React + Redux

Tips and Best Practices for Clean, Reliable & Maintainable Code

by Cody Barrus
github: @goopscoop
medium: @scbarrus

React/Redux Data Flow 30
Second Review

React/Redux Data Flow 30 Second Review

- Our redux code lives in a module (sometimes refered to as DUCKS)
  - With ducks, we store all our related constants, actions, action creators, and reducer in a single file.
  - If another module needs to listen for a particular constant or needs to dispatch a particular action, we export the action here and import it where needed.

React/Redux Data Flow 30 Second Review

- Data lives in the reducer.
- `react-redux`'s connect function passes data to the component through props.
- The component displays the data and listens to events which dispatch an action.

React/Redux Data Flow 30 Second Review

- The action passes updated data it to the reducer.
- The reducer updates the store.
- Updated data through props to the componet.

Redux Modules

The data layer

Redux Modules

- Modules consist of
  - Constants
  - Actions
  - Reducer

Redux Modules

Actions

Redux Modules > Actions

- Actions are payloads of information that send data from your application to your store.
- They are the only source of information for the store.
- You send them to the store using `store.dispatch()`

Redux Modules > Actions

```
const ADD_TODO = '@todos/ADD_TODO' // Constant

// Action
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Redux Modules > Actions

- A simple action just passes data and a type to the reducer

```
const addToDo = (toDo) => ({
    type: ADD_TODO,
    toDo
});
```

Redux Modules > Actions

- Actions can take advantage of the `thunk` and `promise` middlewhere.
  - Adds flexibility to your Actions by giving them access to `state` and allowing them to return promises.
  - Can easily get out of hand.

Redux Modules > Actions

```
// Arbitrary example
const complexAddToDo = (toDo) => {
    return (dispatch, getState) => {
        const {userPrefs} = getState().user;

        return getLists.then(list => {
            dispatch(populateLists(list))
        }).then(() => {
            if (userPrefs.isAwesome) {
                dispatch(addToDo(toDo))
            }
        })
    }
}
```

Redux Modules > Actions

A few pointers to keep these actions reasonable:

- Keep complexity out of your Actions. Pure Actions (w/o side effects) are best actions.
- Prefer data manipulation in the reducer.

Redux Modules > Actions

- Keep API calls in their own util. This keeps your actions cleaner, and simpler to unit test.
    - Handle necessary data manipulation for API calls in this util rather than in the action.

Redux Modules > Actions

getState

- Don't call getState unecessarily. For example, don't...
  - use getState for getting data that's handled by the local reducer. Insead, dispatch an action and access that data from within the reducer itself.
  - call getState more than once.

Redux Modules > Actions

getState continued

- Call getState only once, and near the top of your function.
- Always treat data from the store as though it were immutable.

Redux Modules > Actions

```
const complexAddToDo = (toDo) => {
    return (dispatch, getState) => {
        const {
            user: {userPrefs},
            movies: {titles},
            ....
        } = getState();

        ....
    }
}
```

Redux Modules > Actions

API Util

- Abstaracts API calls from Actions, leaving cleaner, easier to test actions.
- Handle any data manipulation for the sake of API calls here rather than in the action.
- This util is especially nice for complex api calls, as it removes the mental payload of parsing busy `Promise` chains within actions.

Actions Summery

- Keep actions pure and simple.
- `thunk` and `promise` middleware add power, but with great power comes great responsibility.
- API calls live in a seperate util.

Reducer

Redux Modules > Reducer

- The Reducer specifies how the applications state changes in response to an action.

Redux Modules > Reducer

```
const ADD_TODO = '@todoModule/ADD_TODO'; // Constant

const initalState = []

export default const myReducer = (state = initialState, action) => {
    switch (action.type) {
        case ADD_TODO:
            return [
                ...state.slice(0, action.index),
                action.payload,
                ...state.slice(action.index, state.length + 1)
            ]
        defualt:
            return [
                ...state
            ];
    }
}
```

Redux Modules > Reducer

Tips for clean, efficiant reducers:

- The best reducers specialize in a single concern.
- Complex data manipulation lives in the reducer.
- Utilize helper functions and utils to keep your reducer clean and easy to parse.
- Reducers can listen for actions from another module if needed.

Redux Modules > Reducer > Listen to other modules actions

```
// expenseHomeModule.js
const RESET_EXPENSE_STATE = '@expenseHome/RESET_EXPENSE_STATE';

// expenseItemizationModule.js
import {RESET_EXPENSE_STATE} from '../expenseHomeModule';

export default const myReducer = (state = initialState, action) => {
    switch (action.type) {
        ....
        case RESET_EXPENSE_STATE:
            return {
                ...initialState
            }
        ....
    }
}
```

Redux Modules

Constants

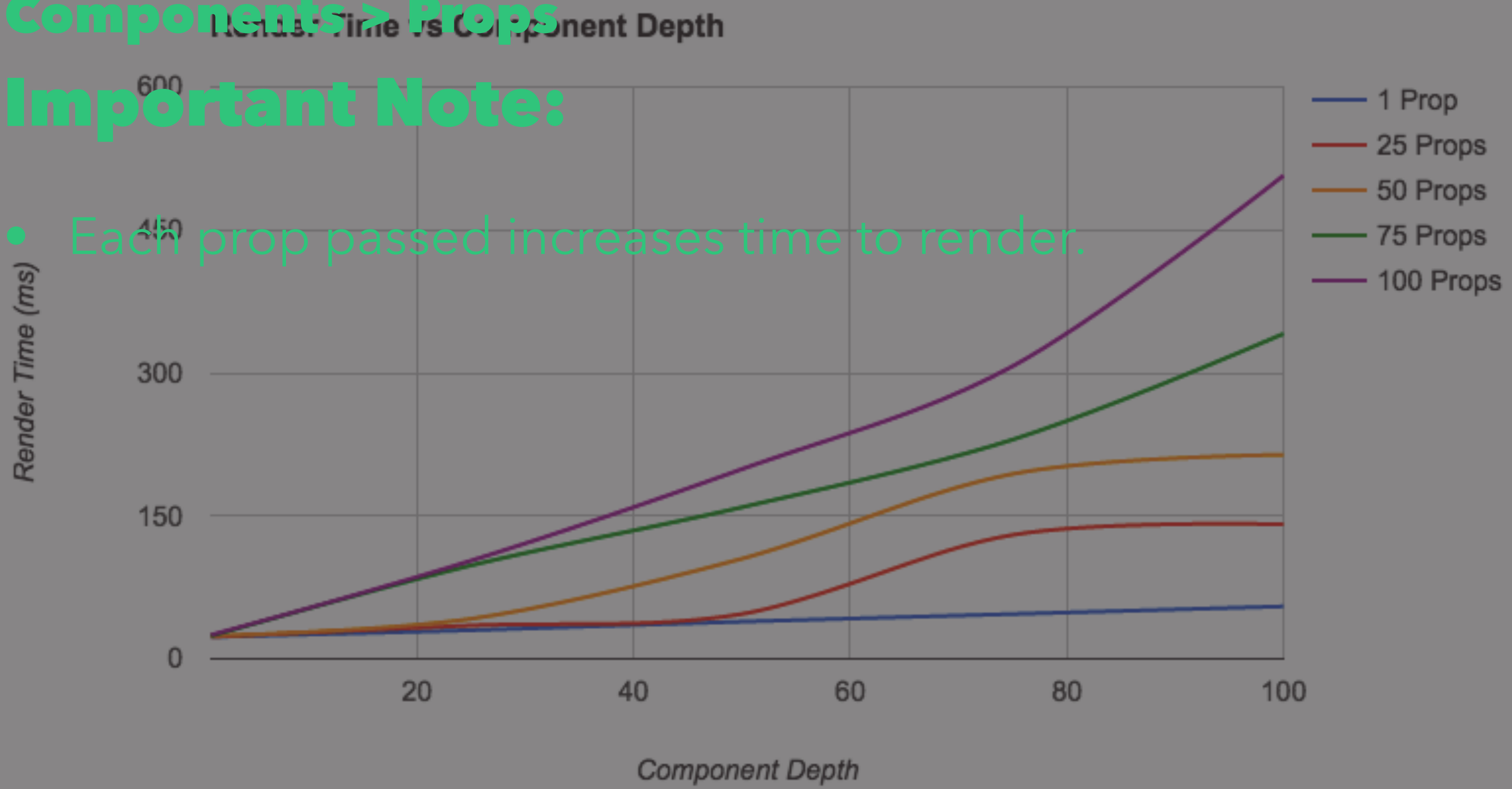
- Preface the constant with the name of the reducer.
- **Ok:** `const SUBMIT_REPORT = 'SUBMIT_REPORT'`
- **Better:** `const SUBMIT_REPORT = '@report/SUBMIT_REPORT'`
  - Leads to simpler debugging.
  - Reduces the likelyhood of constants from other modules conflicting.

Components

Components > Props

Important Note:

- Each prop passed increases time to render.



Components > Props

Passing Props

- There are several ways to pass props:
  - Component to Component
  - Connected Component
  - Higher Order Component

Components > Props

Passing Props - Component to Component

- Simplest method of passing data to a component through props is Component to Component.

Components > Props > Component to Component

```
// Home.jsx

class Home extends React.Component {
    ....
    render() {
        return (
            <div>
                <MyComponent
                    prop1={this.state.thing1}
                    prop2={this.props.thing2}
                    prop3={this.props.thing3}
                    ....
                />
            <//div>
        )
    }
}
```

Components > Props > Component to Component

- **Good for:**
  - A small number of props.
  - Parent component methods.
  - Data local to parent component.

Components > Props > Component to Component

- **Not good for:**
  - A large number of props.
  - Passing actions to child component.
  - Passing data from redux store to child component.
  - Passing `{...props}`.

Components > Props > Component to Component

```
// Home.jsx

class Home extends React.Component {
    ....
    render() {
        return (
            <div>
                <MyComponent
                    prop1={this.state.thing1} // OK
                    prop2={this.props.thing2} // NOT SO GOOD
                    prop3={this.props.thing3}
                    ....
                />
            <//div>
        )
    }
}
```

Components > Props > Component to Component

Passing props from state

- **Good when...**
  - ...data is local to component, and child component is reusible, presentational (dumb) component. *ie. open state of a modal*
- **Not good when...**
  - ...data is better handled in the redux reducer. *ie. data is required for multiple components*

Components > Props > Component to Component

Passing props from props

- **Just don't do it**
  - Creates tight coupling between components.
  - Makes components difficult to maintain.
  - Adds tech debt.
  - Simple data changes will force you to refactor at least 2, possibly more, components.
  - Instead, use connected pattern (more on that later).

Components > Props > Component to Component

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

Components > Props > Component to Component

Passing `{...this.props}`

- **NEVER! It may look cool and easy, but...**
  - Causes even tighter coupling involving at least 3 components!
    - Grandparent (where data is coming from).
    - Parent (where component is initialized)
    - Child (where data is being utilized).

But wait, theres more!

Components > Props > Component to Component

- More `props` equals more time to render and spreading props passes everything we need as well as several that we don't.
  - In forms this can get especially apparent. Slight decreases in perf add up when every keystroke is delayed even slightly

Components > Props > Component to Component

Caveat

- Of course there's an exception to everything, including this. Higher Order Components often make use of `{...props}` which is fine. **Just be sure to think about when this works well and when it doesn't.**

Components > Props > Component to Component

Summery

- Explicitly list props.
- Avoid passing parent props to children (ie. `prop={this.props.foo}`). Instead prefer the connected pattern.
- Avoid spreading props from one component to another (ie. `{...props}`).

Connected Component Pattern

Components > Props > Connected Component

- A connected component uses the `react-redux` connect function to pass props directly from state.

Components > Props > Connected Component

- **The Good**:
  - Greately reduces the code complexity.
  - Removes tight coupling of components.
  - Act's as documentation on actions your components depend on.
- **The Bad**:
  - Requires more boilerplate code.

Components > Props > Connected Component

```
// MyComponent.jsx
import {connect} from 'react-redux';

const MyComponent = ({ prop1, prop2, prop3 }) => {
    return (
        <div>
            {`I am a ${prop1} that ${prop2} when ${prop3}`}
            ....
        <//div>
    )
}

const mapStateToProps = state => ({
    prop1: state.expense.prop1,
    prop2: state.itemization.prop2,
    prop3: state.user.prop3
});

export defualt connect(mapStateToProps)(MyComponent);
```

Components > Props > Connected Component

```
// Home.jsx

class Home extends React.Component {
    render() {
        return (
            <div>
                <MyComponent />  // :) State data is already mapped to props
            <//div>
        )
    }
}
```

Components > Props > Connected Component

- Using this pattern, both components are independant of one another.
  - It can be dropped anywhere and will always work as intended.
- Keeps data flow through your app direct and simple.
- No need to create a seperate container file. That simply adds complexity without any real benefit.

Components > Props > Connected Component

- Remember: avoid passing unnecessary props.

```
// avoid patterns like this, they'll cause a hit to your performance

connect(state => ({
    ...state // Nope!
}));

connect(state => ({
    movies: ...state.movies, // Nah
    books: ...state.books, // Negative
    tvShows: ...state.tvShows // No bueno
}));
```

Components > Props > Connected Component

- Instead, explicity require each prop needed for that particular component.

```
// looks good!

connect(state => ({
    movieTitles: state.movies.titles,
    bookTitles: state.books.titles,
    tvShowTitles: state.tvShows.titles
}));
```

Components > Props > Connected Component

- Benefits of explicitly declaring props include:
  - Easy to see when a component has expanded past its concern.
  - Only maps required props, decreasing time to render.

Components > Props > Connected Component

Summery

- Prefer Connected Pattern over Component to Component pattern.
- Connected componets are simplier to maintain, and reduce tech debt significantly.
- When using connect, avoid the spread opporator because each prop passed hits perf.
- Also spread in connect obscures your props a bit. Explicit is

Higher Order Components (HOC)

Components > Props > Higher Order Components

- A function that takes a component and returns a new component.
- Good for reusing component logic.
- HOCs make it easy to layer on behavior while maintaining a separation of concerns.

Components > Props > Higher Order Components

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentWillReceiveProps(nextProps) {
      console.log('Current props: ', this.props);
      console.log('Next props: ', nextProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it.
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

Components > Props > Higher Order Components

- Adds additional functionality, or injects data, into the component it wraps.
- **Good for:**
  - Behaviour that is needed throughout the app.
  - Common data sets needed in several components.
- **Warning:** HOCs can hurt performance. If you're managing your `props` well else where, you can usually get away with this. If you're not, your User Experiance could deminish.

Components > Props

Props Summery

- Avoid passing unnecessary props.
- Connected Components **>** Higher Order Compontents **>** Component to Component.
- When a component has too many props, consider breaking into several, more focused components.
- All these rules have exceptions. Every circumstance is different.

Class Components

Also applies to `React.createClass` components

Components > Class Components

- The basic building block of every React app

Components > Class Components

- **The good:**
  - Very powerful.
  - Have access to lifecycle methods and `this.state`.
- **The bad:**
  - Can easily become over complicated, too big, or unweildly.
  - `this.state` is the source of many bugs. Better to handle data in the redux module in most cases.

Stateless Functional Components

SFCs

Components > Stateless Functional Components

- SFCs are the simplest way to declare components.
- They are basic javascript functions that take props and return jsx.

Components > Stateless Functional Components

```
function Welcome(props) {
  return <h1>Hello, {props.name}<//h1>;
}
```

Components > Stateless Functional Components

- **The good:**
  - Simpler than `class` components and easier to maintain.
  - Givin the same input, an SFC will always have the same output. *Not so with a `class` component*
  - Do not have access to `state` -- yes, that is a good thing ;)

Components > Stateless Functional Components

- **The bad**
  - SFCs do not have access to state or any React lifecycle methods.
  - That's it really...

Components > Stateless Functional Components

- SFCs **>** `class` components.
- `class` components are best used as the root component of a view, or for components that rely on lifecycle methods. *In all other cases, use SFCs.*

Refs

Components > Refs

- There are two primary ways for a parent componet to reach into a child component
  - surfacing values or methods (such as event hanlders) through props.
  - refs.
- `refs` are generally references to DOM elements within a component.

Components > Refs

```
// with refs
componentDidMount() {
    this.refs.someWidget.focus()
}

// without refs
render() {
    return <Widget focused={true} />;
}
```

Components > Refs

- **The good:**
  - Occasionally helpful. Occasionally.
- **The bad**
  - Increase function calls and property merging.
  - Can obscure a components dependencies.
  - Can easily lead to tight coupling and debugging nightmares.

State

Components > State

- There are several ways to handle the state of a particular component. Let's look at some of the methods and compair.
- `class` components have access to `this.state` whereas SFCs do not.
- Accessing and updating a components state is relatively painless.

Components > State

```
class MyComponent extends React.Component {

    handleChange(value) {
        this.setState({
            foo: value
        });
    }

    render (
        <span>{state.value}</span>
    )
}
```

Components > State

- **The good:**
  - Very easy.
  - Great for managing things that aren't related to data in the redux store. *ie. active states, is modal open, etc*

Components > State

- **The bad:**
  - Relying on component state too much can make components difficult to reuse and maintain.
  - As components multiply, frequent state manipulation can add to your technical debt.
  - Storing data in state can lead to components being too encapsulated.

Components > State

General State tips

- If you need to use 'componentWillRecieveProps' to fit some data change into the component, consider refactoring it to read data from the redux store instead.
- If the component uses state, but doesn't use any lifecycle methods, refactor it into a connected SFC.
- If the component uses state AND lifecycle methods, refactor it to become a connected class component.

Components > State

Summery:

- Connected SFCs **>** `class` components utilizing state + lifecycle methods **>** `class` components only utilizing state.
- State is good for local data such as the open state of a modal.

Components > State

- It can be argued that even this data is better handled in your redux code.
- If a `class` component is using state, and you're forced to use `componentWillRecievеProps`, consider refactoring.

Hopefully you learned something new. Do you have a tip or best practice not listed here? Leave it in the comments!