State Management in React, v2

Steve Kinney

A Frontend Masters Workshop

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

twilio

This a course for keeping your state manageable when it’s no longer a toy application.

In this course, we'll be working with pure React.

So, what are we going to do today?

- Think deeply about what “state” even means in a React application.

- Learn a bit about the inner workings of `this.setState`.
  - How class-based component state and hooks differ.
  - Explore APIs for navigating around prop-drilling.
  - Use reducers for advanced state management.

So, what are we going to do today?

- Write our own custom hooks for managing state.
- Store state in Local Storage.
- Store state in the URL using query parameters.
- Fetch state from a server—because that's a thing.

And now...

Understanding State

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

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.

- **Model state**: This is likely the data in your application. This could be the items in a given list.
- **Ephemeral state**: Stuff like the value of an input field that will be wiped away when you hit “enter.” This could be the order in which a given list is sorted.

Spoiler alert: There is no silver bullet. 😬

And now...

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 to get warmed up.
- https://github.com/stevekinney/simple-counter-react-state
- It will have three buttons:
  - Increment
  - Decrement
  - Reset
- **Your job**: Get decrement and reset working.
- 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 });

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

Any guesses?

0

this.setState() is asynchronous.

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 });
  }

  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.

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

```
const newState = {
    ...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() { … }
}
```

3

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

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

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

```
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() { … }
}
```

 Live Coding 

`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() { … }
}
```

 Live Coding 

Patterns and anti-patterns

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

Twitter, Inc. twitter.com/dan_abramov/status/749710501916139520?r

Home

Tweet

Dan Abram

@dan_abramov

Working on @reac
and Create React
humans.

London, Engla

Joined August

Dan Abramov
@dan_abramov

Following

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>
    );
  }
}
```

```
const renderUserProfile = user ⇒ {
  return (
    <UserProfile
      key={user.id}
      photograph={user.mugshot}
      onLayoff={handleLayoff}
    />
  );
};
```

```
const UserList = ({ users }) ⇒ {
  return (
    <section>
      <VeryImportantUserControls />
      {users.map(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 ] });
        });
      }, 10000)
    }
  }

  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 }
}
```

And now...

An Equally Uncomfortably Close Look at React Hooks

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

const increment = () => setCount(count + 1);
const decrement = () => setCount(count - 1);
const reset = () => setCount(0);
```

```
const increment = () ⇒ {
  setCount(count + 1);
  setCount(count + 1);
};
```

```
const increment = () => {
  setCount(c => c + 1);
};
```

```
const increment = () => {
  setCount(c => c + 1);
  setCount(c => c + 1);
};
```

```
setCount(c ⇒ {
  if (c ⩾ max) return;
  return c + 1;
});
```

```
setCount(c ⟹ {
  if (c ⩾ max) return c;
  return c + 1;
});
```

 Live Coding 

Exercise

- Can you add a second effect that updates the document's title whenever the count changes?

- (Hint: `document.title` is your friend here.)

 Live Coding 

How do lifecycle methods and hooks differ?

```
componentDidUpdate() {
  setTimeout(() ⇒ {
    console.log(`Count: ${this.state.count}`);
  }, 3000);
}
```

```
React.useEffect(() ⇒ {
  setTimeout(() ⇒ {
    console.log(`Count: ${count}`);
  }, 3000);
}, [count]);
```

```
const countRef = React.useRef();
countRef.current = count;

React.useEffect(() => {
  setTimeout(() => {
    console.log(`You clicked ${countRef.current} times`);
  }, 3000);
}, [count]);
```

 Live Coding 

Cleaning Up After `useEffect`

```
useEffect(() ⇒ {
  let x;
  const id = setInterval(() ⇒ {
    console.log(x++);
  }, 3000);
  return () ⇒ {
    clearInterval(x++);
  }
});
```

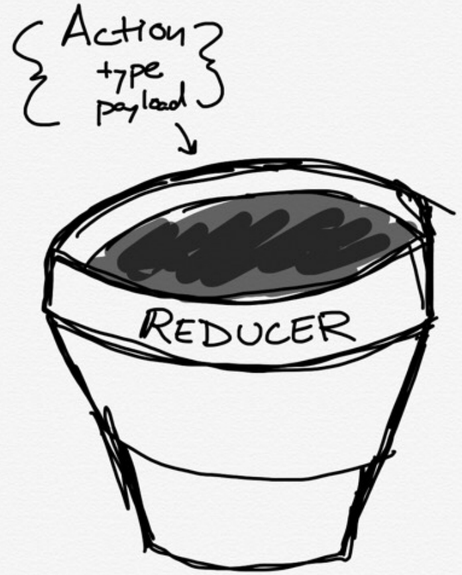
 Live Coding 

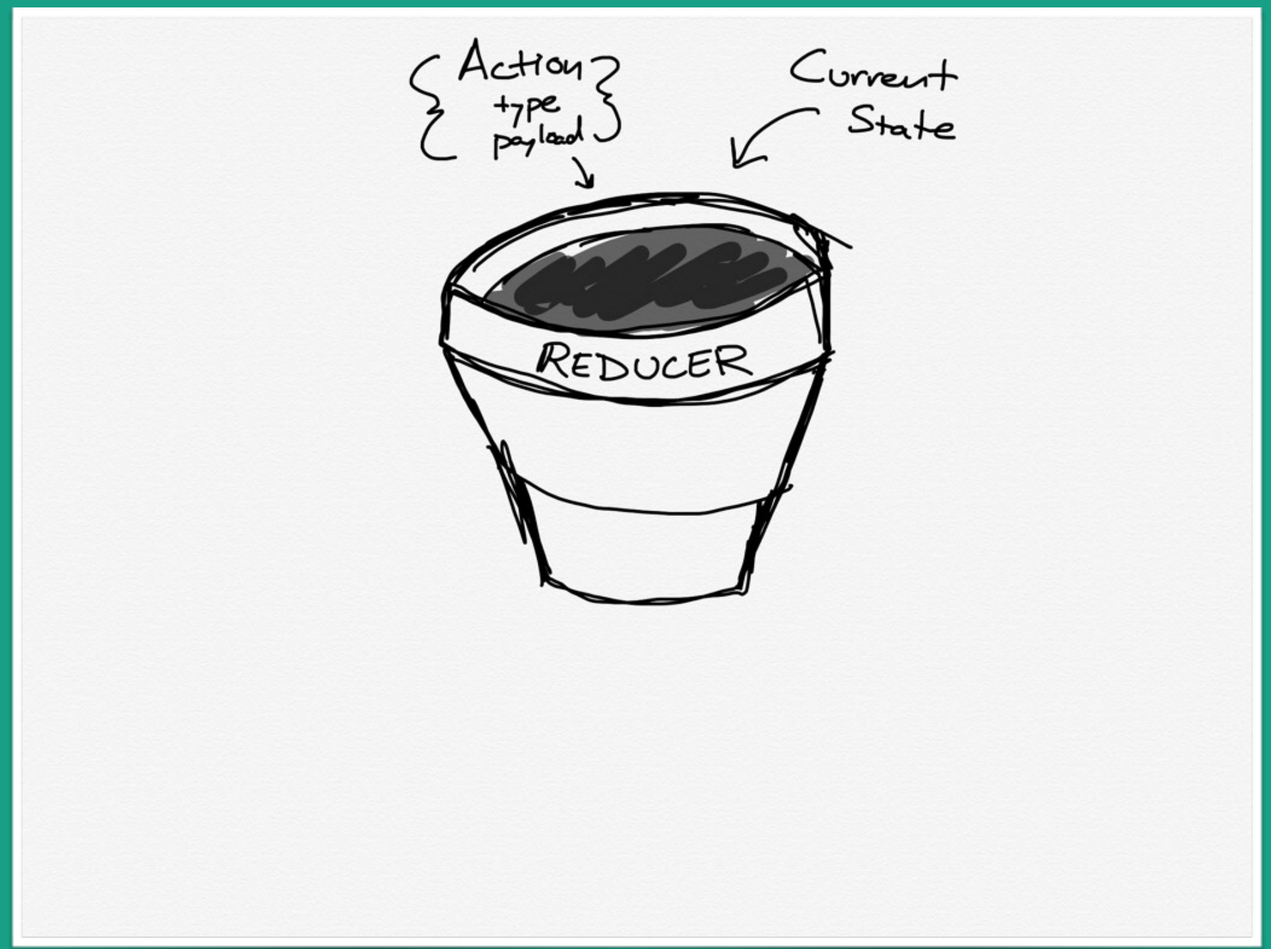
And now...

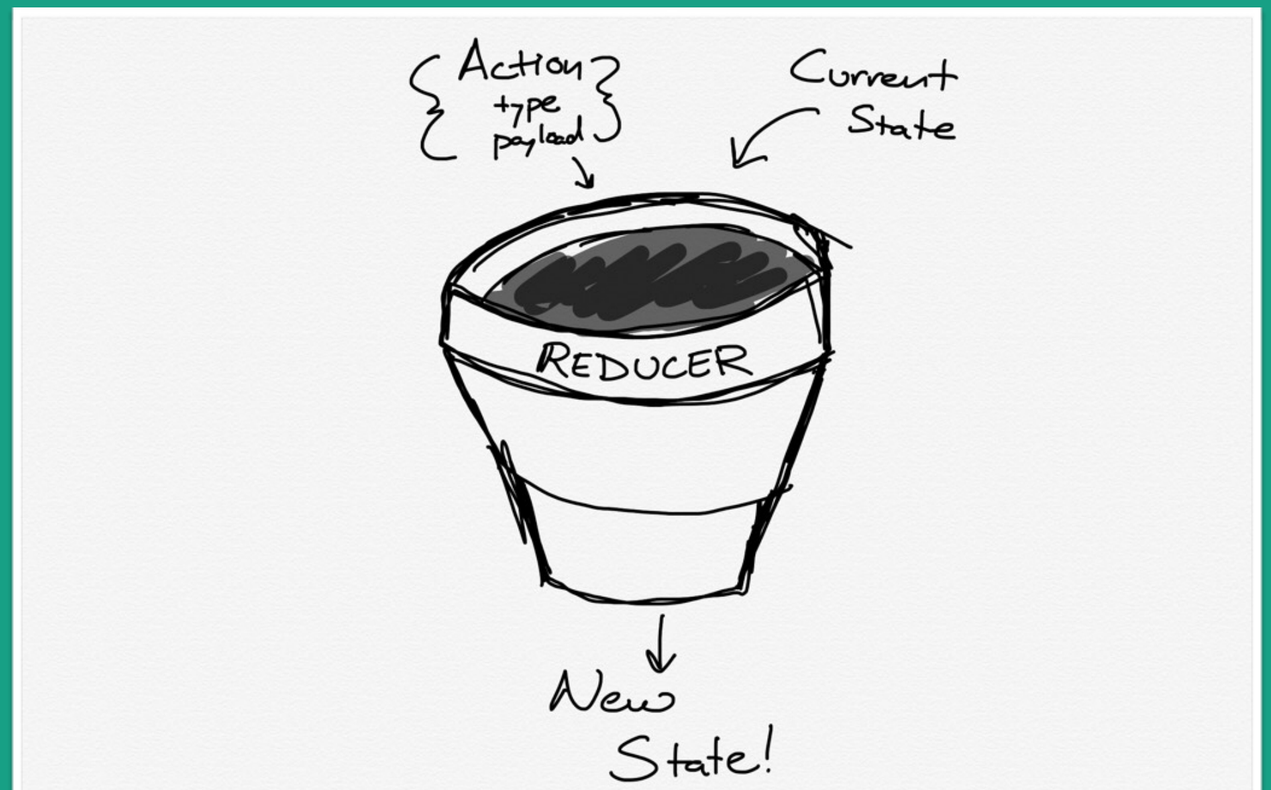
The Joy of `useReducer`

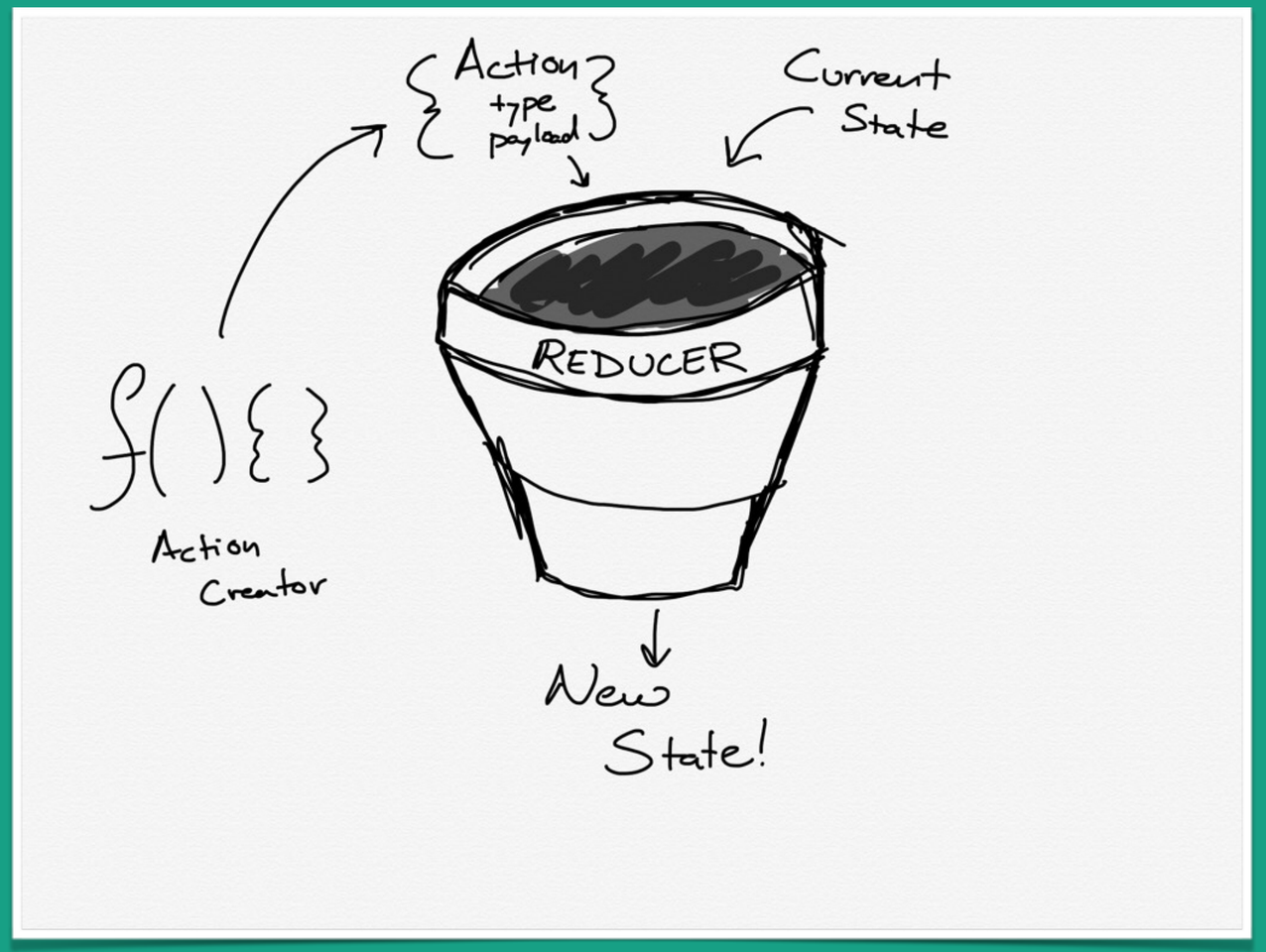
Introducing Grudge List

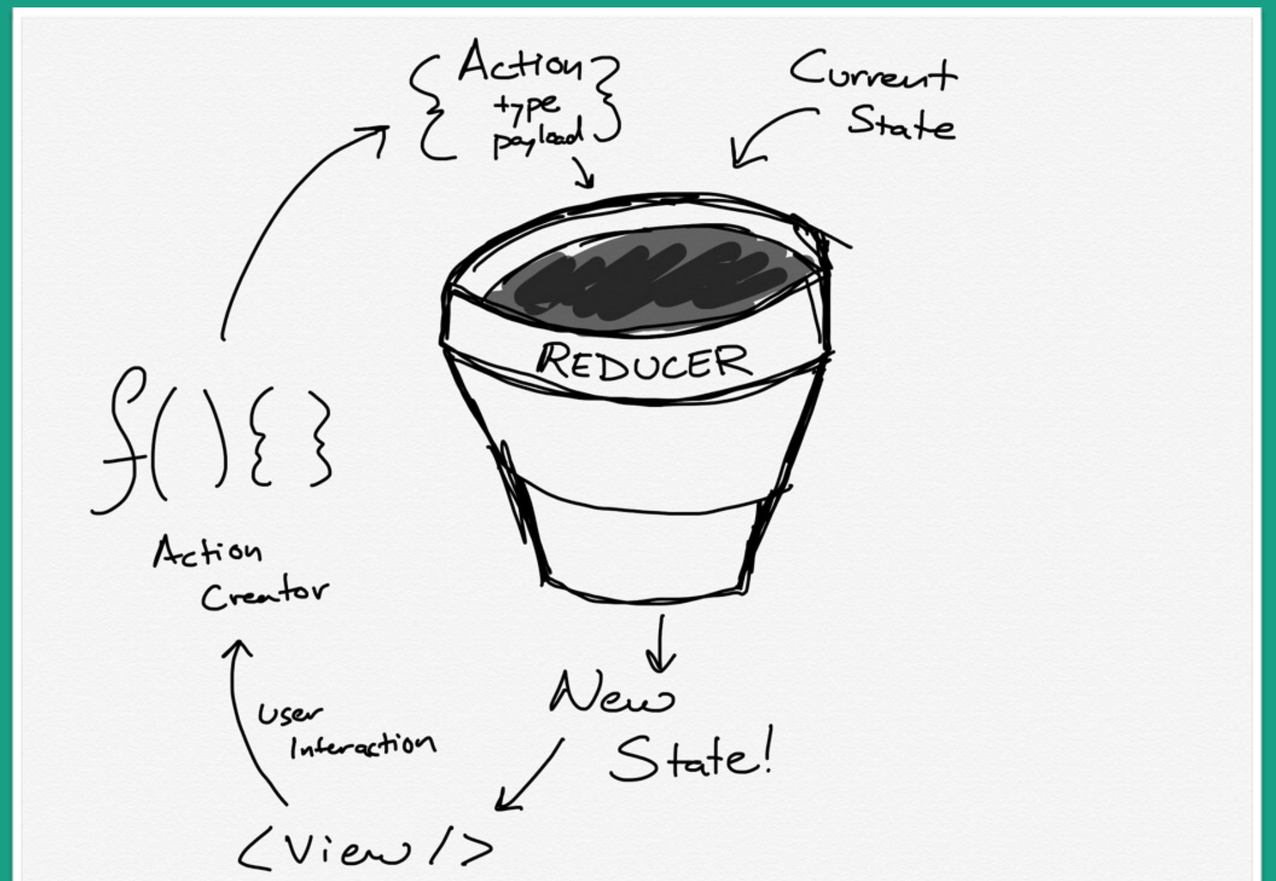

REDUCER


Action
type
payload
REDUCER


Action
type
payload
Current
State
REDUCER


Action
type
payload
Current
State
REDUCER
New
State!


Action
type
payload
Current
State
REDUCER
f( ){}
Action
Creator
New
State!


Action
type
payload
Current
State
REDUCER
f( ){}
Action
Creator
User
Interaction
New
State!
<View />

useReducer()

What's the deal with `useReducer()`?

- So, it turns out that it has nothing to do with Redux.
- But, it *does* allow you to use reducers—just like Redux.
- The cool part is that it allows you to create interfaces where you (or a friend) can pass in the mechanics about how to update state.

 Live Coding 

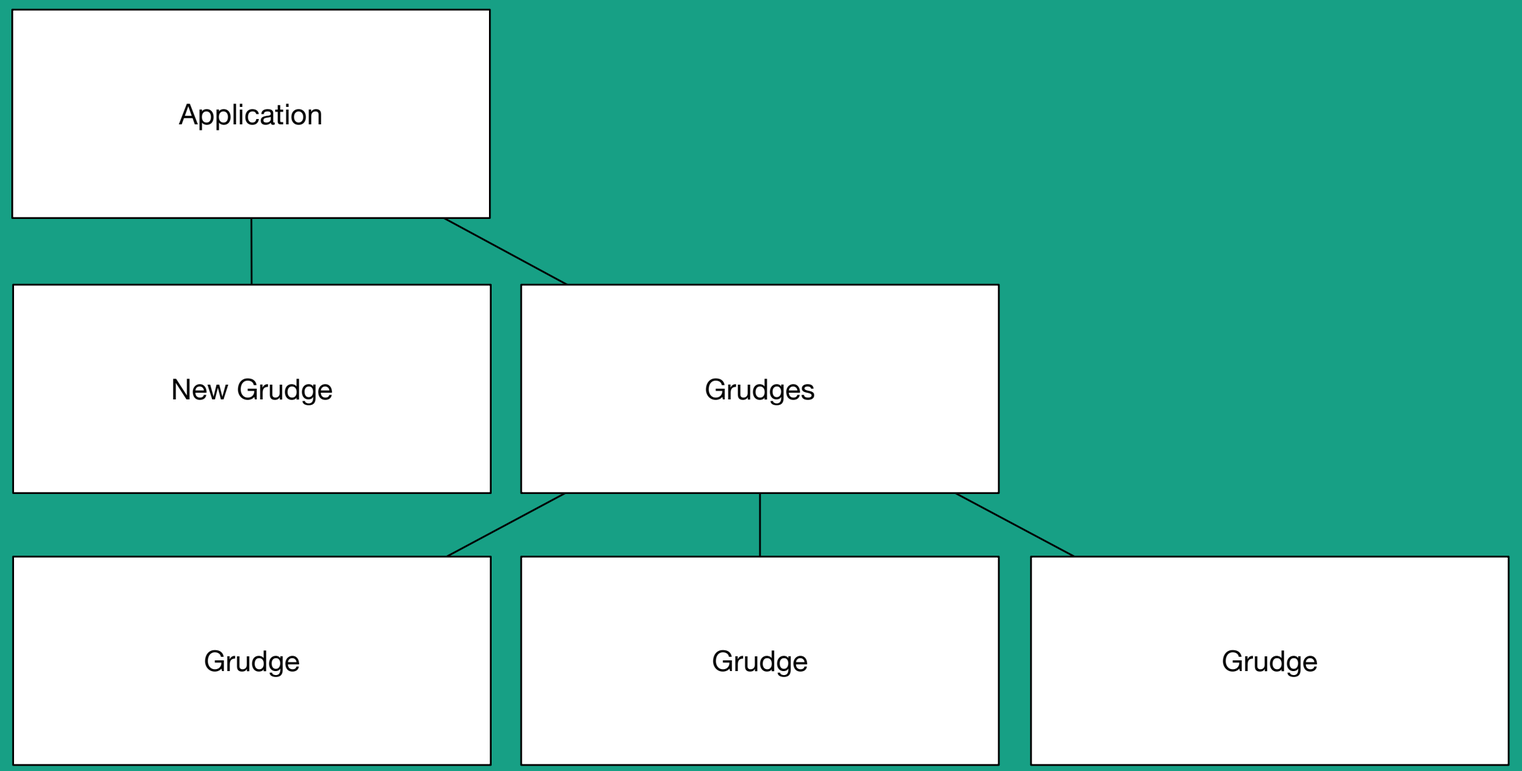
Exercise

- Be a better person than me.
- I’ve implemented the ability to add a grudge.
- Can you implement the ability to forgive one?

And now...

The Perils Prop Drilling

Prop drilling occurs when you have deep component trees.


Application
New Grudge
Grudges
Grudge
Grudge
Grudge

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

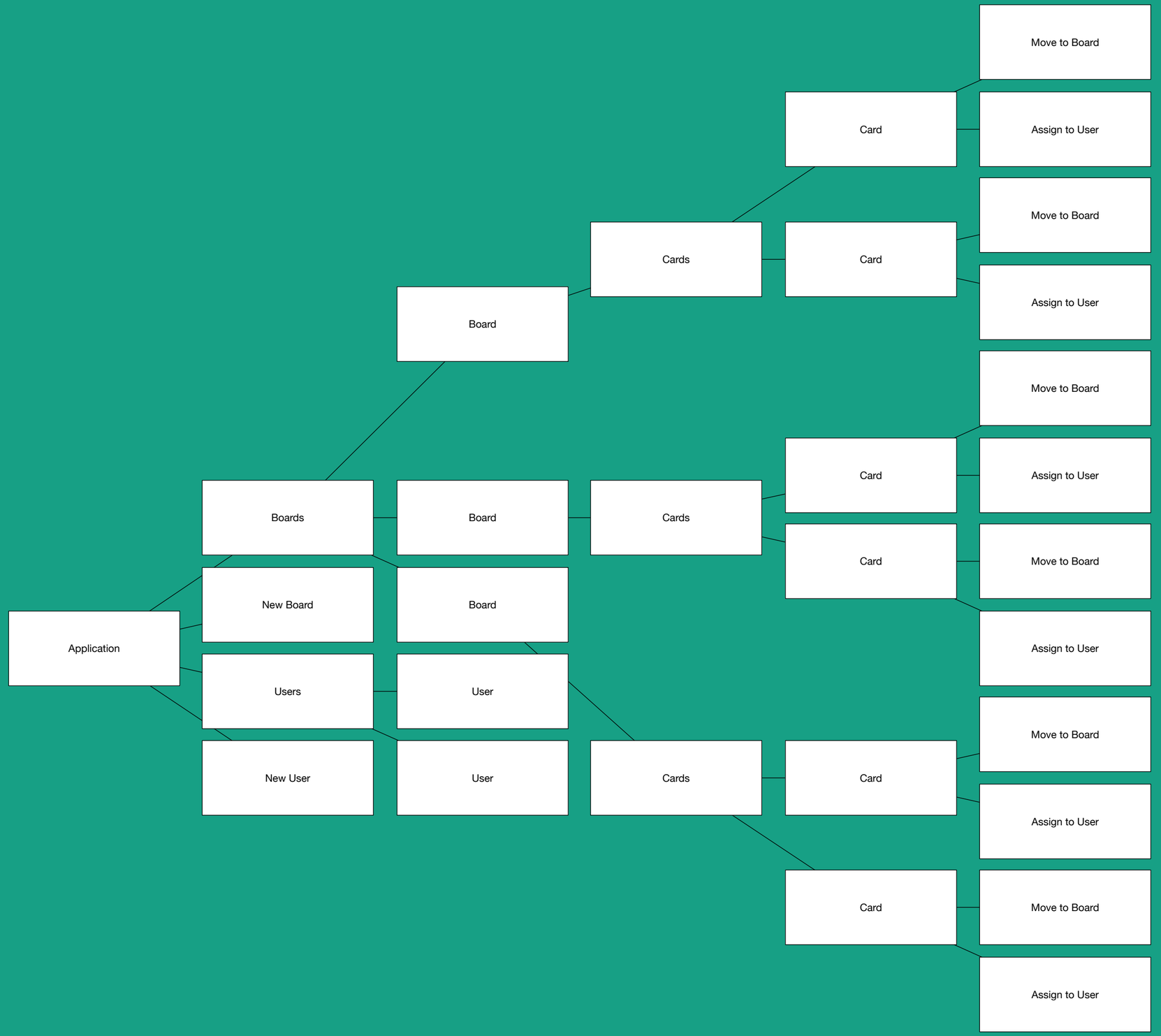
Learn enough React to make a mess

The "prop drilling" struggle is _real_.

Card assigned to **Steve Kinney**.

Steve Kinney

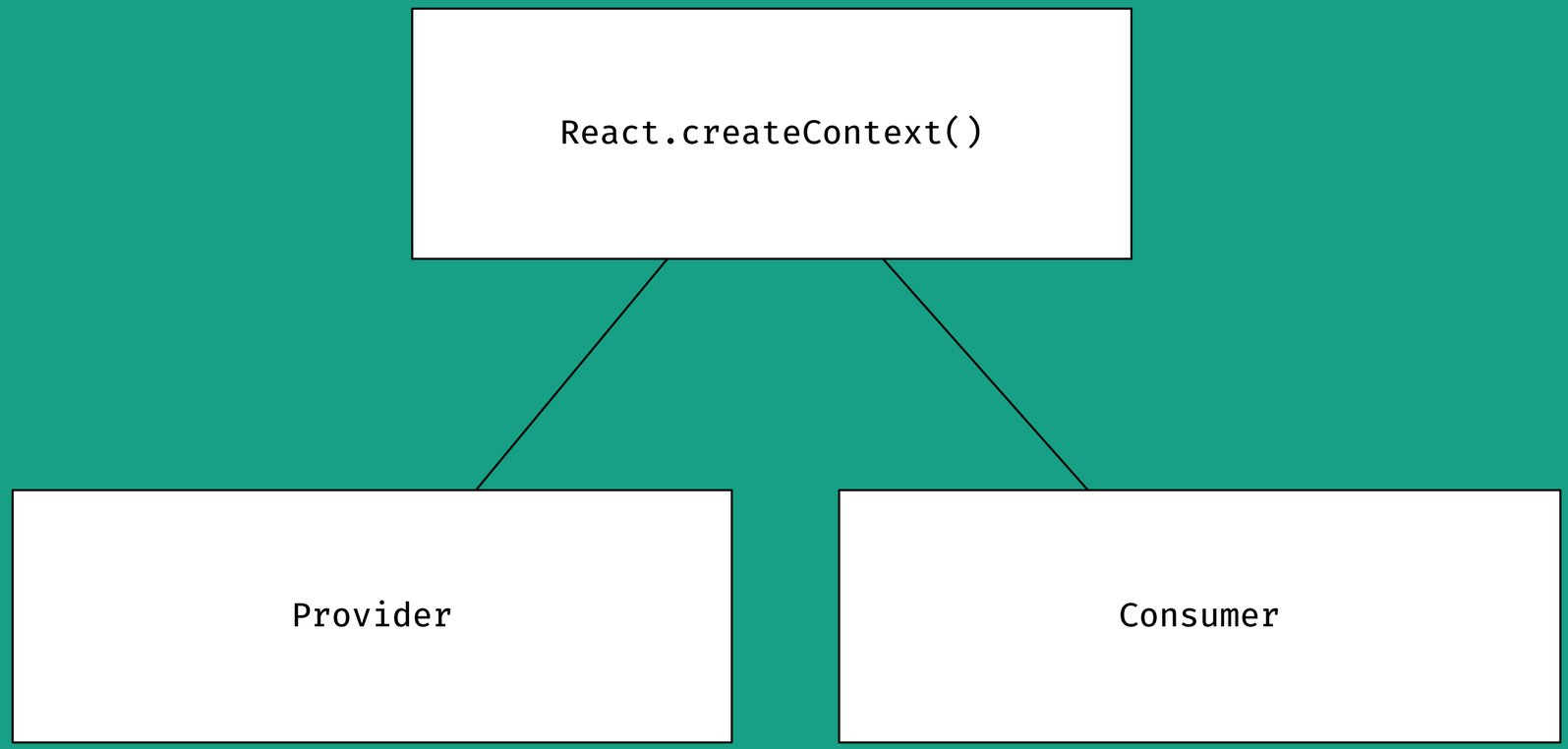
Toggle Options


Application
Boards
New Board
Board
Users
User
New User
User
Board
Cards
Card
Move to Board
Assign to User
Card
Move to Board
Assign to User
Board
Cards
Card
Move to Board
Assign to User
Card
Move to Board
Assign to User
Cards
Card
Move to Board
Assign to User
Card
Move to Board
Assign to User

And now...

The Context API

Context provides a way to pass data through the component tree without having to pass props down manually at every level.


React.createContext()
Provider
Consumer

```
import React from 'react';

const SuperCoolContext = React.createContext();

SuperCoolContext.Provider;
SuperCoolContext.Consumer;
```

```
<CountContext.Provider value={0}>
  <CountContext.Consumer>
    {value ⟹ <p>{value}</p>}
  </CountContext.Consumer>
</CountContext.Provider>
```

```
const CountContext = createContext();

class CountProvider extends Component {
  state = { count: 0 };

  increment = () => this.setState(({ count }) => ({ count: count + 1 }));
  decrement = () => this.setState(({ count }) => ({ count: count - 1 }));

  render() {
    const { increment, decrement } = this;
    const { count } = this.state;
    const value = { count, increment, decrement };

    return (
      <CountContext.Provider value={value}>
        {this.props.children}
      </CountContext.Provider>
    );
  }
}
```

 Live Coding 

Some Tasting Notes

- We lost all of our performance optimizations when moving to the Context API.

- What's the right answer? It's a trade off.

- Grudge List might seem like a toy application, but it could also represent a smaller part of a larger system.

- Could you use the Context API to get things all of the way down to this level and then use the approach we had previously?

And now...

How you structure your state matters

Some High Level Guidance

- Keep your data structures flat.

- Prefer objects to arrays.

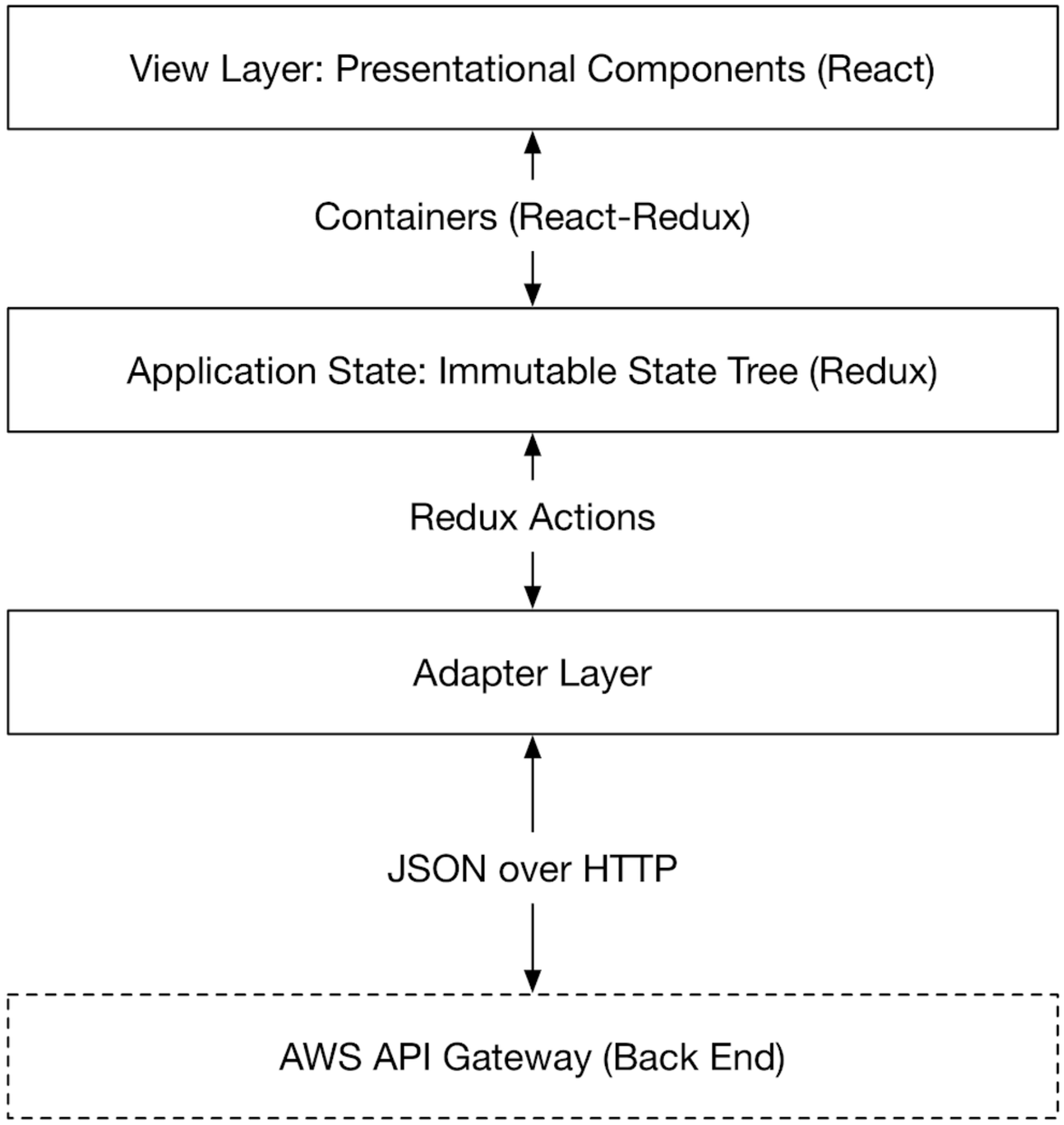
```
const board = {
  lists: [
    {
      id: '1558196567543',
      title: 'Backburner',
      cards: [
        {
          id: '1558196597470',
          title: 'Learn to Normalize Data',
          description: 'Iterating through arrays is rough business',
        },
        // …
      ],
    },
    // …
  ],
};
```

```
const board = {
  lists: {
    '1558196567543': {
      title: 'Backburner',
      cards: ['1558196597470'],
    },
    // …
  },
  cards: {
    '1558196597470': {
      title: 'Learn to Normalize State',
      description: 'This is much better.',
      assignedTo: '1',
    },
    // …
  },
  users: {
    '1': {
      name: 'Steve Kinney',
    },
  },
};
```

```
const removeCard = (listId, cardId) => {
  const targetList = lists.find(list => listId === list.id);

  const remainingCards = targetList.cards.filter(({ id }) => id !== cardId);
  const updatedList = { ...targetList, cards: remainingCards };

  const updatedLists = lists.map(list => {
    return list.id === listId ? updatedList : list;
  });

  setLists(updatedLists);
};
```

```
const removeCard = (listId, cardId) => {
  const list = this.state.lists[listId];
  const cards = omit(this.cards, cardId);
  const lists = {
    ...this.state.lists,
    [listId]: {
      ...list,
      cards: list.cards.filter(card => card.id === cardId),
    },
  };
  this.setState({ lists, cards });
};
```

```
const board = {
  lists: {
    '1558196567543': {
      title: 'Backburner',
      cards: ['1558196597470'],
    },
    // …
  },
  cards: {
    '1558196597470': {
      title: 'Learn to Normalize State',
      description: 'This is much better.',
      assignedTo: '1',
    },
    // …
  },
  users: {
    '1': {
      name: 'Steve Kinney',
    },
  },
};
```



User interface components deal with as little state as possible to make them agnostic of the lower layers as possible. They receive state and actions through `react-redux` to trigger changes.

The adapter layer takes the JSON payloads that we receive from the API(s) and transforms it into data structures based on the UI. This means that if APIs change, the rest of the client-side application is unaffected.

Moral of the Story

- Being thoughtful about how you structure your state can have implications for the maintainability of your application.
- It can also have some implications for performance as well.
- Modern versions of React come with tools for eliminating some of the pain points that used to require third party libraries.

And now...

What about fetching data?

`useEffect` is your friend.

 Live Coding 

Exercise

- Can you factor this out into a useFetch hook?
- It should take an `endpoint` as an argument.
- It should return `response`, `loading`, and `error`.
- You might want the following in order to get the right data back out.
    - `const characters = (response && response.characters) || [];`

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 is code to be executed later.

We’ve been a bit quiet about asynchronous code.

Here is the thing with reducers—
they only accept objects as actions.

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

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

 Live Coding 

Exercise

- There is another component called CharacterSearch.
- It would be cool if we could use the `api/search/:query` endpoint to get all of the characters that match whatever is typed in that bar.
- Can you implement that?

And now...

Advanced Patterns: Implementing Undo & Redo

```
{
  past: [allPastStates],
  present: currentStateOfTheWorld,
  future: [anyAndAllFutureStates]
}
```

And now...

Having Our Cake and Implementing It Too: `setState` *Using Hooks*

 Live Coding 

And now...

Using the Route to Manage State: A Brief Treatise

Why keep state in the route?

- It’s an established pattern that predates most of what we’re doing in the browser these days.
- It allows our users to save and share the URL with all of the state baked right in.

useQueryParam()

https://codesandbox.io/s/counter-use-query-params-6xpzo

And now...

The Ecosystem of Third-Party Hooks

https://nikgraf.github.io/
react-hooks/

Fin.