React Design Best Practices

A Software Presentation From ZERRTECH

Jeremy Zerr

Site: https://www.zerrtech.com
https://www.linkedin.com/in/jrzerr
https://twitter.com/jrzerr

Josh Taylor

Site: https://www.zerrtech.com
https://www.linkedin.com/in/jj-taylor/
https://twitter.com/joshuaj_taylor

THAT REACT
SO HOT RIGHT NOW
imgflip.com

The Plan

- ☐ General Design Considerations
- ☐ Project Setup/Structure
  - ☐ Boilerplates
  - ☐ Package Manager
  - ☐ Node Version Manager
  - ☐ Code style
  - ☐ Style approach
  - ☐ Folder Structure
- ☐ Project Code
  - ☐ Use Redux
  - ☐ Use Babel-Polyfill
  - ☐ Error Handling
  - ☐ Version Checking
  - ☐ Kinds of Components
  - ☐ Handling Side Effects
  - ☐ Routers
  - ☐ Testing
  - ☐ Documenting in Code

Our Philosophy

- Be practical - know the ideal but be realistic
- Don’t require devs to remember a bunch of rules
- Use tools that encourage education instead of automagically fixing stuff
- Don’t be so set in your ways that you ignore an option that is the right fit for a particular project but not others

Best Practices

Learn from the mistakes of others!

Ways to improve code

- Questions to ask yourself when fixing/refactoring code
  - Could I have prevented this bug from happening?
  - What did I do to cause this difficulty? Takes responsibility
- Learn from refactoring and do it better the first time on the next project

Look out for code smells

- Duplicated code
- Large class
- Too many arguments/attributes
- Lines that are too long
- Your linter should help grow your intuition on these so they become second nature

Project Setup/Structure Best Practices

create-react-app vs other boilerplates

- The official boilerplate
- Excellent documentation
- Familiar to client devs
- Promising future support
- It’s ejectable

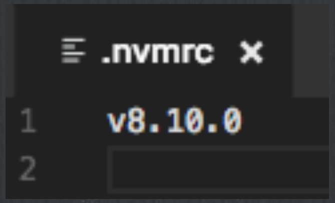
Use a Package manager w/ a lock file

- Lock file is required for accurate reproducibility
- Old way
  - npm + npm shrinkwrap - manual process
- New ways
  - npm v5.0+ now has lock file built in
  - yarn
    - Had a lock file from the beginning
    - Faster install w/ parallelism
- Make sure lock file is committed in repository
- Clearly state package manager of choice in Readme

LIFE IS LIKE AN NPM INSTALL
YOU NEVER KNOW WHAT YOU'RE GONNA GET
imgflip.com

Node Version Manager (NVM)

- ☐ Not specifying the node version can cause reproducibility issues down the road
- ☐ Good practice to match production environment
- ☐ Add an .nvmrc file that specifies the exact node version in the codebase.



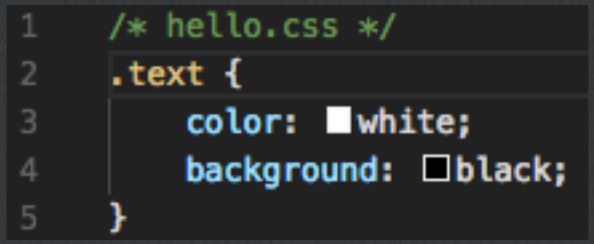
```
v8.10.0

```

Code Style

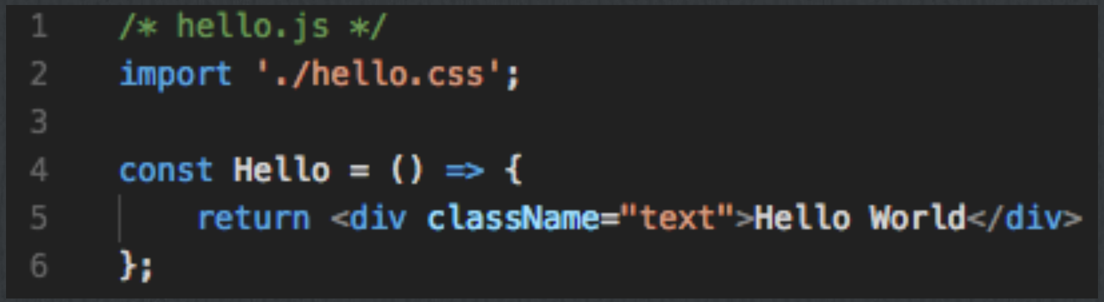
- Goal: Encourage code readability/consistency
- Linter
    - Use create-react-app lint rules in .eslintrc file + additional rules.
    - In VS Code use “ESLint” extension to highlight broken rules (won’t show up in console)
- Editor Configuration File
    - We use an .editorconfig file to enforce editor formatting rules, like spaces and end of line/ file newline.
    - In VS Code use “EditorConfig for VS Code” extension.
- We don’t use auto code formatters like "prettier”
- We prefer educating the developer on changes necessary to meet code style guidelines

Styles
.css files vs CSS-in-JS

- We prefer CSS files over CSS-in-JS, CSS modules, or inline styles
  - Easier for designers to modify CSS files
  - No JS/React knowledge is necessary
- Examples
  - CSS files



```css
/* hello.css */
.text {
    color: white;
    background: black;
}
```



```js
/* hello.js */
import './hello.css';

const Hello = () => {
    return <div className="text">Hello World</div>
};
```

Styles
.css files vs CSS-in-JS

- ☐ **Styled components (styled-components)**

```
import styled from 'styled-components';

const Text = styled.div`
  color: white,
  background: black
`
return <Text>Hello World</Text>;
```

- ☐ **Styles inline**

```
return <div style={{ color: 'white', background: 'black' }}>Hello World</div>;
```

- ☐ **CSS Modules**

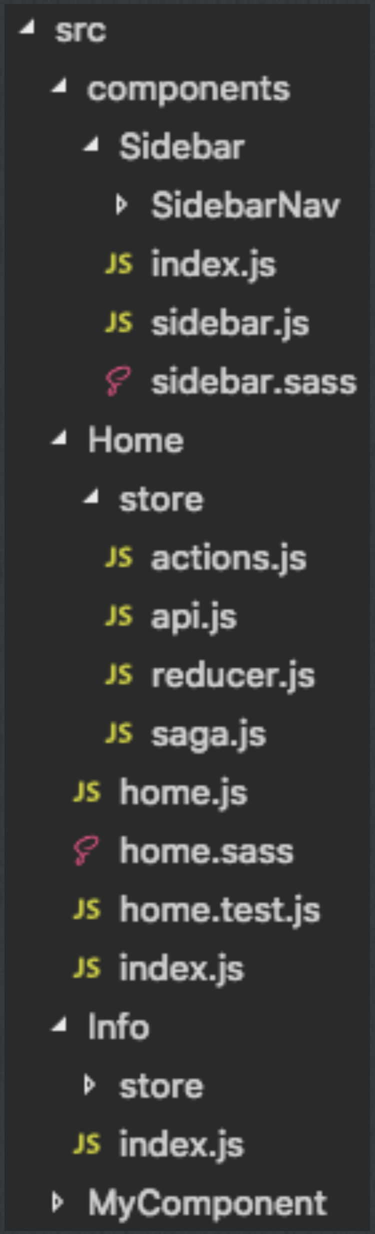
```
/* hello.css */
:local(.text) {
    color: ■white;
    background: □black;
}
```

```
/* hello.js */
import styles from './hello.css';

const Hello = () => {
    return <div className={styles.text}>Hello World</div>
};
```

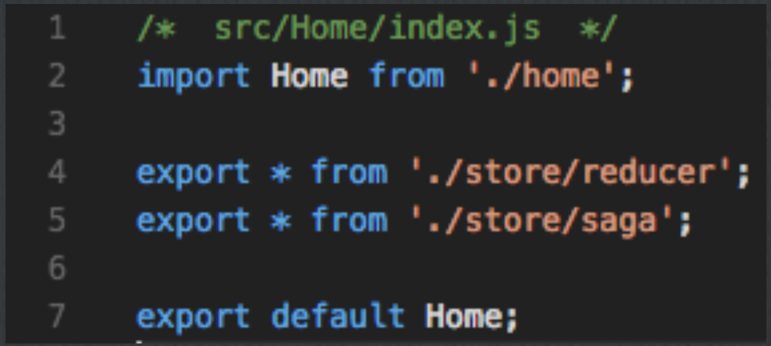
CSS Preprocessors
LESS vs SASS

- We choose the one that is most popular with the libraries we use
  - Bootstrap v2/v3 used LESS so we have used LESS
  - Bootstrap v4 uses SASS so we plan to use SASS more often
- Leave the generated CSS files and maps out of repo/codebase
- When using “import ‘./mycomponent.css’” in components, avoid CSS naming collisions by using a unique className on component’s parent element

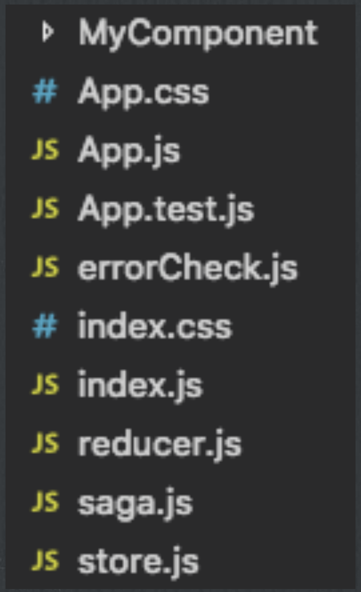
Folder Structure



```
src
  components
    Sidebar
      SidebarNav
      index.js
      sidebar.js
      sidebar.sass
  Home
    store
      actions.js
      api.js
      reducer.js
      saga.js
    home.js
    home.sass
    home.test.js
    index.js
  Info
    store
    index.js
  MyComponent
```

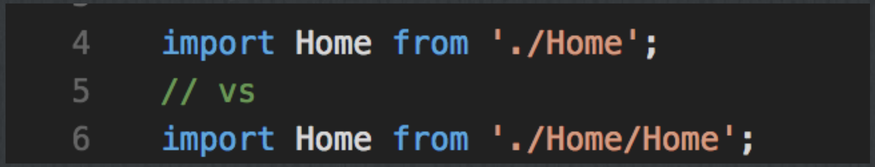


```js
/*  src/Home/index.js  */
import Home from './home';

export * from './store/reducer';
export * from './store/saga';

export default Home;
```



```
MyComponent
App.css
App.js
App.test.js
errorCheck.js
index.css
index.js
reducer.js
saga.js
store.js
```

- ☐ All components have their own folder
  - ☐ Contains all related code and styles
  - ☐ Sub components included in folder
- ☐ Put React component code in a named .js file (.jsx is not recommended)
  - ☐ Stack trace and editor readability

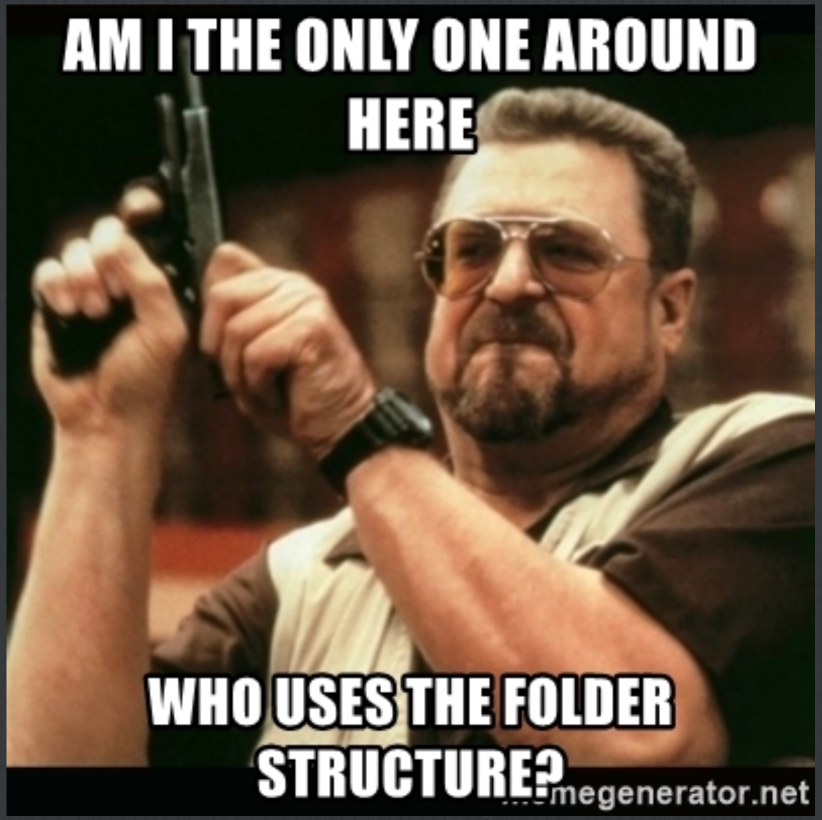


```js
import Home from './Home';
// vs
import Home from './Home/Home';
```

- ☐ Include related reducers, action creators, sagas in store/
- ☐ Export everything in an index.js file

Folder Structure

- Why we chose this structure
  - It scales well
  - Allows for code-splitting
  - Locality of all related code and styles



Project Code
Best Practices

Use Redux in most cases

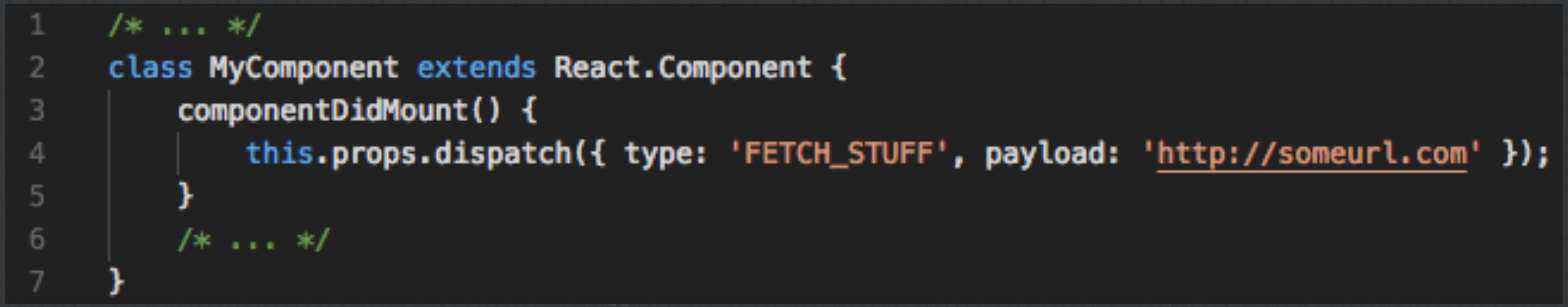
- We use Redux almost exclusively
  - One-way data flow coupled with the React virtual DOM computations provides performant web apps
- Redux + Redux Dev Tools === Awesome

Use Action Creators in Redux

- Actions in Redux are objects that have a type and payload
- The payload is specific to the action type
- Tough to know the payload structure for a particular type of action without a standard defined
- Action Creators turn actions into functions that have a name and can be imported
- Parameters to Action Creators can be formally defined data structures using JSDoc or Typescript. Making them easy to use across the code base
- Minimizes the searching a developer has to do to use something

Use Action Creators in Redux

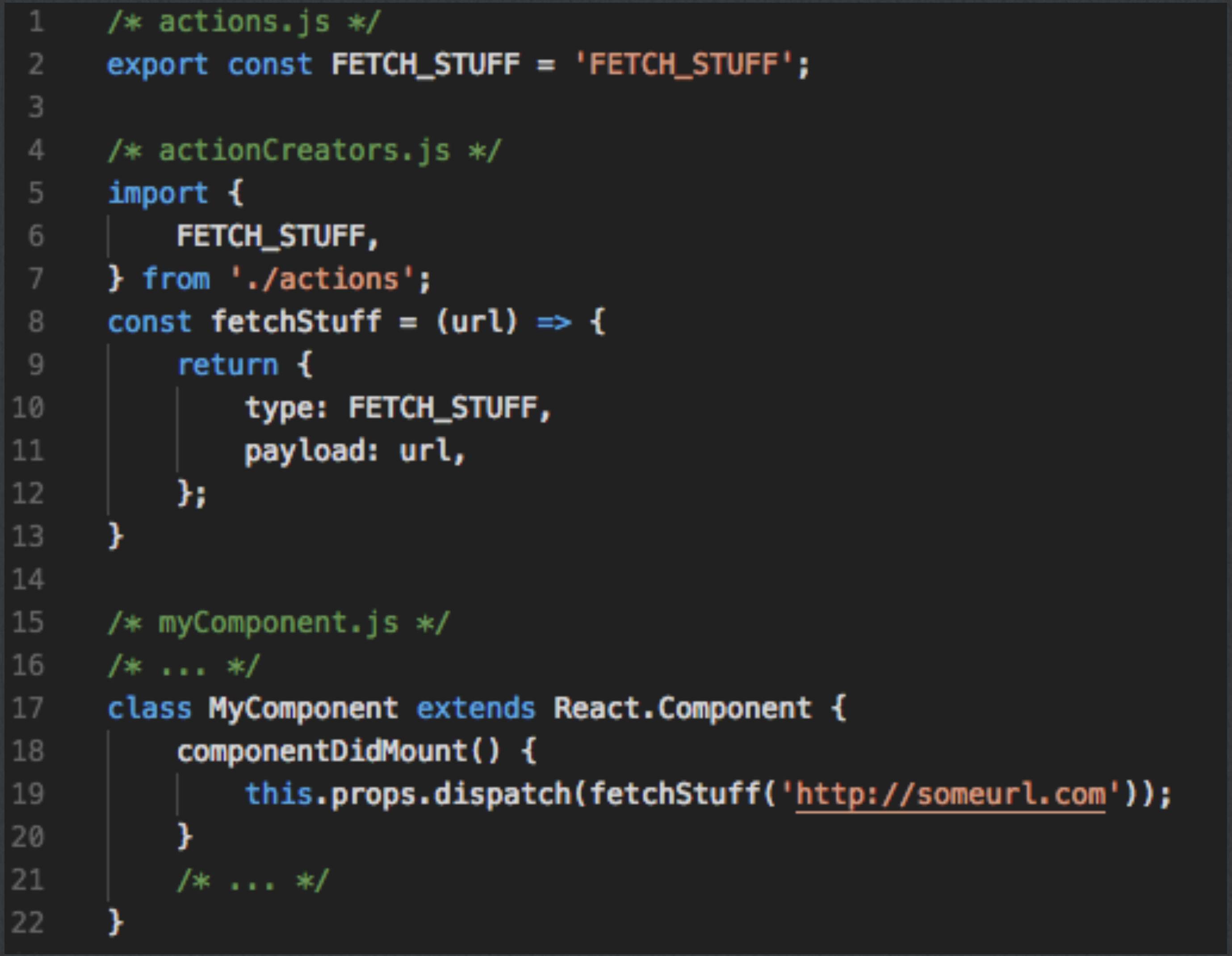
Here is how it looks without an action creator



```
/* ... */
class MyComponent extends React.Component {
    componentDidMount() {
        this.props.dispatch({ type: 'FETCH_STUFF', payload: 'http://someurl.com' });
    }
    /* ... */
}
```

Use Action Creators in Redux

Adding an action creator creates a standard form for the action



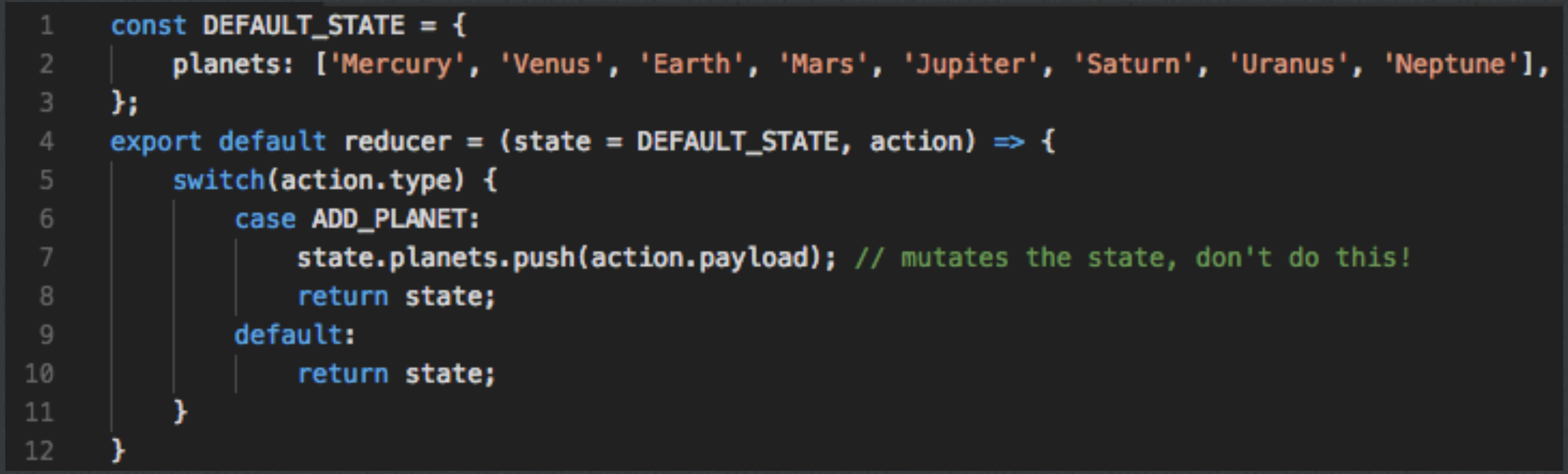
```
/* actions.js */
export const FETCH_STUFF = 'FETCH_STUFF';

/* actionCreators.js */
import {
    FETCH_STUFF,
} from './actions';
const fetchStuff = (url) => {
    return {
        type: FETCH_STUFF,
        payload: url,
    };
}

/* myComponent.js */
/* ... */
class MyComponent extends React.Component {
    componentDidMount() {
        this.props.dispatch(fetchStuff('http://someurl.com'));
    }
    /* ... */
}
```

Use immutable data changes within your reducers

- Use only immutable data changes within your reducers to unlock the performance of your web app
- Allows PureComponent to be used, increasing performance
- We don’t use ImmutableJS often, but we should use it for the data structures inside Redux reducers

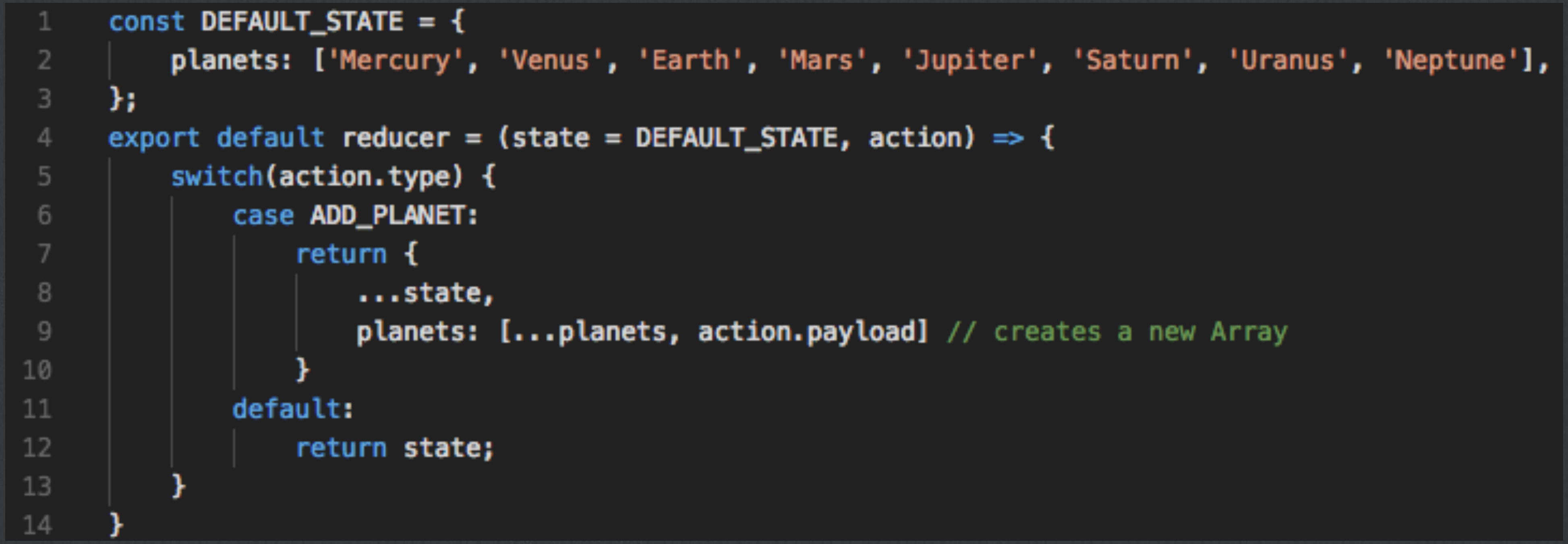
Use immutable data changes within your reducers

Bad



```
const DEFAULT_STATE = {
    planets: ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'],
};
export default reducer = (state = DEFAULT_STATE, action) => {
    switch(action.type) {
        case ADD_PLANET:
            state.planets.push(action.payload); // mutates the state, don't do this!
            return state;
        default:
            return state;
    }
}
```

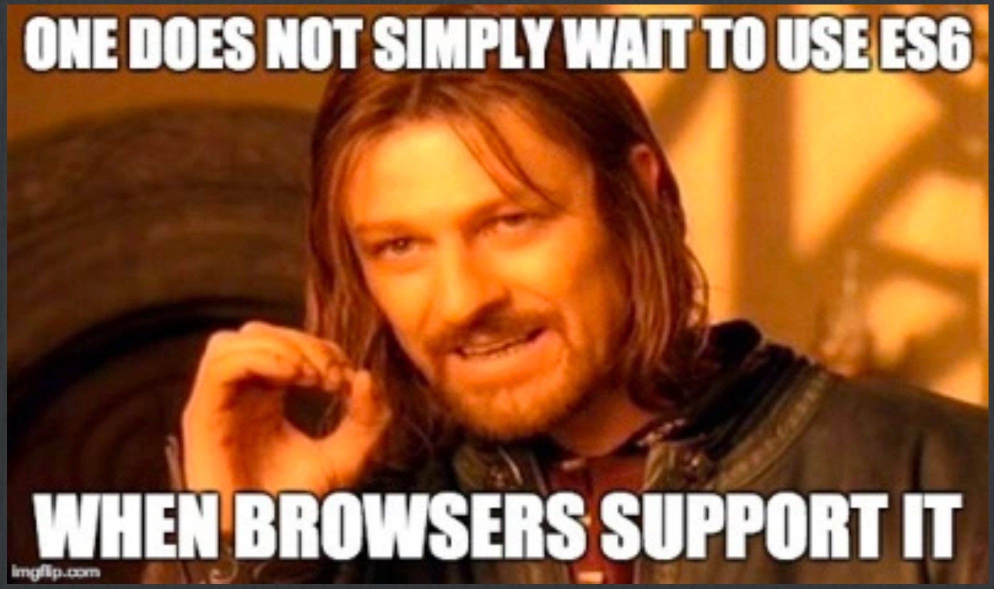
Good

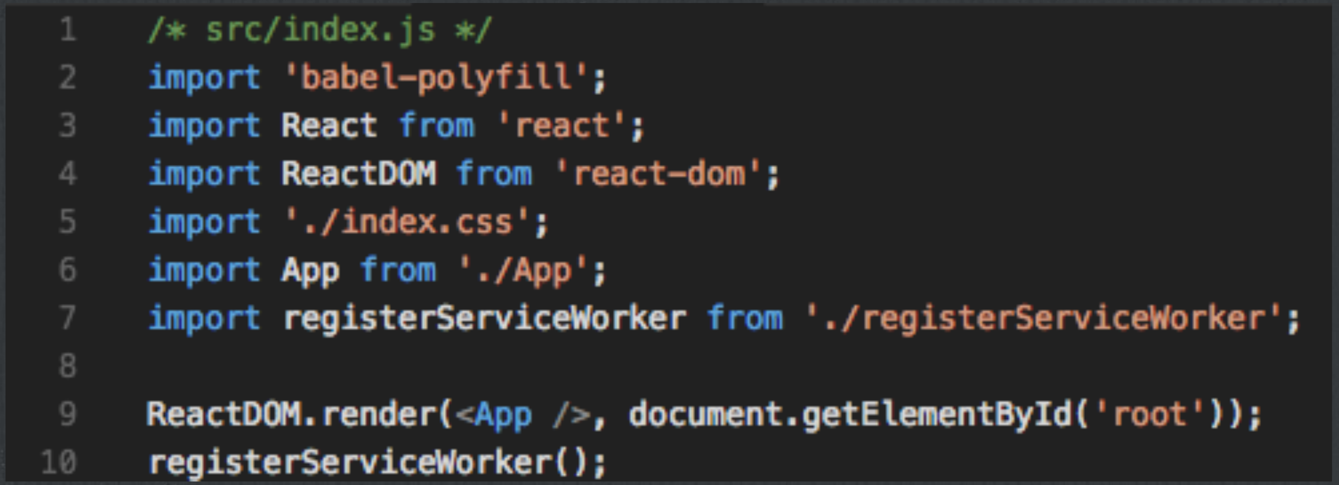


```
const DEFAULT_STATE = {
    planets: ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune'],
};
export default reducer = (state = DEFAULT_STATE, action) => {
    switch(action.type) {
        case ADD_PLANET:
            return {
                ...state,
                planets: [...planets, action.payload] // creates a new Array
            }
        default:
            return state;
    }
}
```

Use Babel-Polyfill

- Using ES6 features can cause problems in Firefox and Internet Explorer
  - Array.from, other Array methods, and some Map methods
- We choose to take the code size hit (50-60kb) and not limit our usage of ES6 features
- Babel version can only be changed if we eject create-react-app
- $ yarn add babel-polyfill



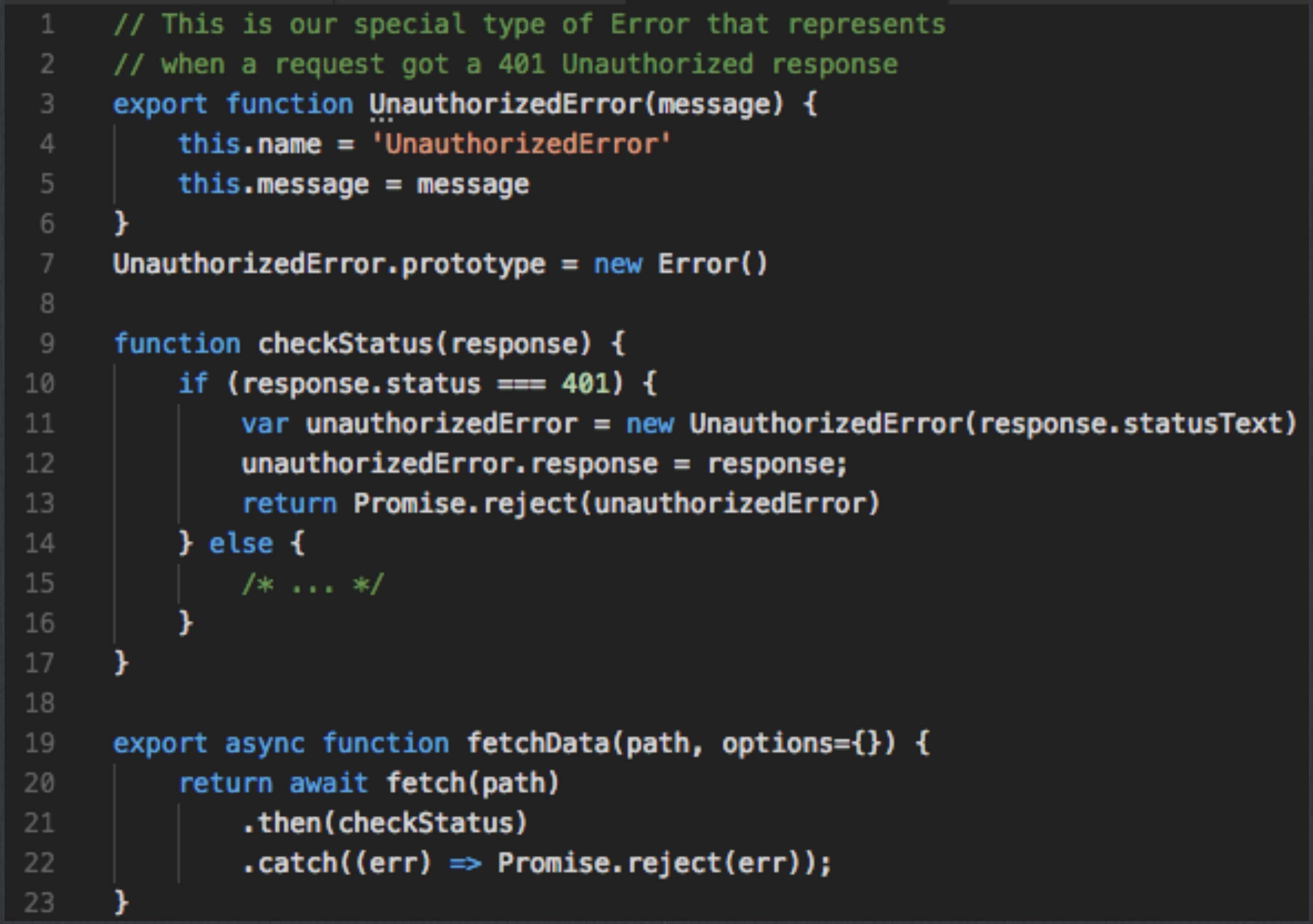


```
/* src/index.js */
import 'babel-polyfill';
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

Error Handling

- We transform common Errors to be more descriptive
    - For example, we transform 401 Unauthorized into a custom UnauthorizedError and re-throw it
    - Error-dependent code is easier to read
    - Abstracts the response checking logic to a central location.

Error Handling: Example



```
// This is our special type of Error that represents
// when a request got a 401 Unauthorized response
export function UnauthorizedError(message) {
    this.name = 'UnauthorizedError'
    this.message = message
}
UnauthorizedError.prototype = new Error()

function checkStatus(response) {
    if (response.status === 401) {
        var unauthorizedError = new UnauthorizedError(response.statusText)
        unauthorizedError.response = response;
        return Promise.reject(unauthorizedError)
    } else {
        /* ... */
    }
}

export async function fetchData(path, options={}) {
    return await fetch(path)
        .then(checkStatus)
        .catch((err) => Promise.reject(err));
}
```

Error Handling - Sentry

- Send unhandled errors to a monitoring service
  - We use Sentry
  - Own your errors. Be aware of them. Fix them!
  - Sentry can also include redux state and action history
  - raven-js is the official Sentry npm package
  - raven-for-redux is the redux integration npm package we prefer

Version Checking

- Problem
  - What if your users are still using an old version of your SPA because they haven’t refreshed in a week?
  - How do they get your newest code?

Version Checking

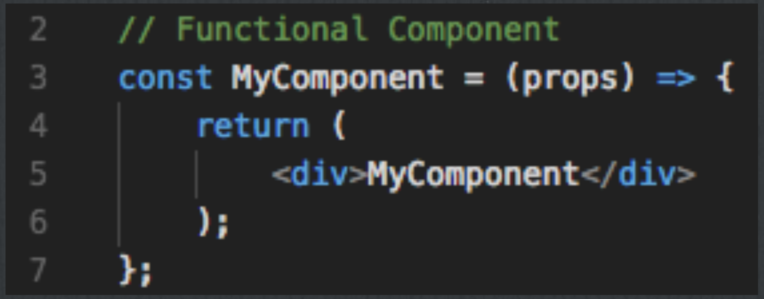
- Solution - Track the running and released versions
  - Prompt user to refresh or force a reload on old version
- Released version - track using a JSON file in the codebase
  - We use public/manifest.json
- Running version - Fetch the JSON file on initial load

Version Checking

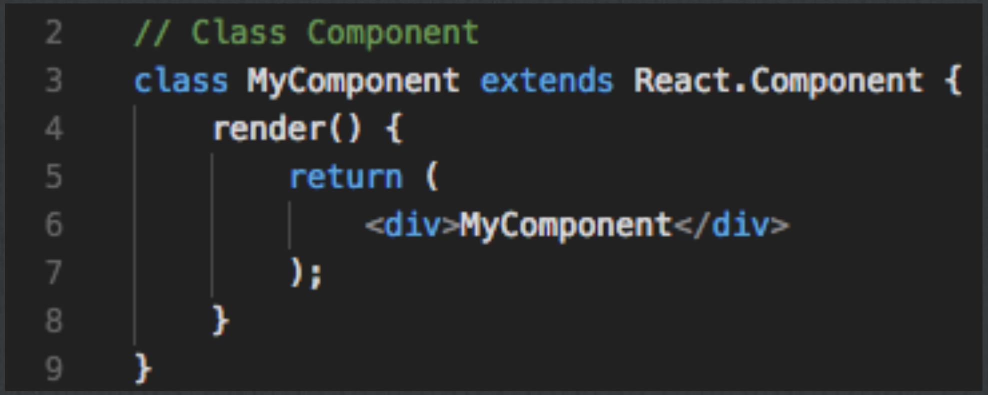
- Periodically fetch the JSON version file to compare versions
    - Trigger on user interactions, on route changes, and/or at intervals
- Make sure the JSON file and index.js are never cached
    - Add randomly generated garbage to the URL like /manifest.json?t=28239828282
- An alternative - backend tracks the released frontend version and compares on API requests
- Why we choose to compare on the frontend
    - No extra database/redis read
    - Don’t have to update/release backend on every frontend change

Function vs Class

- Choose Functions when possible
  - Pros - simpler, easier to understand, more memory efficient, easier to test
  - Cons - Lack lifecycle methods and state.



```
// Functional Component
const MyComponent = (props) => {
    return (
        <div>MyComponent</div>
    );
};
```

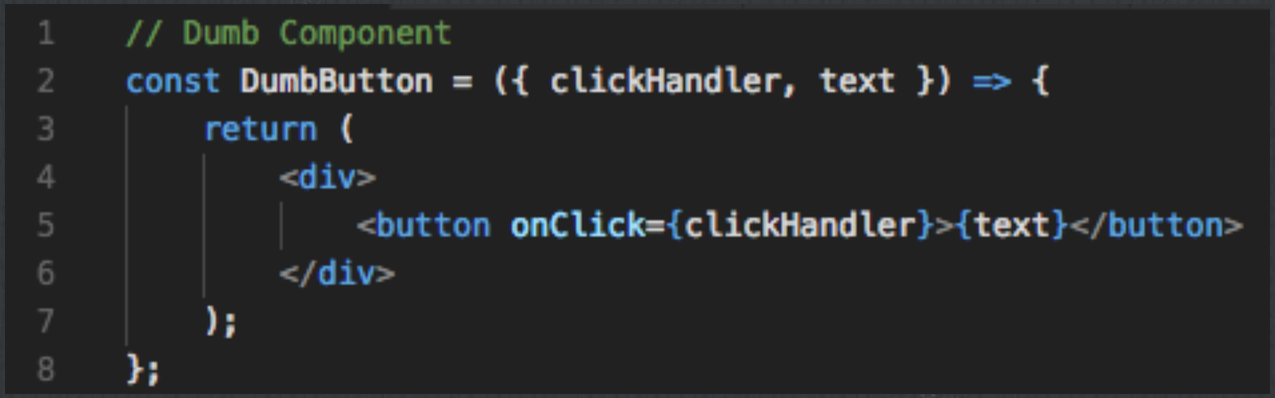


```
// Class Component
class MyComponent extends React.Component {
    render() {
        return (
            <div>MyComponent</div>
        );
    }
}
```

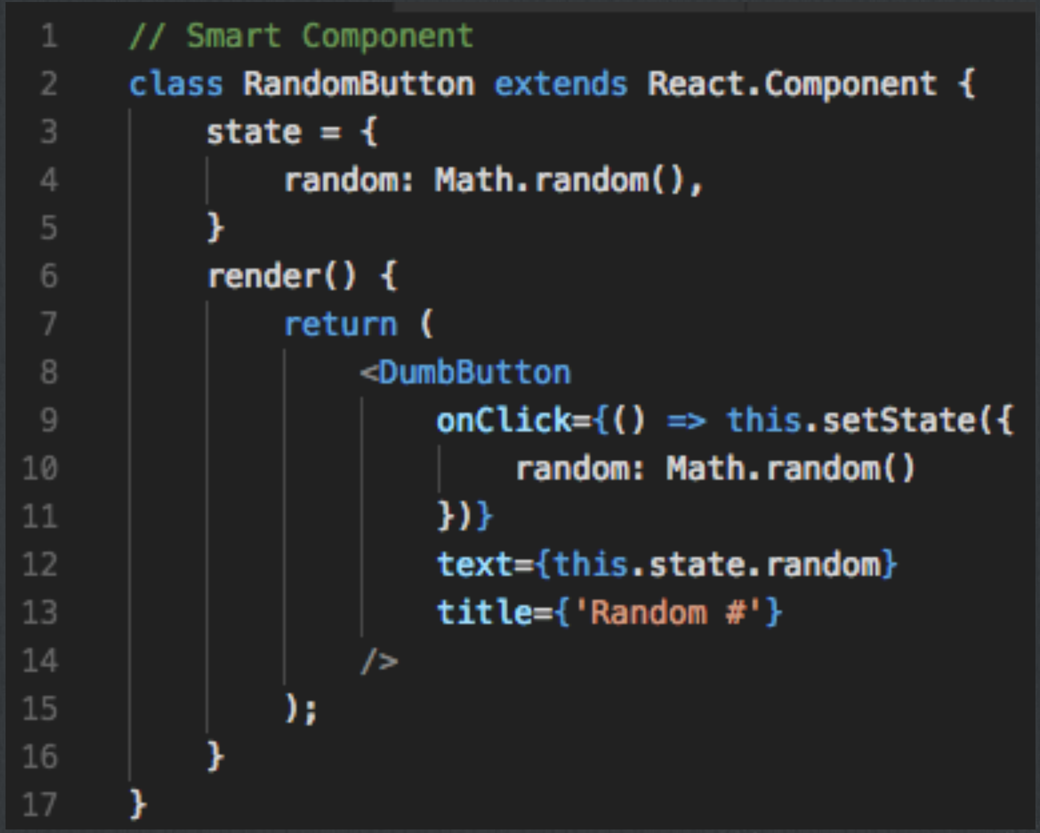
Dumb vs Smart

- Dumb/presentational components present stuff, generally should be pure components.
- Smart/container components manipulate/provide data to other components
- When possible decouple data handling from the markup by creating dumb components
    - Allows using dumb components with multiple smart components

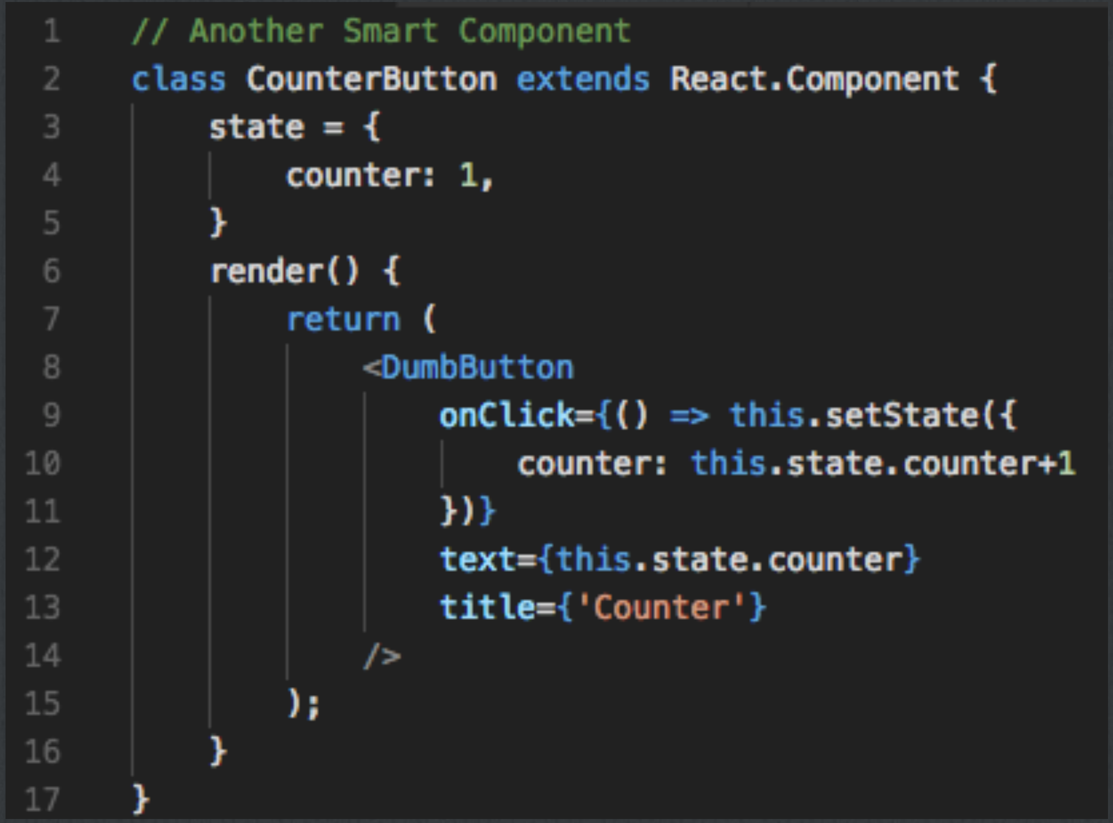
Dumb vs Smart



```
// Dumb Component
const DumbButton = ({ clickHandler, text }) => {
    return (
        <div>
            <button onClick={clickHandler}>{text}</button>
        </div>
    );
};
```



```
// Smart Component
class RandomButton extends React.Component {
    state = {
        random: Math.random(),
    }
    render() {
        return (
            <DumbButton
                onClick={() => this.setState({
                    random: Math.random()
                })}
                text={this.state.random}
                title={'Random #'}
            />
        );
    }
}
```

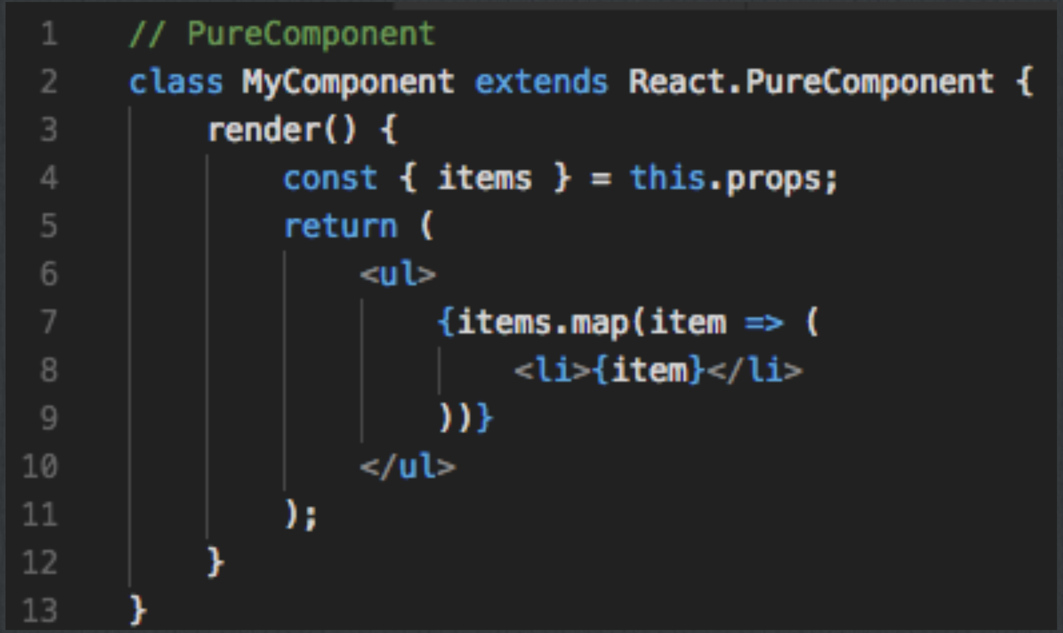


```
// Another Smart Component
class CounterButton extends React.Component {
    state = {
        counter: 1,
    }
    render() {
        return (
            <DumbButton
                onClick={() => this.setState({
                    counter: this.state.counter+1
                })}
                text={this.state.counter}
                title={'Counter'}
            />
        );
    }
}
```

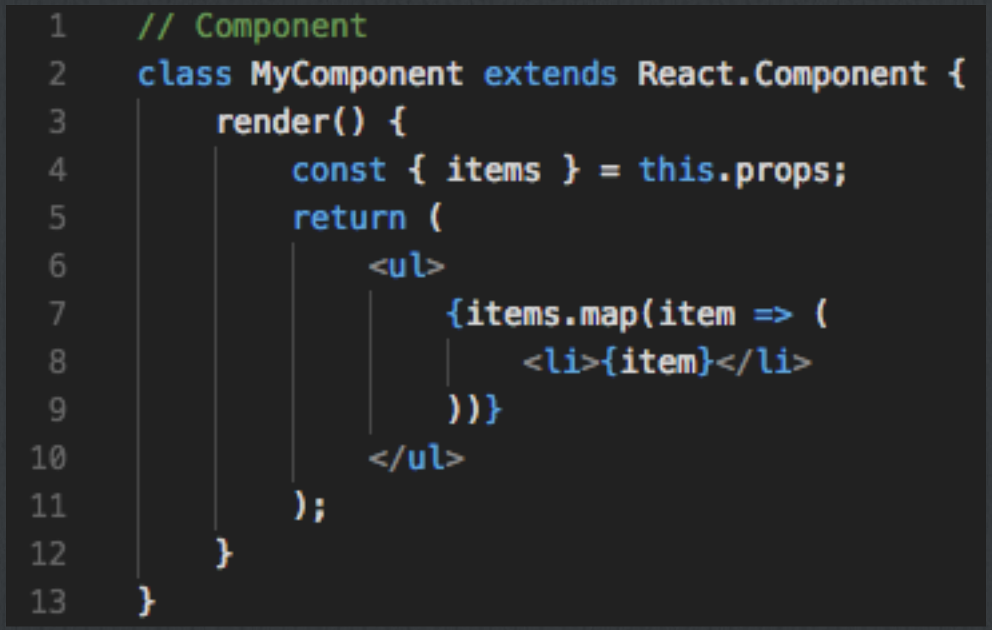
PureComponent vs Component

- Use PureComponent when possible
  - Only re-renders when data has changed.
  - Works great with immutable data
  - Improves performance, prevents unnecessary re-renders
  - Easy to add - one line modified

PureComponent vs Component



```
1   // PureComponent
2   class MyComponent extends React.PureComponent {
3       render() {
4           const { items } = this.props;
5           return (
6               <ul>
7                   {items.map(item => (
8                       <li>{item}</li>
9                   ))}
10              </ul>
11          );
12      }
13  }
```



```
1   // Component
2   class MyComponent extends React.Component {
3       render() {
4           const { items } = this.props;
5           return (
6               <ul>
7                   {items.map(item => (
8                       <li>{item}</li>
9                   ))}
10              </ul>
11          );
12      }
13  }
```

- Only line 2 changed
- The big change happens in shouldComponentUpdate()
  - Returns True by default
  - PureComponent overrides this with a shallow compare

Side effects
Thunks vs Sagas vs Epics

- Side effects = async API calls
- Thunks (redux-thunk)
  - Simple, but lack flexibility
- Sagas (redux-saga)
  - Flexibility - taking actions when you want
  - Fit into redux flow well
- Epics (redux-epic)
  - Flexible
  - Streams can add complexity
- We choose Sagas/Epics over Thunks for added flexibility/features

Routers - History

- React Router was the first go-to routing solution.
- Redux introduced separate application and routing state
- react-router-redux introduced the concept of multiple sources of props where state was split between redux and within the URL
- Redux Little Router took the React Router philosophy but moved routing state into Redux' application state.
- Redux-First Router took it another step by removing routing components: <Route /> and <Fragment />

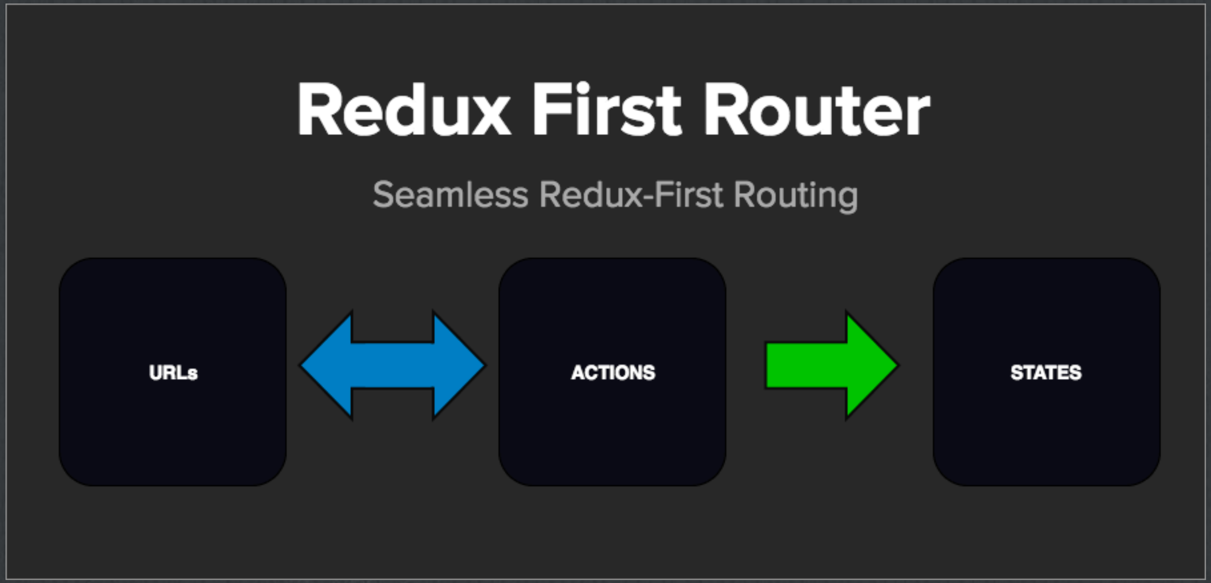
React Router

- We have used this in past projects (even with Redux)
- Obvious choice for applications not using Redux.

Redux Little Router

- Good alternative to React Router if Redux-First Router didn't exist

Redux-First Router

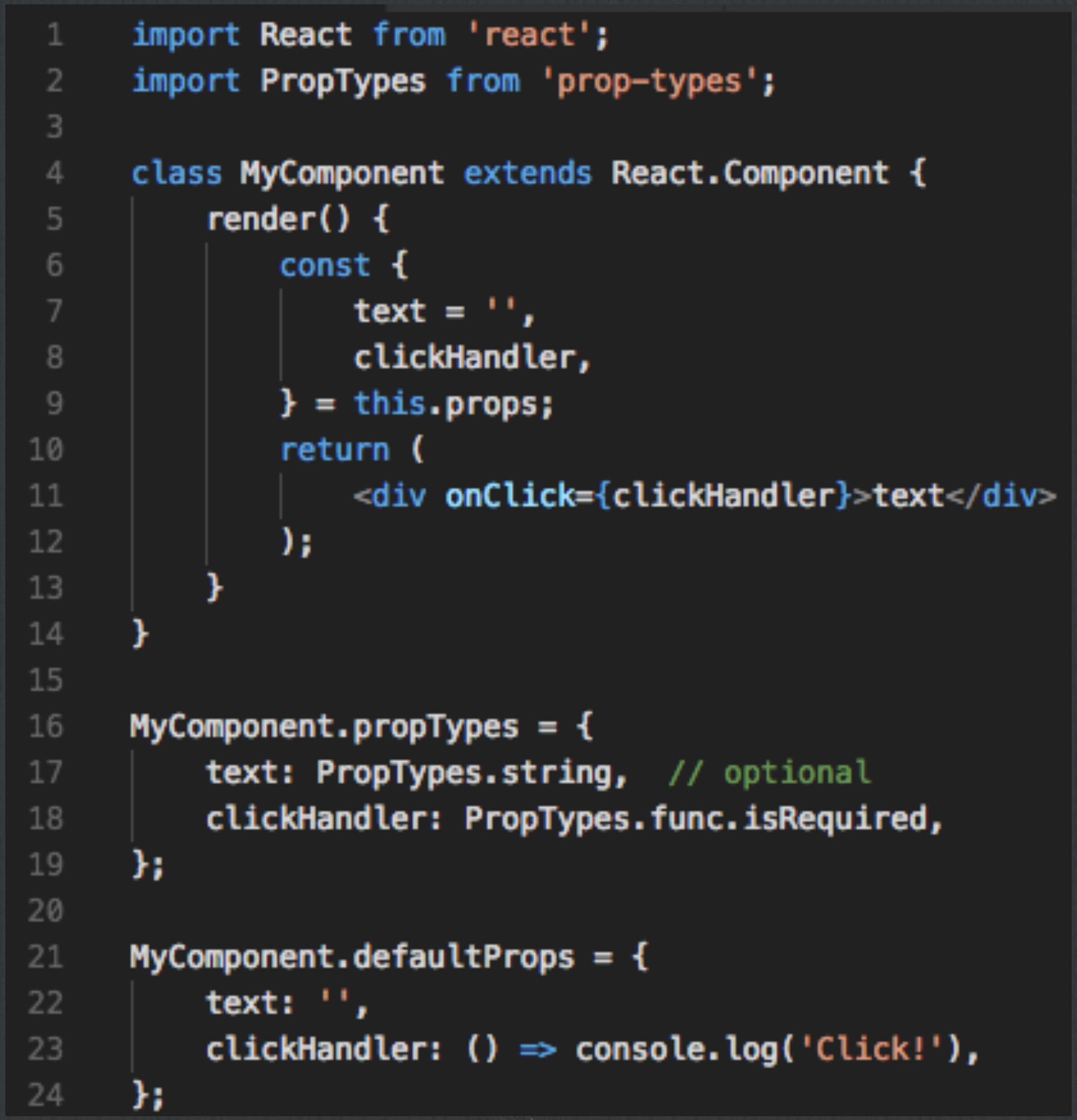


- ☐ Our preference w/ Redux
- ☐ Fits seamlessly into the Redux store
- ☐ Trigger side effects on *specific* route changes
- ☐ Every route change has a different action type (compared to Redux Little Router's single action type)
- ☐ History of a user's route changes
- ☐ We use action creators to do stuff like goHome() or goVideoDetail(video_id)

Testing

- No opinion on libraries
- create-react-app comes with Jest
- How much testing is good enough?
  - 100%!! But that's never practical/realistic
- Prioritize
  - Complex code
  - "Popular" code
  - Low-hanging fruit
  - Tests for bug fixes

Documentation in Code



```
import React from 'react';
import PropTypes from 'prop-types';

class MyComponent extends React.Component {
    render() {
        const {
            text = '',
            clickHandler,
        } = this.props;
        return (
            <div onClick={clickHandler}>text</div>
        );
    }
}

MyComponent.propTypes = {
    text: PropTypes.string,  // optional
    clickHandler: PropTypes.func.isRequired,
};

MyComponent.defaultProps = {
    text: '',
    clickHandler: () => console.log('Click!'),
};
```

- PropTypes (prop-types)
  - Can prevent logic errors
  - Documents in simple, readable code
- defaultProps
  - Set defaults in a standard way
  - Evaluated by PropTypes
  - Override defaults by passing ‘null’

Documentation in Code

- Typescript
  - Overkill on most smaller projects
  - Factor in client's technical abilities
  - Easier dev on-boarding on large projects
  - @types can be missing for some libraries
  - Our friend "any" has come to the rescue many times.
- JSDoc
  - Alternative to TypeScript
  - VS Code supports JSDoc
  - Most common standard for documenting JS code

Q&A

- ☐ Those are our opinions - what are yours?
- ☐ What did we miss?

Thanks! Connect with us!
We would love to build your next app

A Software Presentation From ZERRTECH

Jeremy Zerr

Site: https://www.zerrtech.com
https://www.linkedin.com/in/jrzerr
https://twitter.com/jrzerr

Josh Taylor

Site: https://www.zerrtech.com
https://www.linkedin.com/in/jj-taylor/
https://twitter.com/joshuaj_taylor