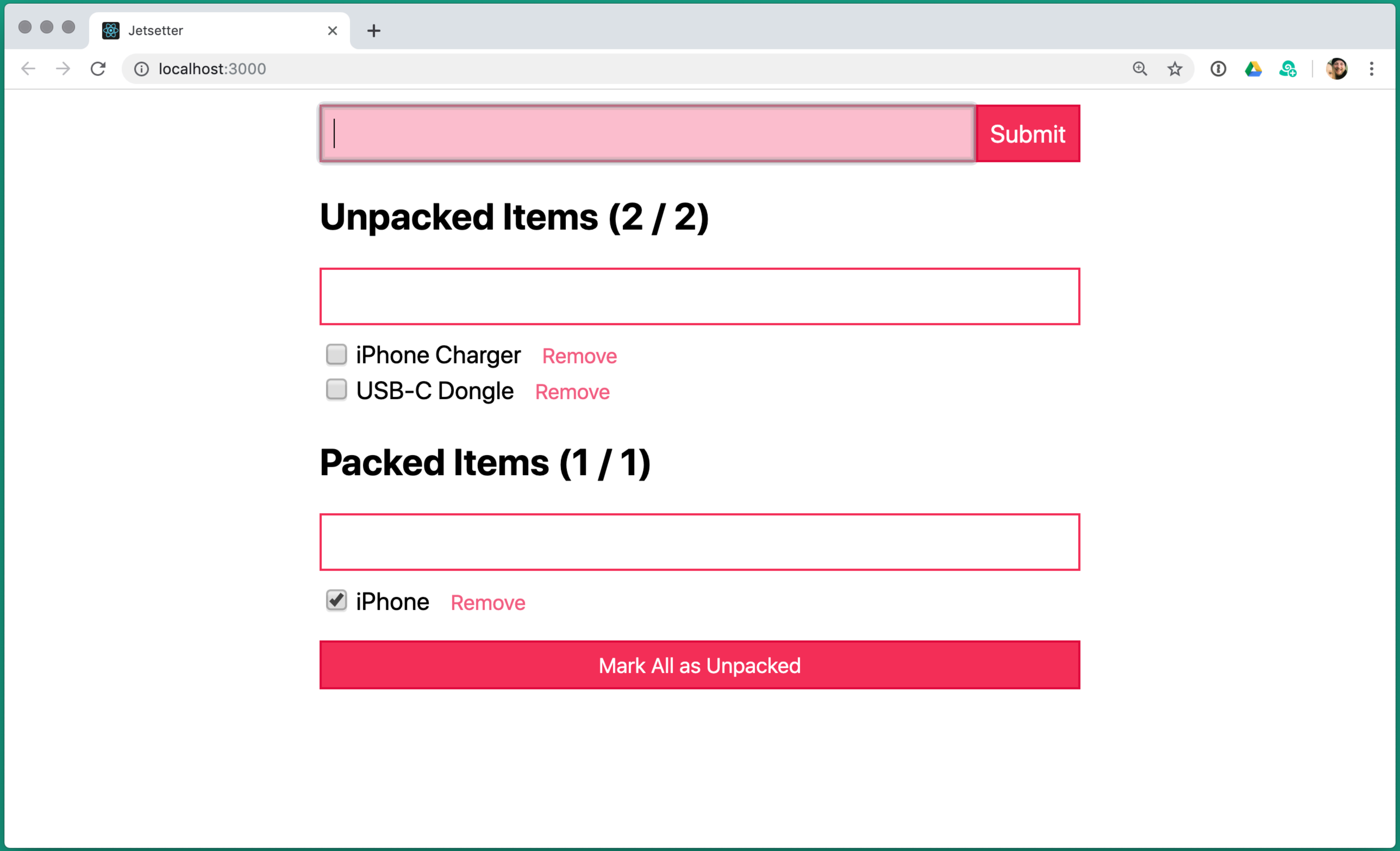
Learn enough React to make a mess

The "prop drilling" struggle is _real_.

Card assigned to **Steve Kinney**.

Steve Kinney

Toggle Options


Jetsetter

localhost:3000

Submit

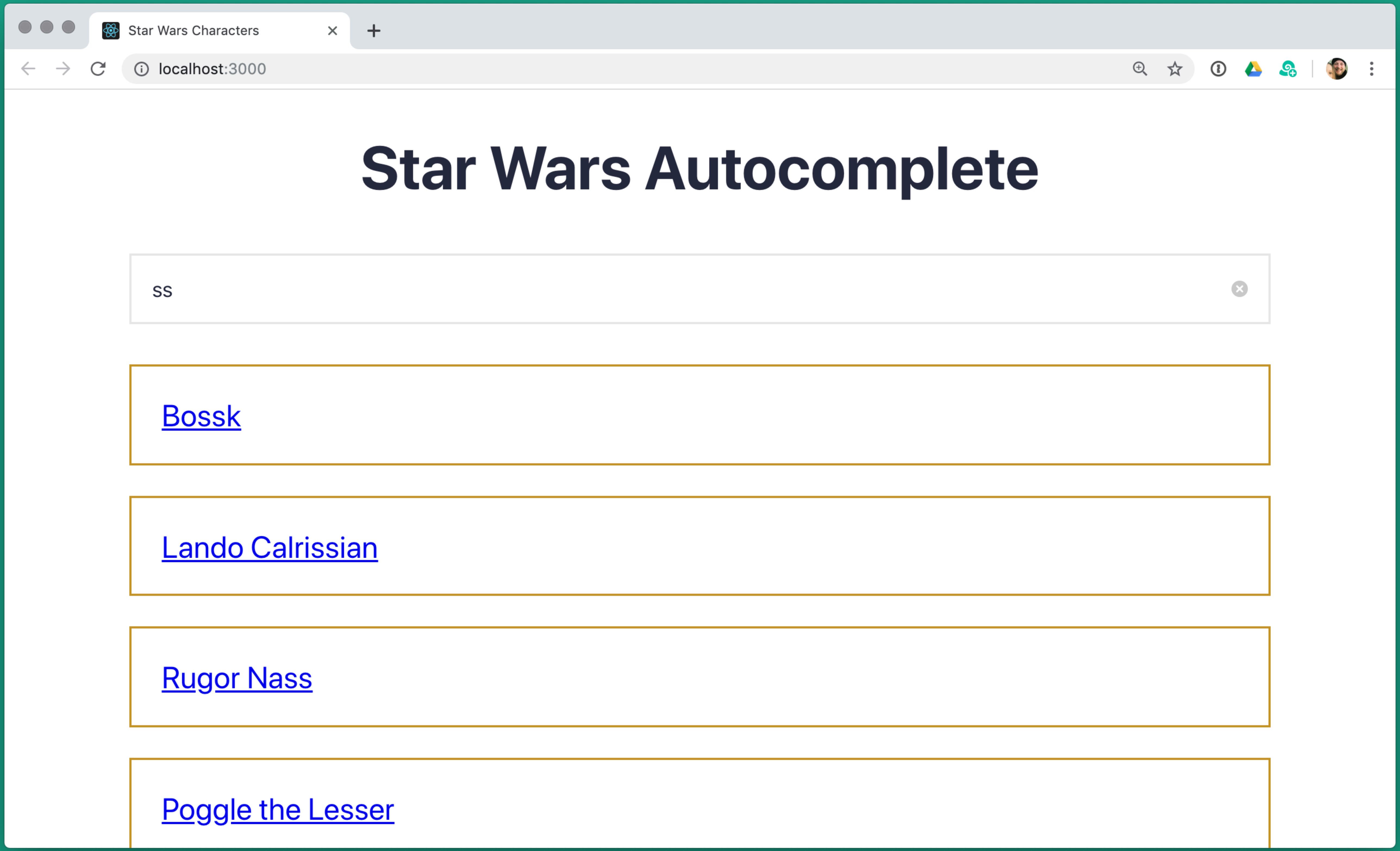
Unpacked Items (2 / 2)

- [ ] iPhone Charger Remove
- [ ] USB-C Dongle Remove

Packed Items (1 / 1)

- [x] iPhone Remove

Mark All as Unpacked



Star Wars Characters

localhost:3000

Star Wars Autocomplete

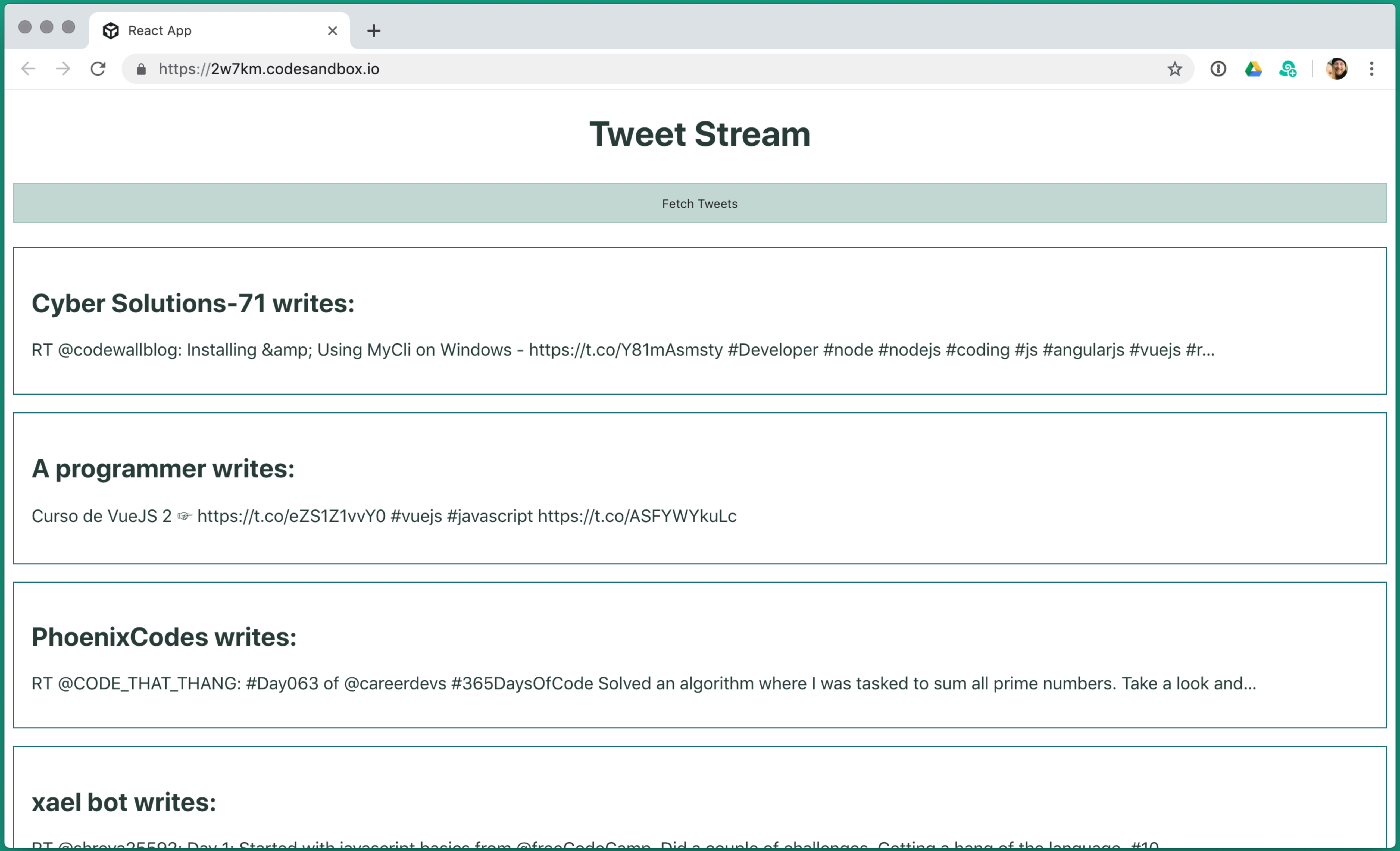
ss

Bossk

Lando Calrissian

Rugor Nass

Poggle the Lesser



Tweet Stream

Fetch Tweets

Cyber Solutions-71 writes:

RT @codewallblog: Installing &amp; Using MyCli on Windows - https://t.co/Y81mAsmsty #Developer #node #nodejs #coding #js #angularjs #vuejs #r...

A programmer writes:

Curso de VueJS 2 👉 https://t.co/eZS1Z1vvY0 #vuejs #javascript https://t.co/ASFYWYkuLc

PhoenixCodes writes:

RT @CODE_THAT_THANG: #Day063 of @careerdevs #365DaysOfCode Solved an algorithm where I was tasked to sum all prime numbers. Take a look and...

xael bot writes:

RT @shreya35592: Day 1: Started with javascript basics from @freeCodeCamp. Did a couple of challenges. Getting a hang of the language. #10

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.

Prologue

Some terminology and concepts before we get started

Pure vs. Impure Functions

Pure functions take arguments and return values based on those arguments.

Impure functions an mutate things from outside their scope or produce side effects.

```
// Pure
const add = (a, b) => {
  return a + b;
}
```

```
// Impure
const b;

const add = (a) ⇒ {
  return a + b;
}
```

```javascript
// Impure
const add = (a, b) => {
  console.log('lolololol');
  return a + b;
}
```

```
// Impure
const add = (a, b) => {
  Api.post('/add', { a, b }, (response) => {
    // Do something.
  })
};
```

Mutating arrays and objects is also impure.

Not Mutating Objects and Arrays

```javascript
// Copy object
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original);
```

```javascript
// Copy object
const original = { a: 1, b: 2 };
const copy = { ...original };
```

```
// Extend object
const original = { a: 1, b: 2 };
const extension = { c: 3 };
const extended = Object.assign({}, original, extension);
```

```javascript
// Extend object
const original = { a: 1, b: 2 };
const extension = { c: 3 };
const extended = { ...original, ...extension };
```

```
// Copy array
const original = [1, 2, 3];
const copy = [1, 2, 3].slice();
```

```
// Copy array
const original = [1, 2, 3];
const copy = [ ...original ];
```

```
// Extend array
const original = [1, 2, 3];
const extended = original.concat(4);
const moreExtended = original.concat([4, 5]);
```

```
// Extend array
const original = [1, 2, 3];
const extended = [ ...original, 3, 4 ];
const moreExtended = [ ...original, ...extended ];
```

Chapter One

Redux without React

What is Redux?

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

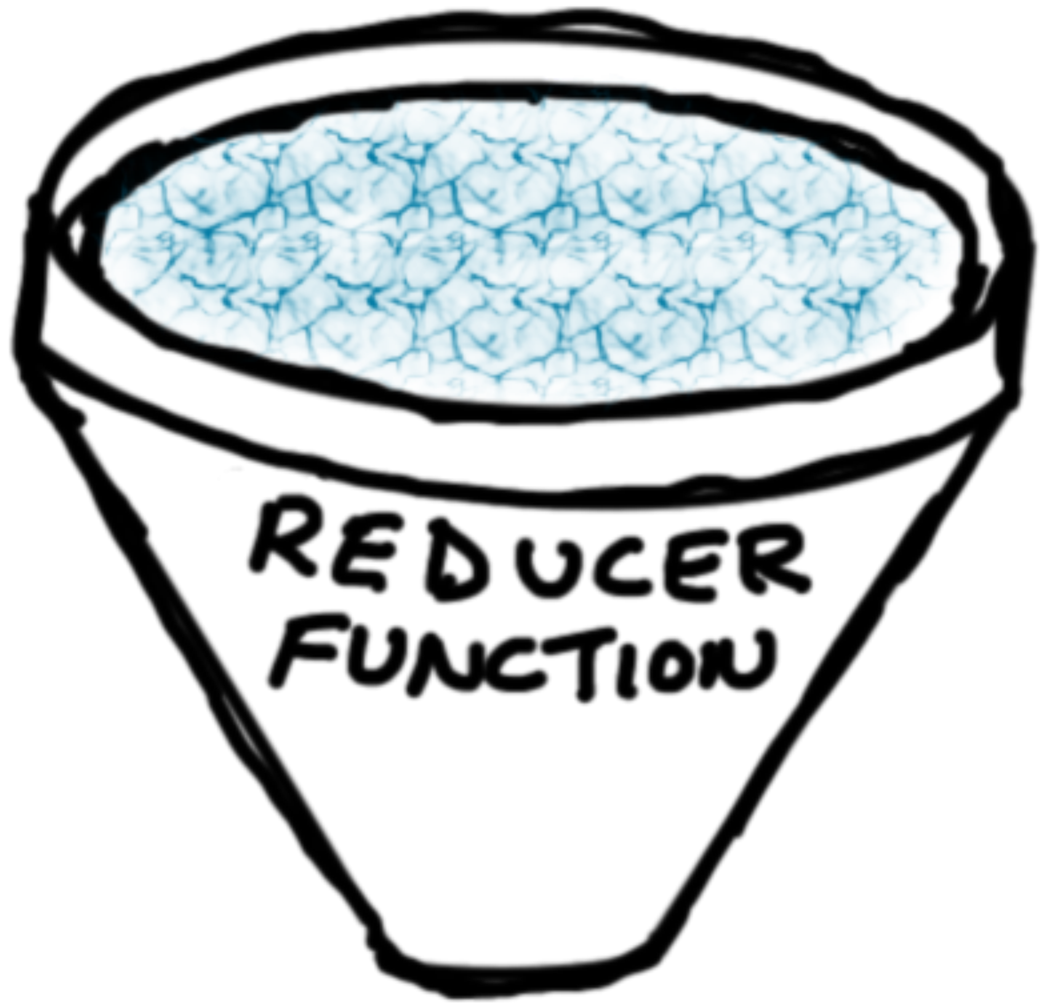
The whole state tree of your application is kept in one store.

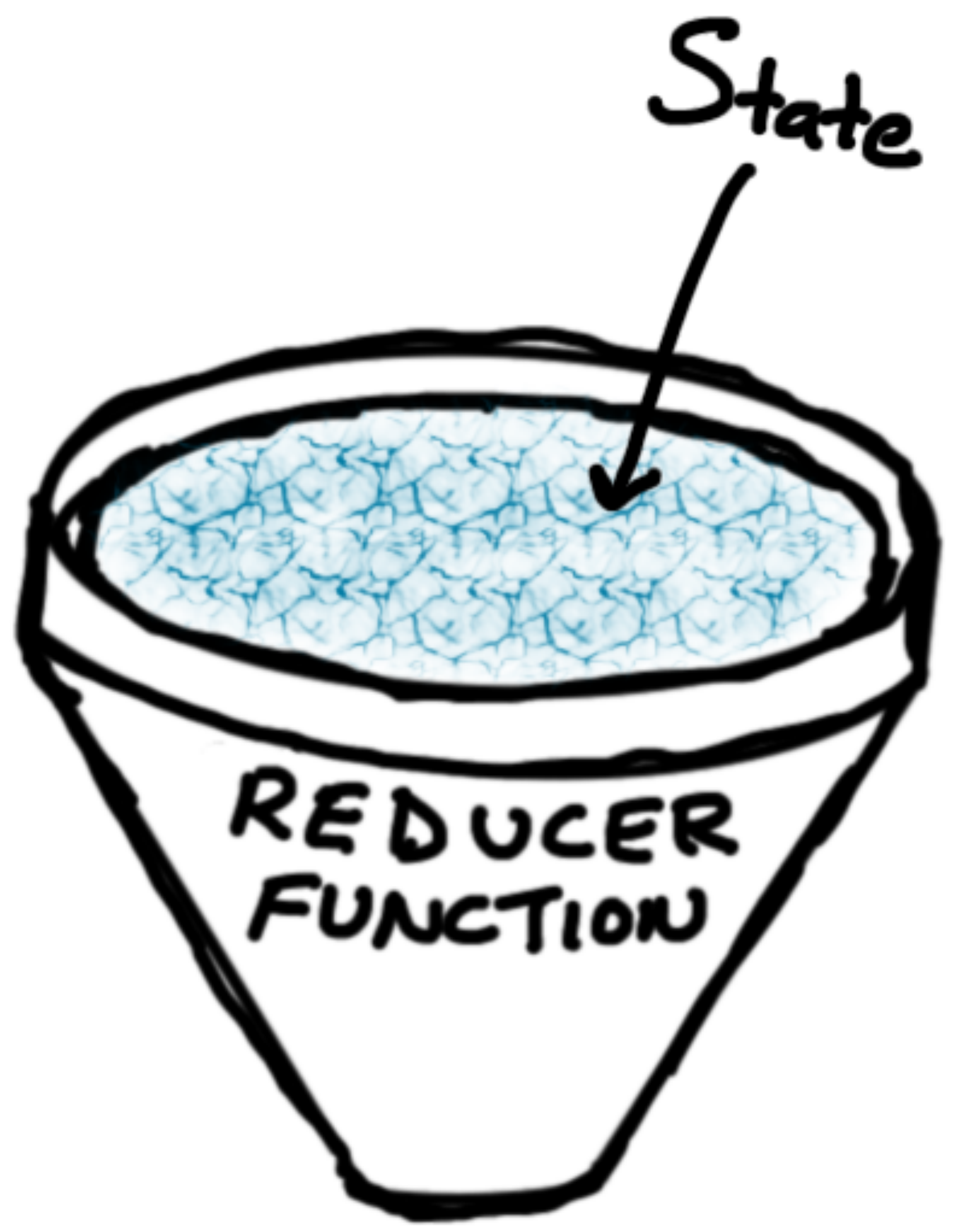
Just one plain old JavaScript object. 😋

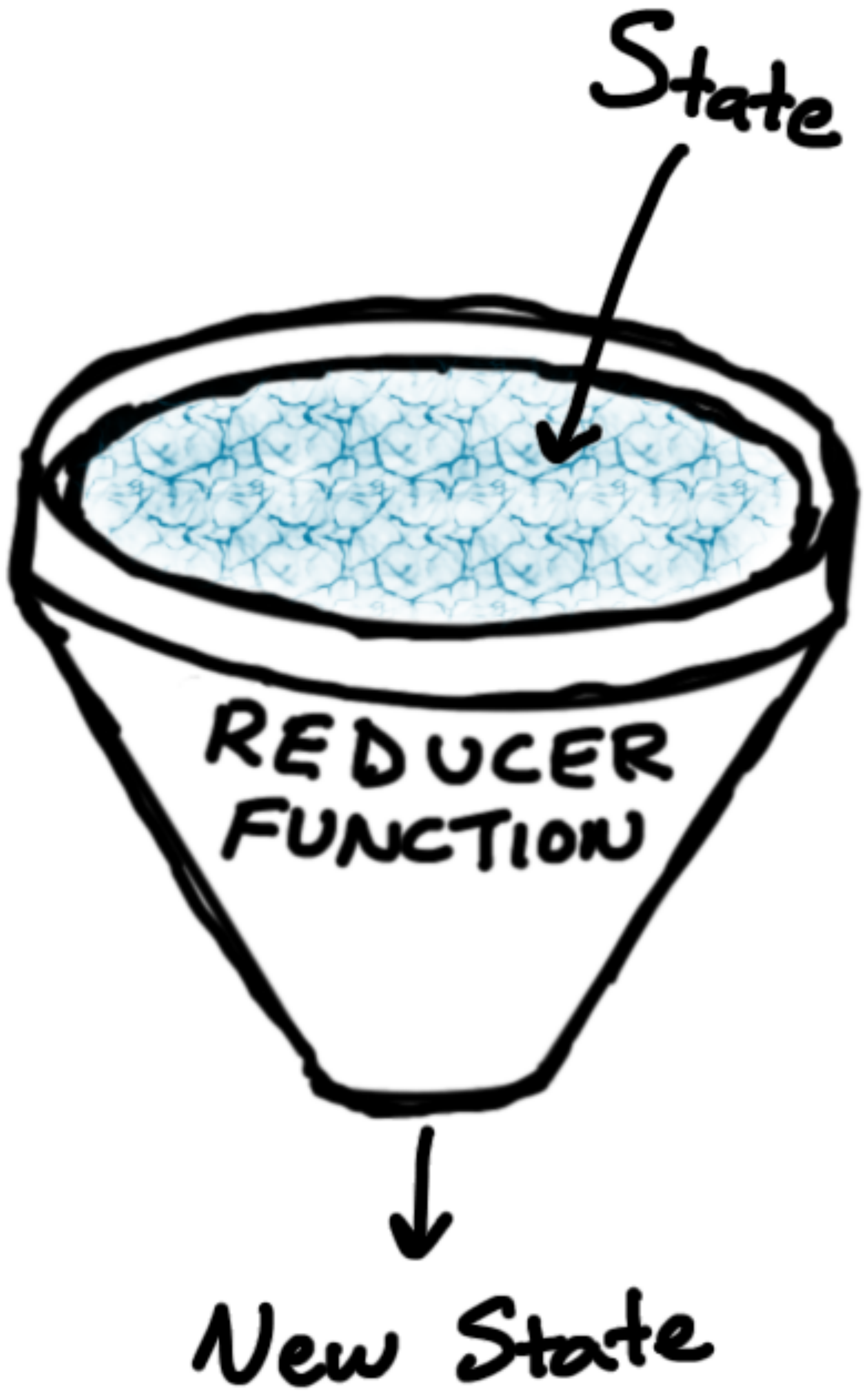
One does not simply modify the state tree.

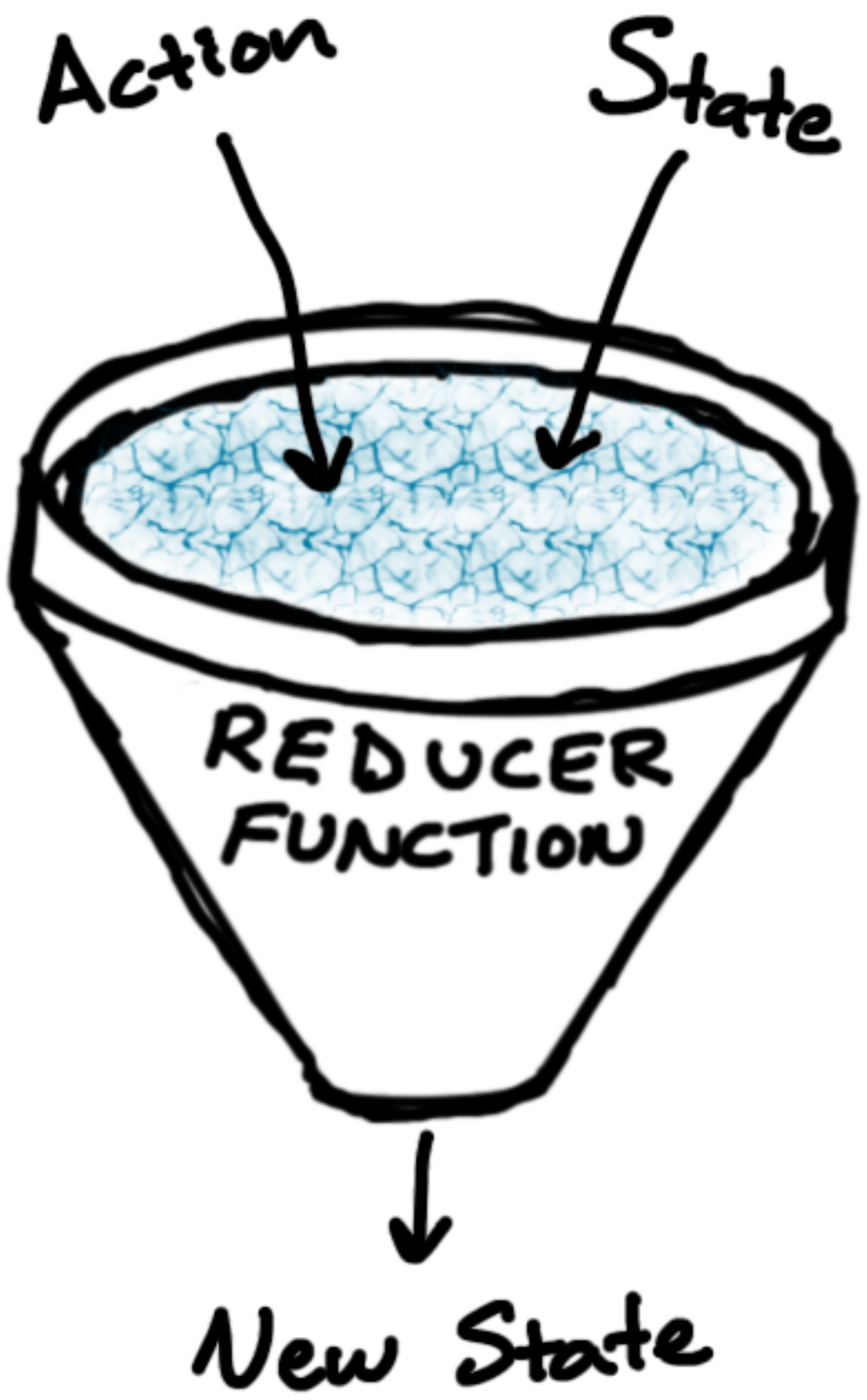
Instead, we dispatch actions.

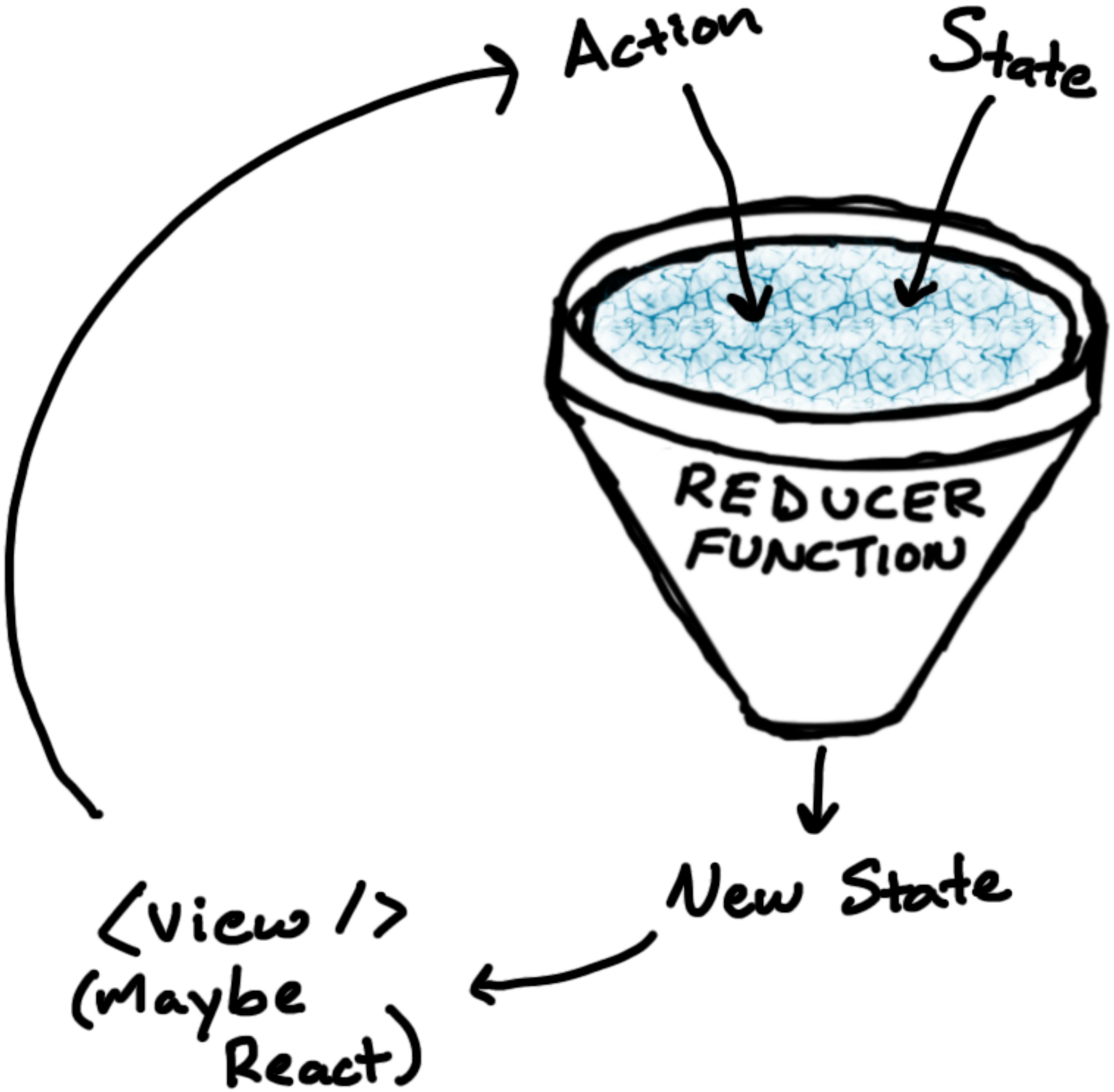
And now: A very scientific illustration.

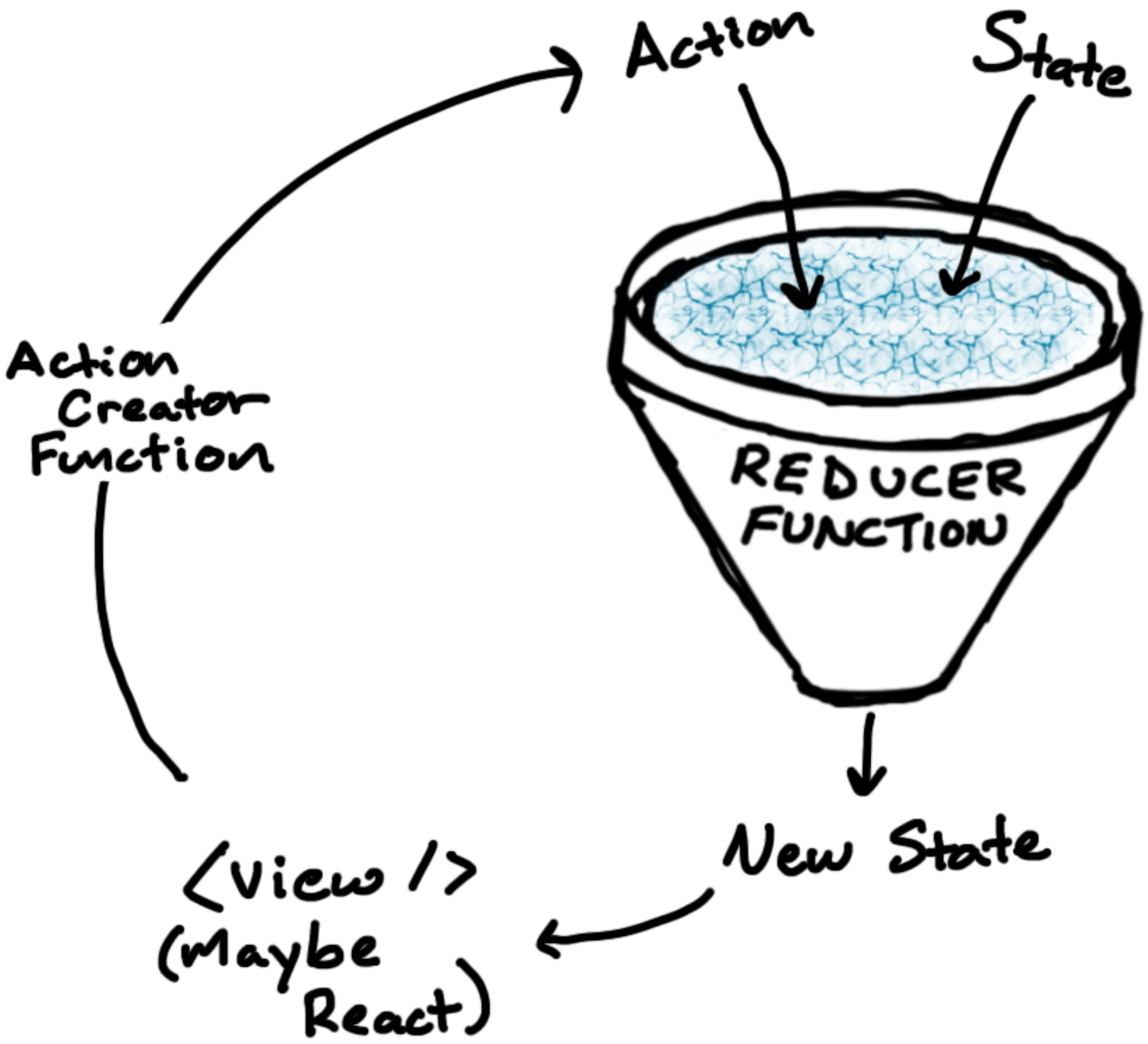

REDUCER
FUNCTION


State
REDUCER FUNCTION


State
REDUCER FUNCTION
New State


Action
State
REDUCER FUNCTION
New State


Action
State
REDUCER FUNCTION
New State
<View />
(Maybe React)


Action
State
REDUCER FUNCTION
New State
<View />
(Maybe React)
Action Creator Function

Redux is *small*.

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

http://bit.ly/redux-fun

Chapter Two

Redux and React

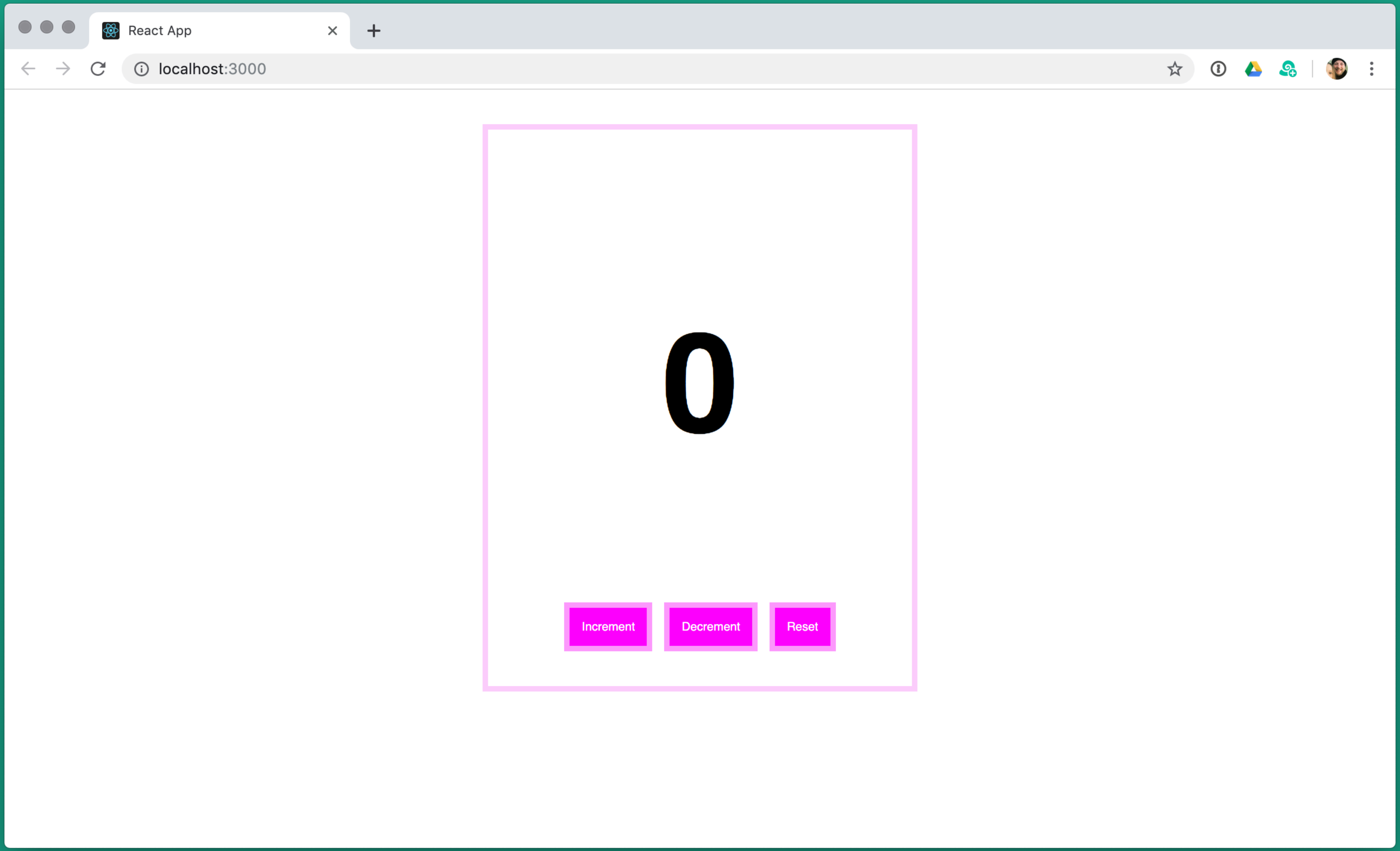
We’re going to do that thing again.

- 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-redux

Let’s do this out of order…

- I’m going to hook Redux up to a React application.
- *Then* we’ll dive into the concepts.


React App
localhost:3000
0
Increment
Decrement
Reset

Exercise

- Clone and install https://github.com/stevekinney/redux-counter.
- I added the ability to increment the counter.
- You’re on the hook to decrement it. Easy peasy. (What does that even mean?)

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

Even *smaller* than Redux!

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

connect();

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

```
const mapDispatchToProps = (dispatch) => {
  return {
    increment() { dispatch(increment()) },
    decrement() { dispatch(decrement()) }
  }
};
```

Remember bindActionCreators?

```
const mapDispatchToProps = (dispatch) ⇒ {
  return bindActionCreators({
    increment,
    decrement
  }, dispatch)
};
```

Even better!

```
const mapDispatchToProps = {
  increment,
  decrement,
};
```

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

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 {
    items: state.items
  }
};
```

This would be the entire state tree.

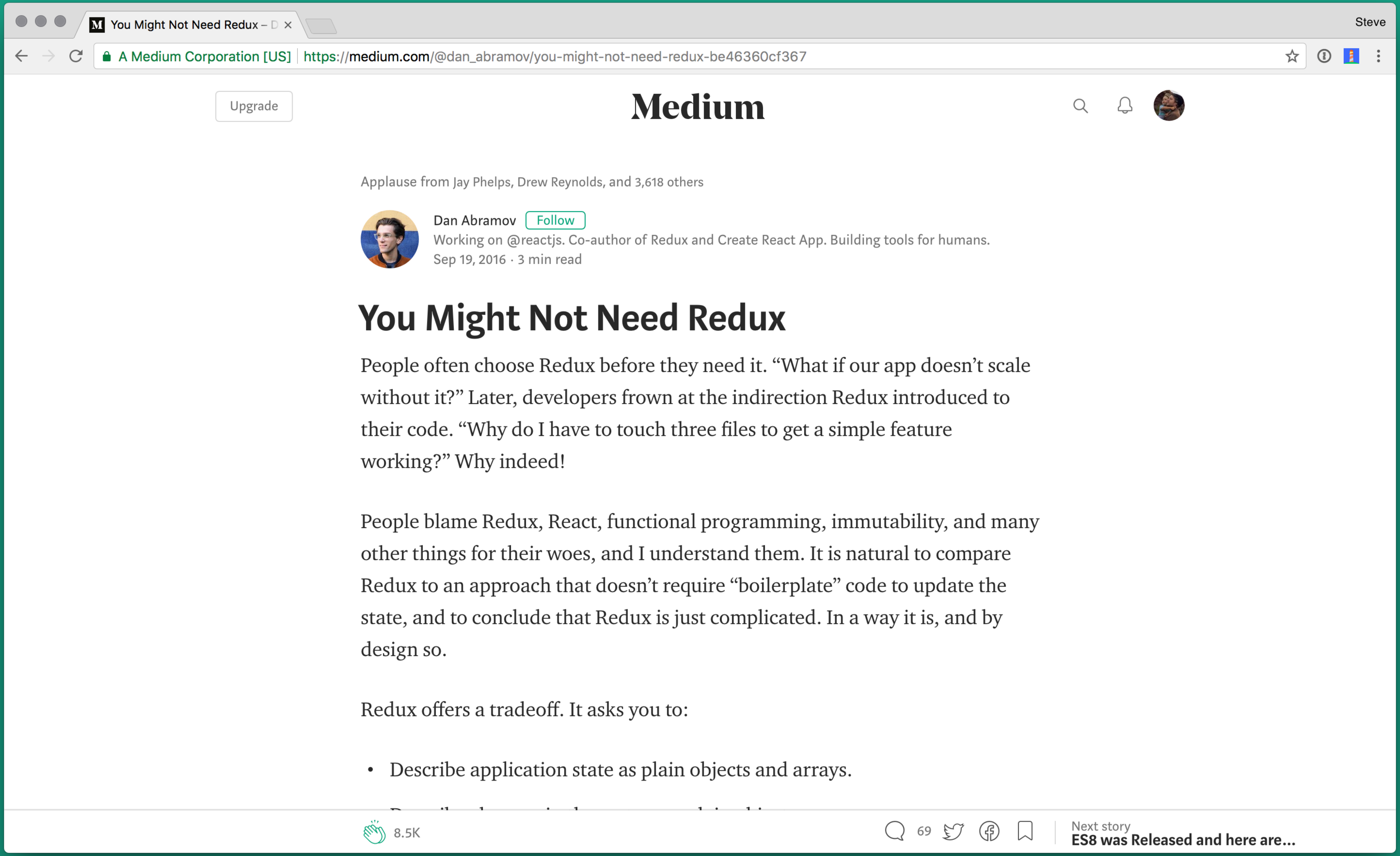
```
const mapStateToProps = (state) => {
  return state;
};
```

This would be just the packed items.

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

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

React State vs. Redux State


You Might Not Need Redux – D
Steve
A Medium Corporation [US]
https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367
Upgrade
Medium
Follow
8.5K
69
Next story
ES8 was Released and here are…

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

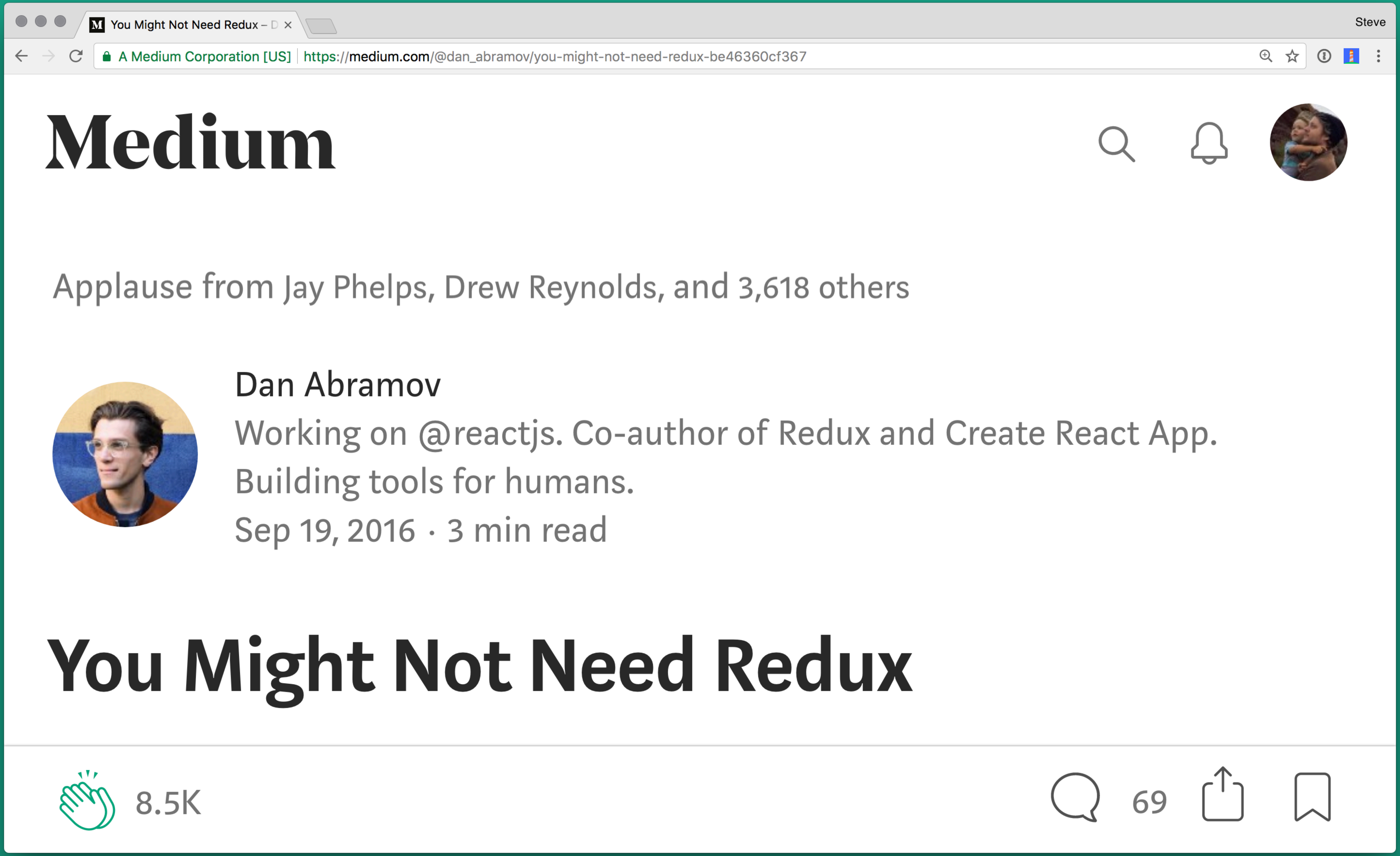
You Might Not Need Redux

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.


You Might Not Need Redux – D

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

69

```
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: uniqueId() });
    this.setState({ value: '' });
  };

  render() { … }
}
```

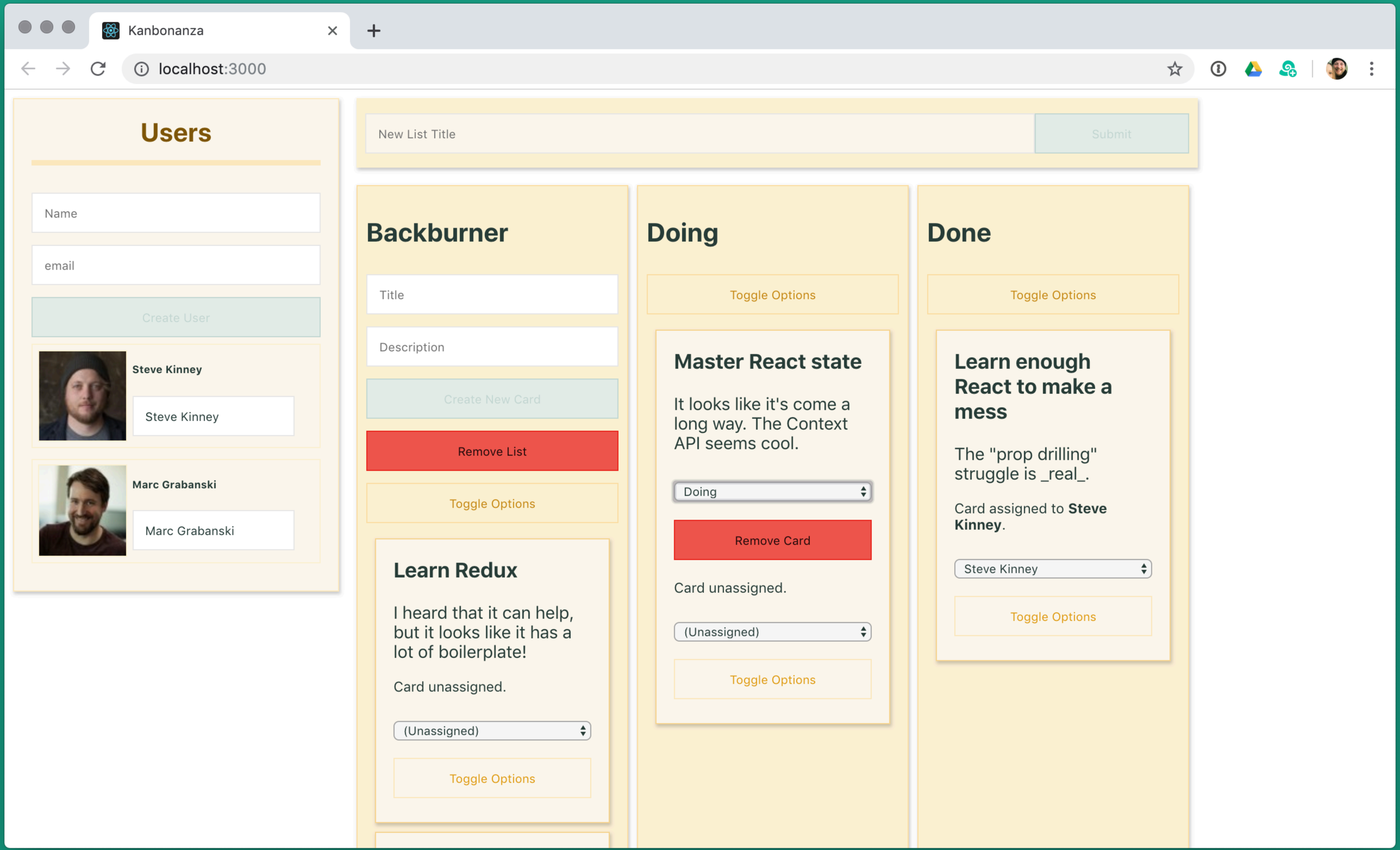
Now, it will be in four files!

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

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

Chapter Three

Normalizing Our Data



Learn Redux

I heard that it can help, but it looks like it has a lot of boilerplate!

Card unassigned.

Master React state

It looks like it's come a long way. The Context API seems cool.

Card unassigned.

Learn enough React to make a mess

The "prop drilling" struggle is _real_.

Card assigned to **Steve Kinney**.

Nota bene: We're going to start from the `redux-basis` branch of https://github.com/stevekinney/kanbananza.

Exercise

- Check out `reducers/cards-reducer.js` and make it look suspiciously like the reducer for lists.
- Hook it into `reducers/index.js`.
- Create a `CardContainer` that looks at `ownProps.cardId` in order grab a card from state.
- In `components/List.js`, map over `list.cards` in order to create a `CardContainer` for each ID in the array.

Exercise

- I implemented the ability to create a card.
- Your job is to implement the same for creating a list.

Exercise

- Refactor card creation to use our handy new abstraction.
- Here is a **hint**: take some inspiration from what we just did with lists.

Exercise

- This should be old hat at this point, but we want to wire up the `Users` component and the `User` component.

Exercise

- Alright—you’re going to create a new user.

Chapter Four

Selectors and Reselect

Live Coding

Let’s say I did this refactor...

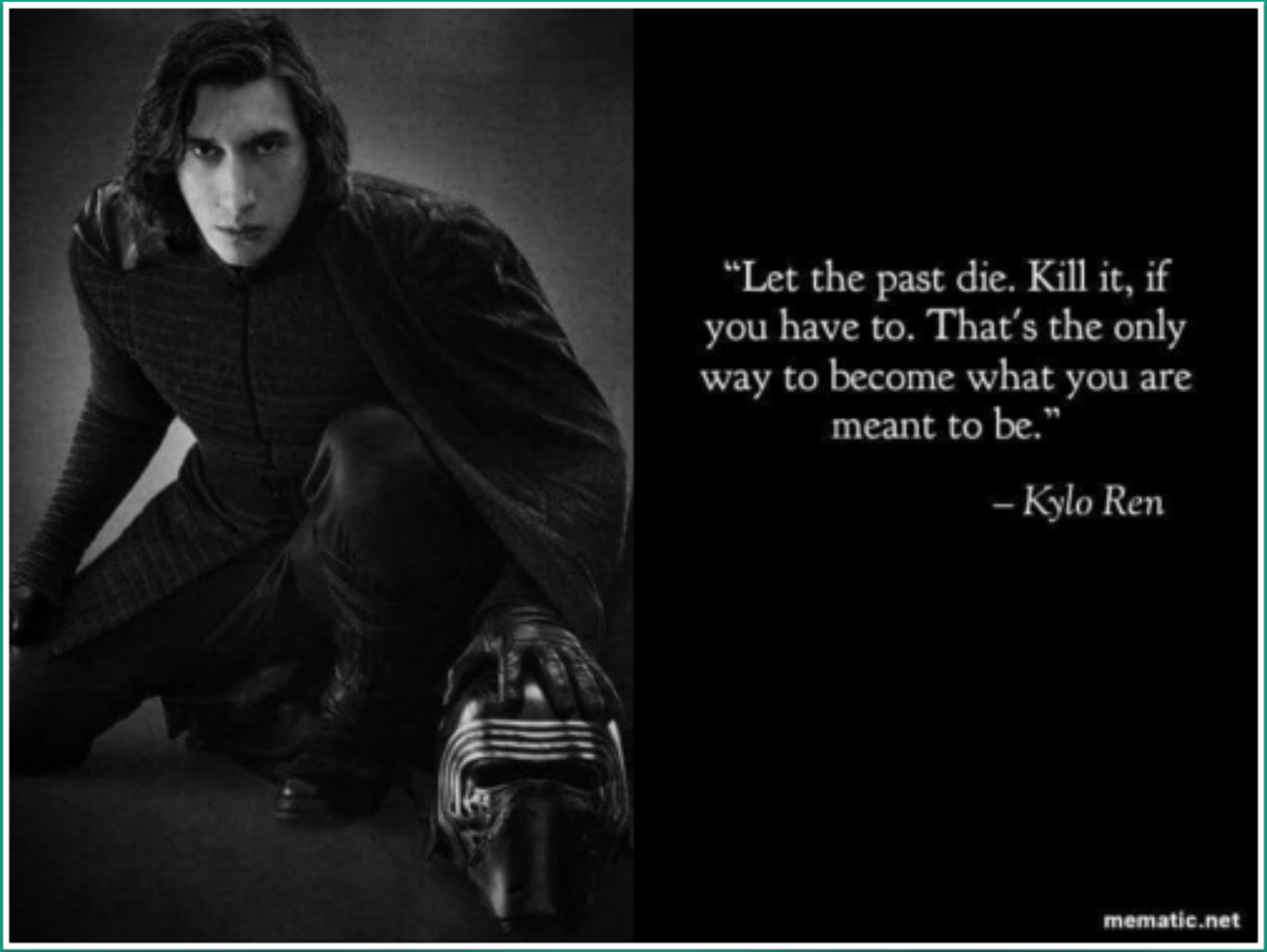
```javascript
import { connect } from 'react-redux';
import Users from '../components/Users';

const getUsers = state => {
  console.log('getUsers', state.users.ids);
  return state.users.ids;
};

const mapStateToProps = state => {
  return { users: getUsers(state) };
};

export default connect(mapStateToProps)(Users);
```

Exercise

- Why are the users reloading when I change a card?
- Nothing changed with the users!
- Can you implement a selector to stop this tomfoolery?

An aside: Implementing Undo and Redo

Holding onto the past, present, and future.



"Let the past die. Kill it, if you have to. That's the only way to become what you are meant to be."

– *Kylo Ren*

```
if (action.type === ADD_NEW_ITEM) {
  const { item } = action.payload;
  return {
    past: [present, ...past],
    present: [...present, item],
    future,
  };
}
```

```
if (action.type === UNDO_ITEM_ACTION) {
  if (!past.length) return state;
  const newFuture = [ present, ...future ];
  const [ newPresent, ...newPast ] = past;
  return {
    past: newPast,
    present: newPresent,
    future: newFuture
  }
}
```

```
if (action.type === REDO_ITEM_ACTION) {
  if (!future.length) return state;
  const [newPresent, ...newFuture] = future;
  const newPast = [ present, ...past ];
  return {
    past: newPast,
    present: newPresent,
    future: newFuture
  }
}
```

Chapter Five

Redux Thunk

Thunk?

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

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

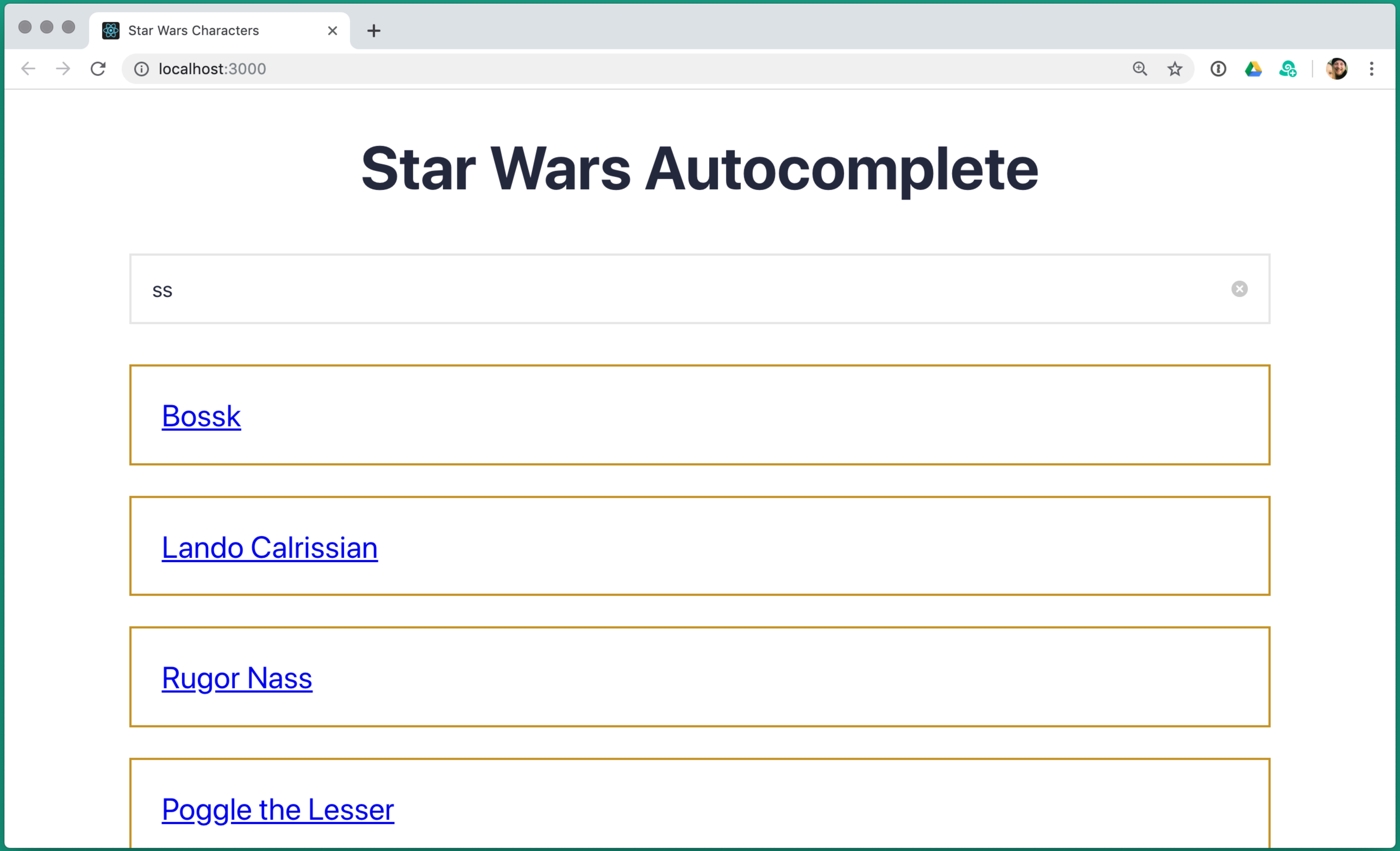
The major idea behind a thunk is that its code to be executed later.

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

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,
      });
    });
  };
};
```



Star Wars Autocomplete

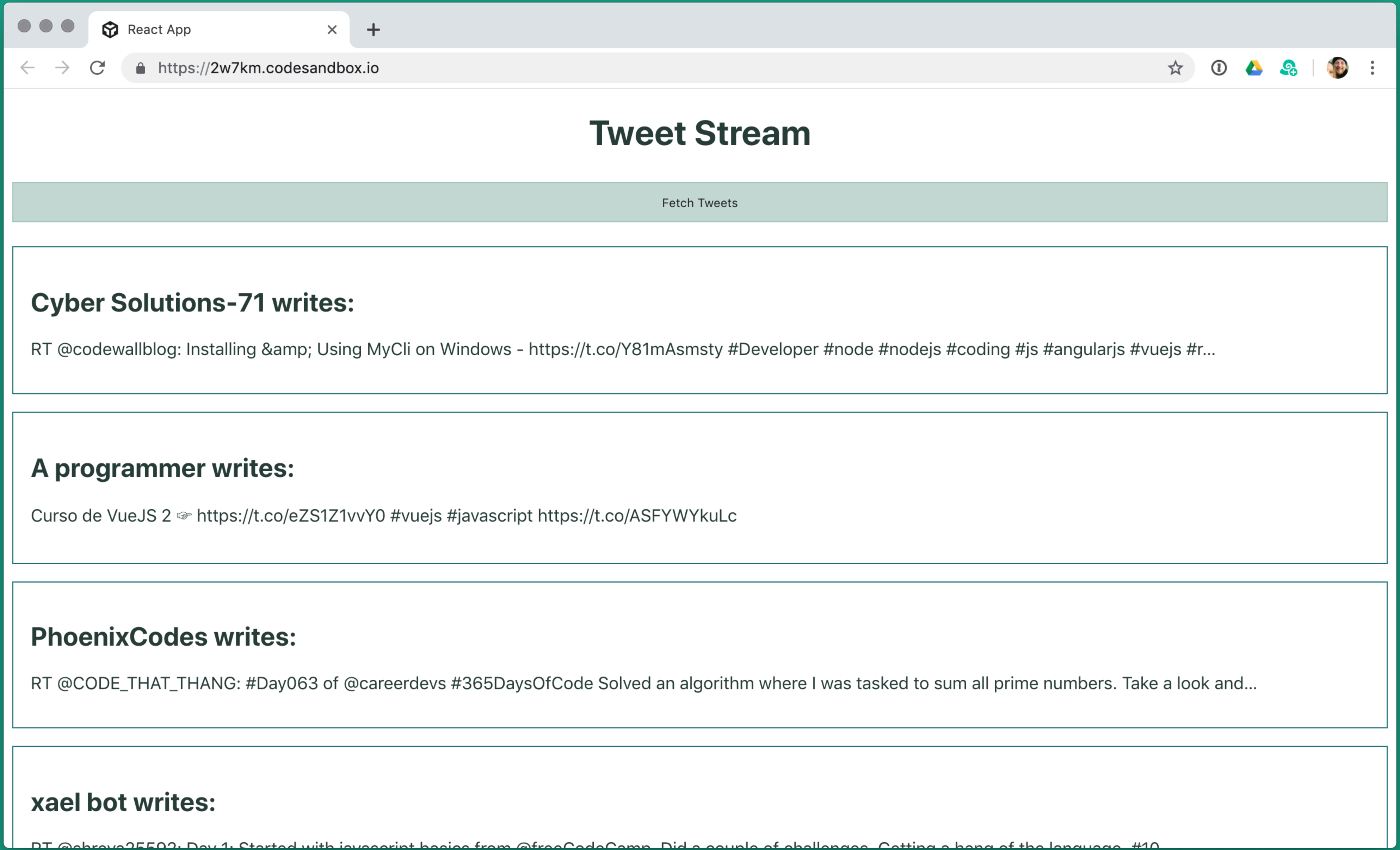
ss

Bossk

Lando Calrissian

Rugor Nass

Poggle the Lesser



Tweet Stream

Fetch Tweets

Cyber Solutions-71 writes:

RT @codewallblog: Installing &amp; Using MyCli on Windows - https://t.co/Y81mAsmsty #Developer #node #nodejs #coding #js #angularjs #vuejs #r...

A programmer writes:

Curso de VueJS 2 👉 https://t.co/eZS1Z1vvY0 #vuejs #javascript https://t.co/ASFYWYkuLc

PhoenixCodes writes:

RT @CODE_THAT_THANG: #Day063 of @careerdevs #365DaysOfCode Solved an algorithm where I was tasked to sum all prime numbers. Take a look and...

xael bot writes:

RT @shreya25592: Day 1: Started with javascript basics from @freeCodeCamp. Did a couple of challenges. Getting a hang of the language. #10

Exercise

- Implement Redux Thunk in order to dispatch a function that will in tern dispatch an action when we hear back from the API.
- Your humble instructor is not responsible for whatever tweets have the word JavaScript in them.

Chapter Seven

Redux Observable

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

```javascript
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.

And now: Just enough RxJS to get yourself in trouble.

What is an observable?

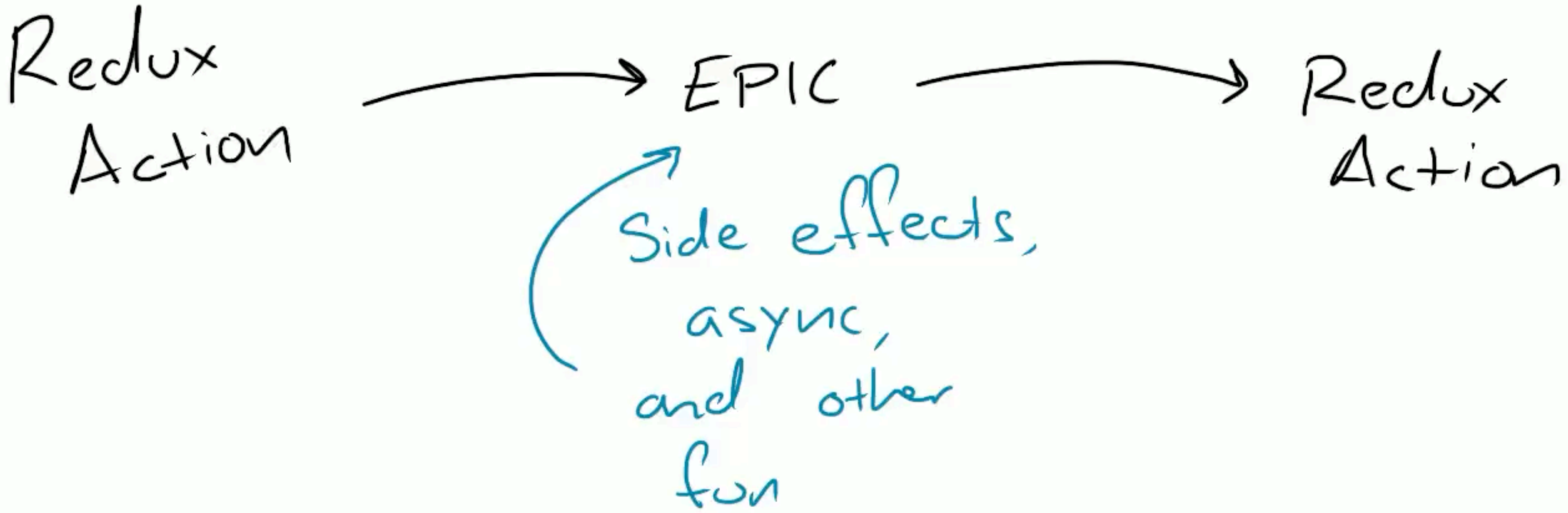
- A stream of zero, one, or more values.
- The stream comes in over a series of time.
- The stream is cancelable.

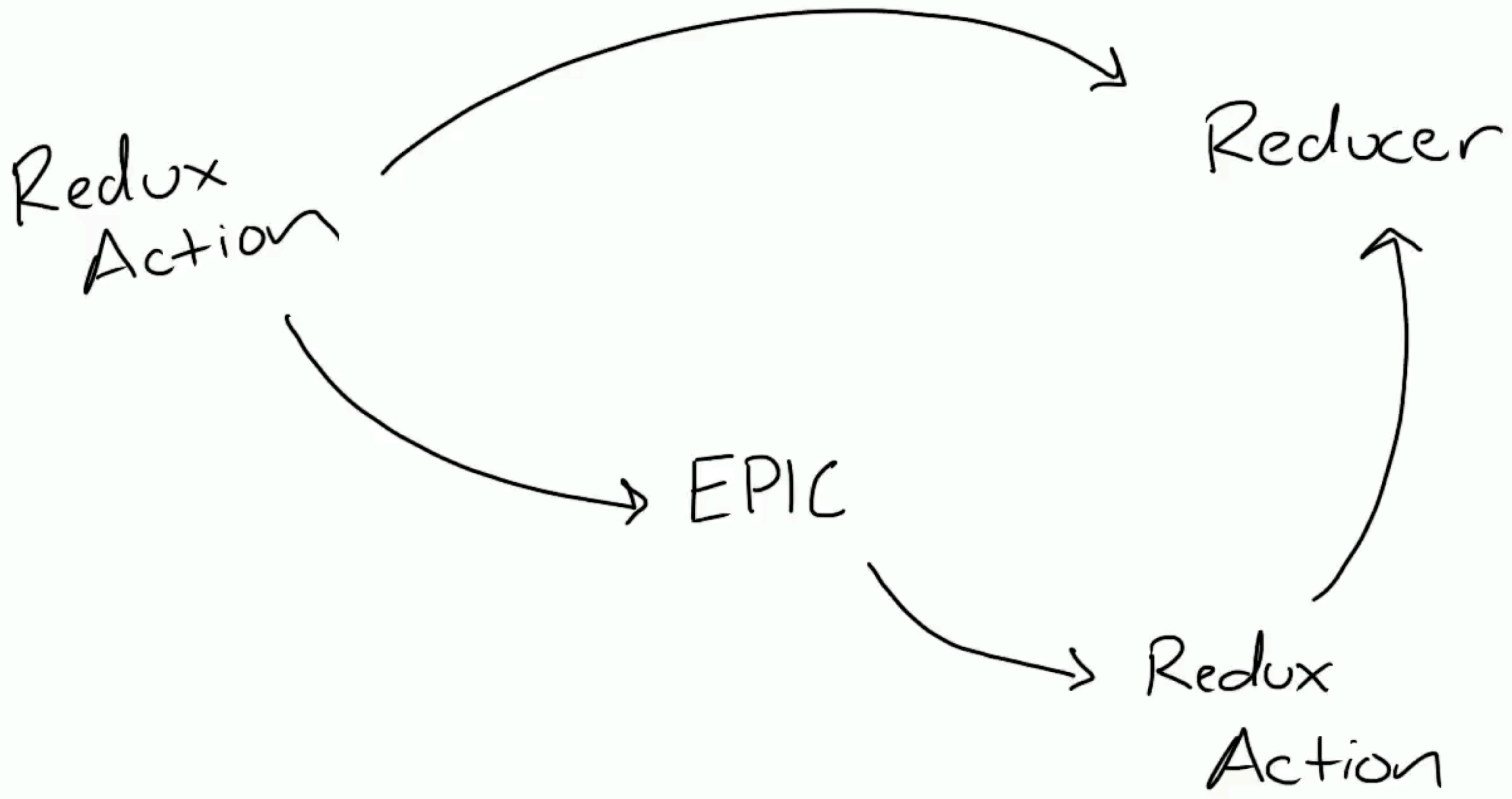
What is Redux Observable?

- Redux Observable is a combination of RxJS and Redux.
- Side effect managment using "epics."

What is an epic? 🙄

- A function that takes a stream of all actions dispatched and returns a stream of new actions to dispatch.


Redux Action
EPIC
Redux Action
Side effects, async, and other fun

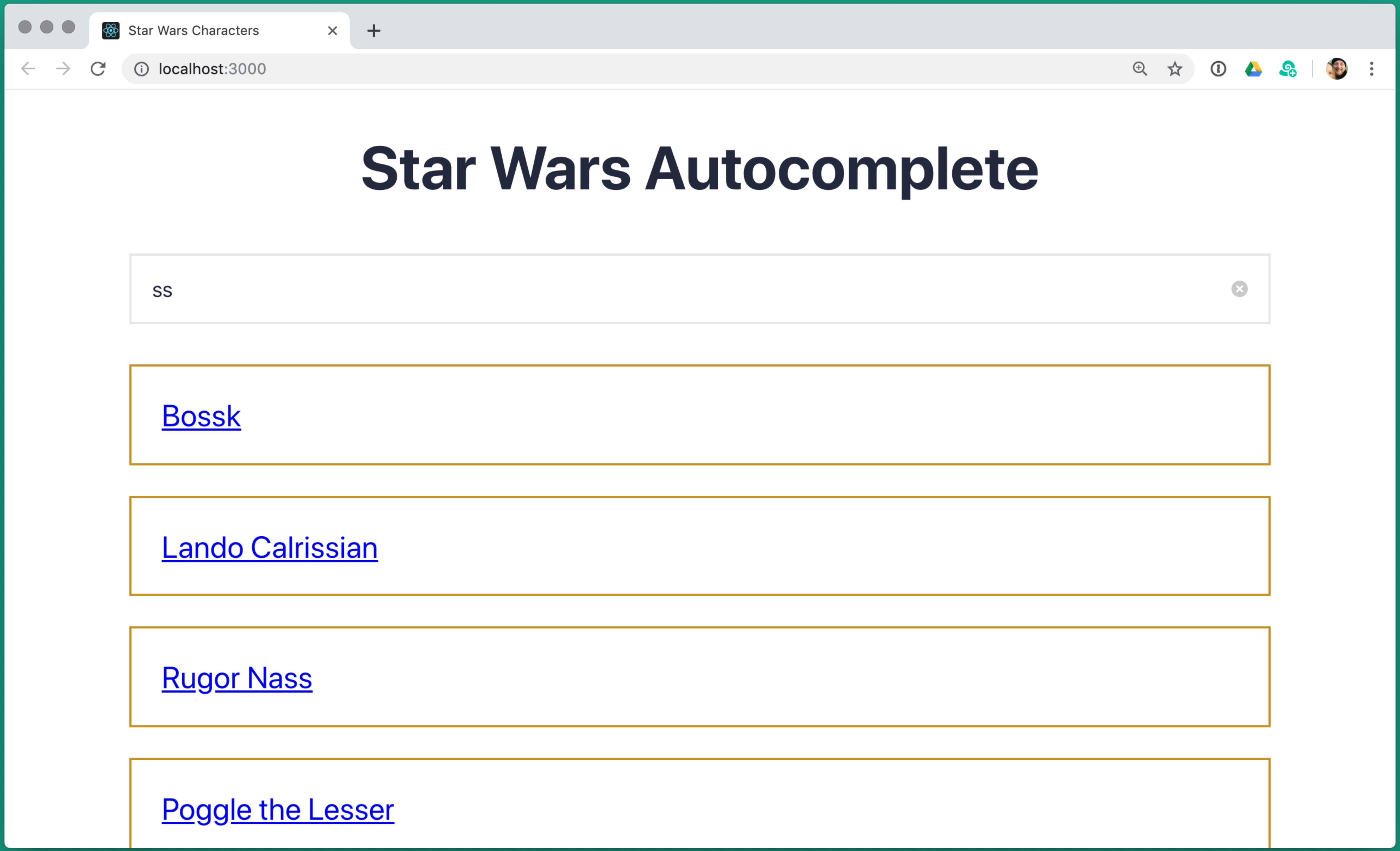

Redux Action
Reducer
EPIC
Redux Action

The Basic Example

```
const pingPong = (action, store) => {
  if (action.type === ‘PING’) {
    return {
      type: ‘PONG’
    };
  }
};
```

The Basic Example

```
const pingPongEpic = (action$, store) ⇒
  action$.ofType(‘PING’)
    .map(action ⇒ ({ type: ‘PONG’ }));
```



Star Wars Autocomplete

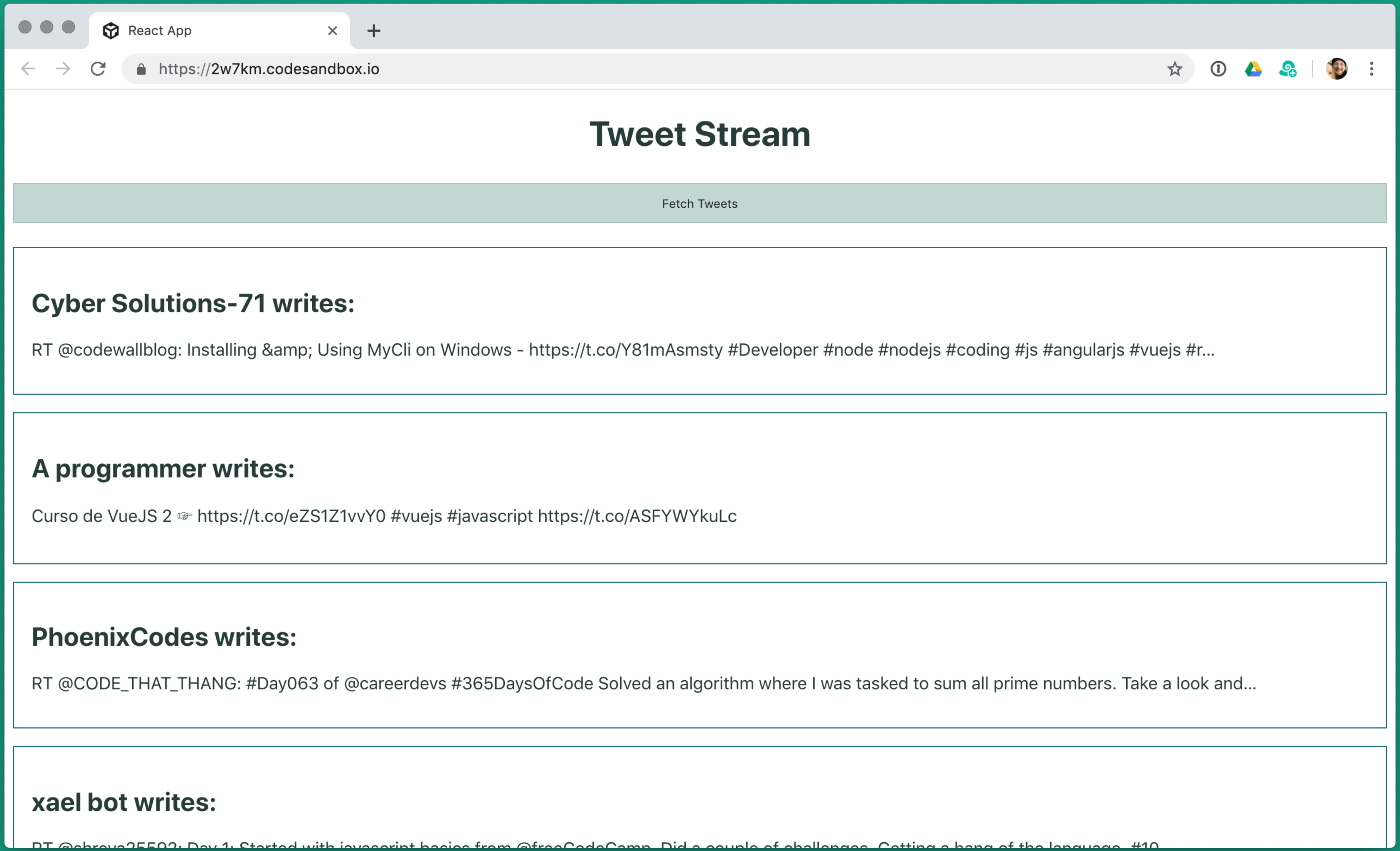
ss

Bossk

Lando Calrissian

Rugor Nass

Poggle the Lesser



React App

https://2w7km.codesandbox.io

Tweet Stream

Cyber Solutions-71 writes:

RT @codewallblog: Installing &amp; Using MyCli on Windows - https://t.co/Y81mAsmsty #Developer #node #nodejs #coding #js #angularjs #vuejs #r...

A programmer writes:

Curso de VueJS 2 👉 https://t.co/eZS1Z1vvY0 #vuejs #javascript https://t.co/ASFYWYkuLc

PhoenixCodes writes:

RT @CODE_THAT_THANG: #Day063 of @careerdevs #365DaysOfCode Solved an algorithm where I was tasked to sum all prime numbers. Take a look and...

xael bot writes:

Exercise

- Implement Redux Observable in order to dispatch a function that will in tern dispatch an action when we hear back from the API.
- **Again**—Your handsome instructor is not responsible for whatever tweets have the word JavaScript in them.

Lodash for async. — Ben Lesh, probably.

Chapter Eight

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}`;
  }
}
```

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

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

Much Better! 😎

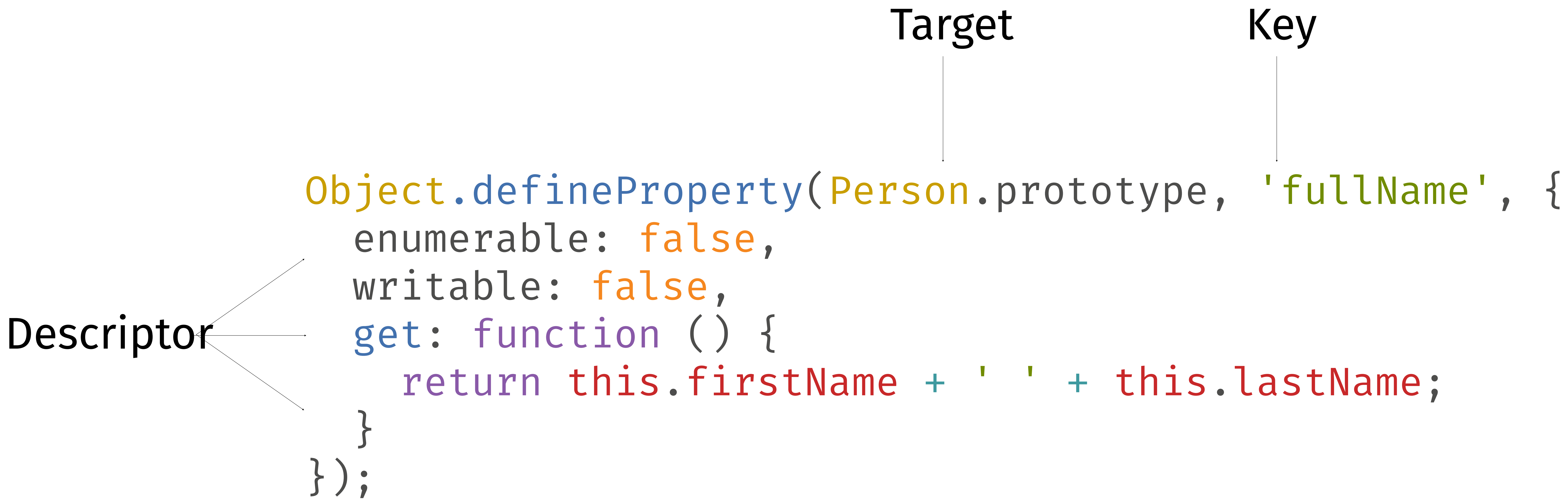
Getters and setters may seem like some fancy new magic, but they’ve been around since ES5.

Not as 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.



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

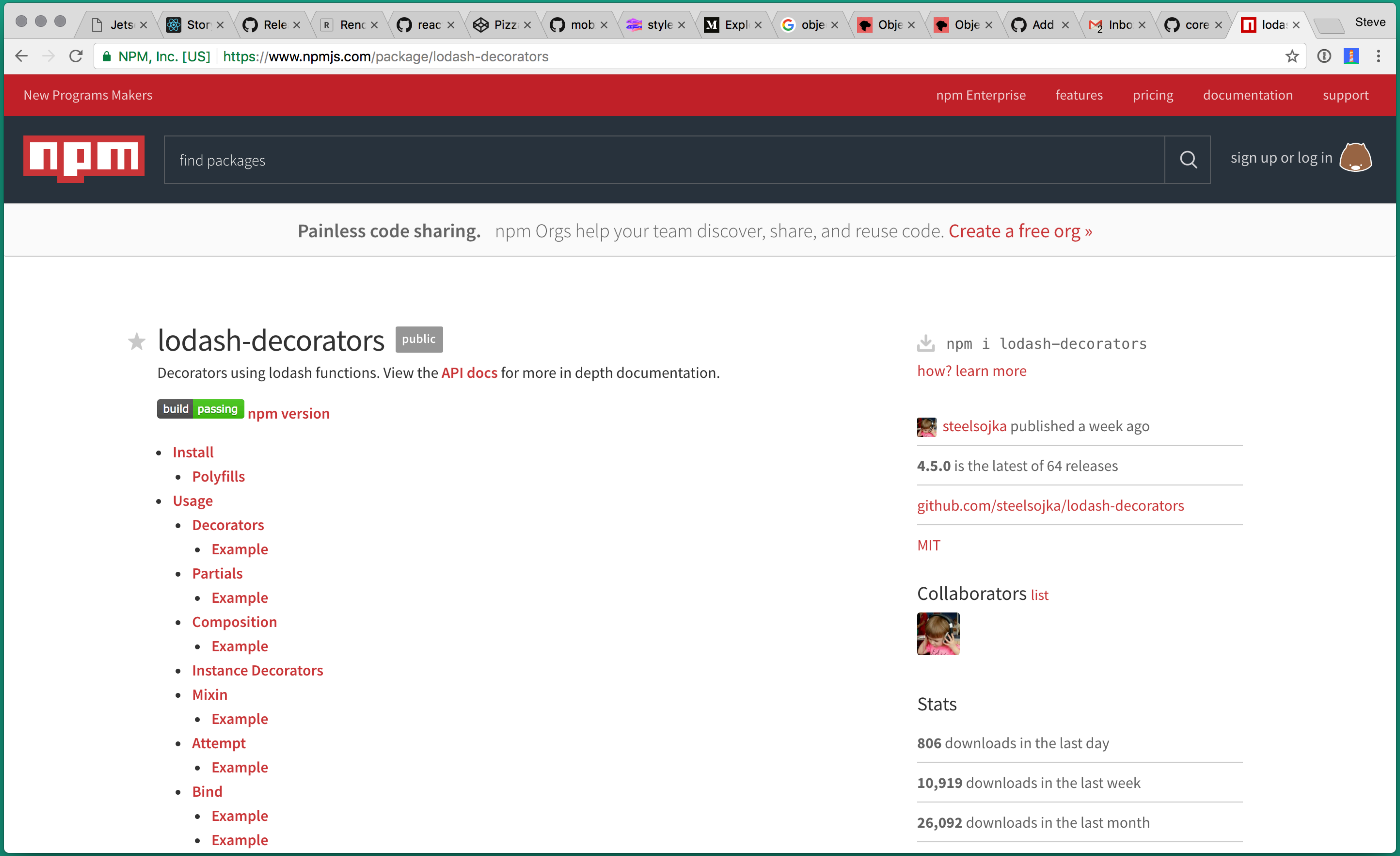
```
function decoratorName(target, key, descriptor) {
  // …
}
```

```
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
```



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

MIT

Collaborators list

Stats

806 downloads in the last day

10,919 downloads in the last week

26,092 downloads in the last month

A big problem with decorators is that they aren’t exactly “real.”

Okay, so… MobX

Imagine if you could simply change your objects.

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 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.

Epilogue

Closing Thoughts

MobX versus Redux

~~MobX versus Redux~~

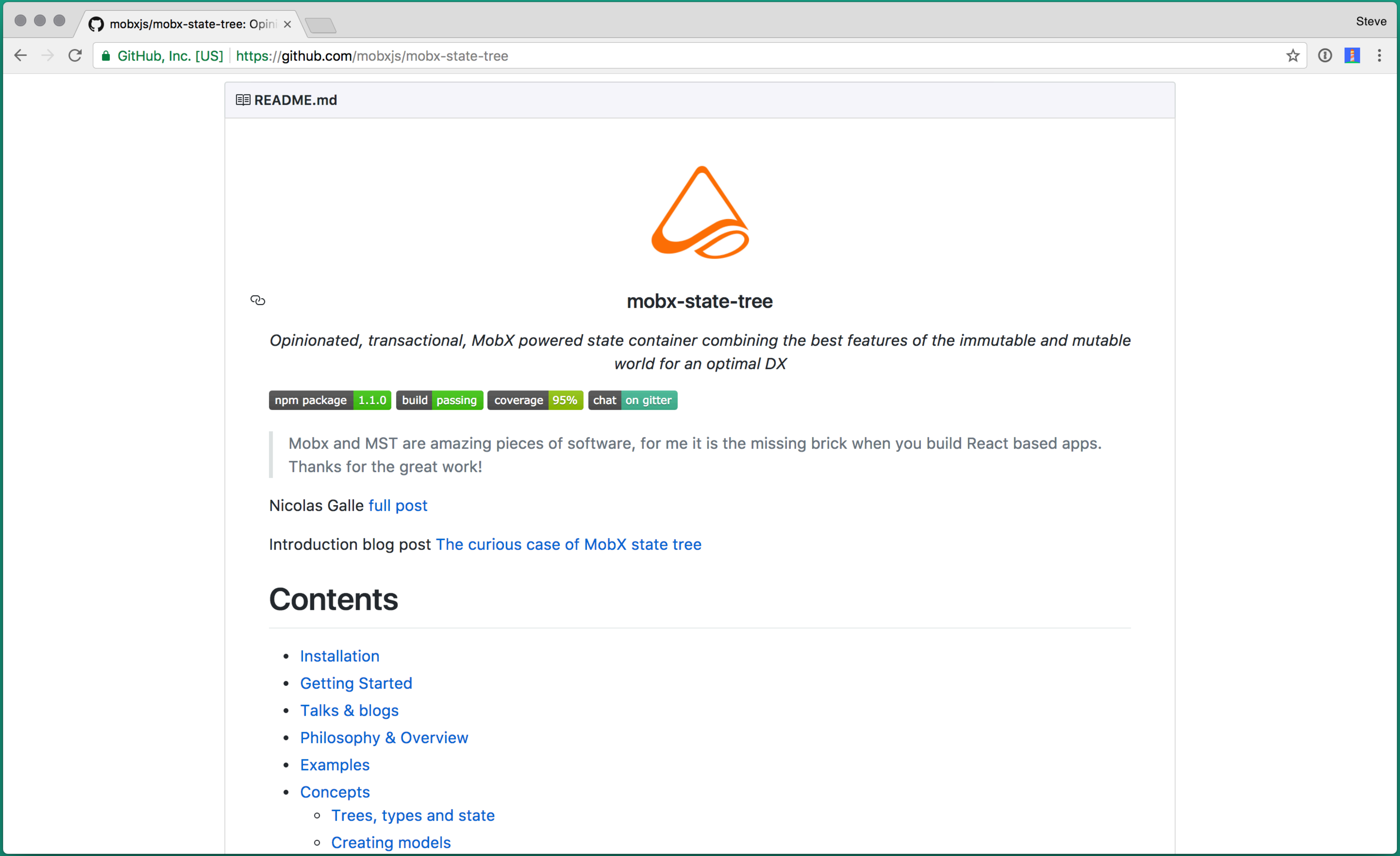
Dependency Graphs versus Immutable State Trees

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


mobxjs/mobx-state-tree: Opini
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

```
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] }
  }
}
```

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 observables?

Can you implement undo with API requests?

You now have a good sense of the lay o’ the land.

Questions?