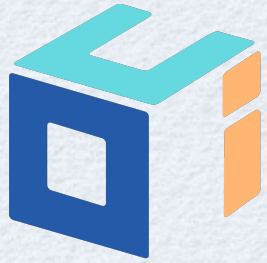
React Simply

slides at https://github.com/mvolkmann/talks

R. Mark Volkmann
Object Computing, Inc.
http://objectcomputing.com
Email: mark@objectcompuing.com
Twitter: @mark_volkmann
GitHub: mvolkmann



OCI | TRAINING

A Love Story

- I initially fell in love with React because of its simplicity
- But a surprising thing happened
- Many React developers created and adopted add-ons that ramped up the complexity
- These are great additions for the right kinds of projects
- However, I am convinced that they are not needed for most applications
- Let's explore how we can keep it simple and something we can love!

Topics

This talk assumes you are already familiar with React and want to learn how to make using it easier.

- **create-react-app** for a great start on new web apps
- **Sass** for CSS preprocessing
- **class public fields** to remove need for pre-binding
- **Prettier** for automated, consistent code formatting
- **ESLint** for JavaScript linting
- **Flow** for adding types to JavaScript
- **Jest** and **Enzyme** for tests, including code coverage
- **Husky** for Git hooks
- **CircleCI** for continuous integration

- **async** and **await** for asynchronous operations like REST calls
- Managing **CSS**
- Managing **routes** without React Router
- Managing **state** without and with Redux

WARNING!
You won't agree with all my opinions.
That's okay.

create-react-app

https://github.com/facebookincubator/create-react-app

- Tool that creates a great starting point for new React apps
- **`npm install -g create-react-app`**
- **`create-react-app app-name`**
  - takes about 20 seconds to complete because it downloads and installs many npm packages
- **`cd app-name`**
- **`npm start`**
  - starts local HTTP server
  - opens default browser to local app URL

Benefits of create-react-app

- Creates directory structure and files including `package.json`
- Installs and configures many tools and libraries
- Provides a local web server for use in development
- Provides **watch and live reload**
- Uses **Jest** test framework which supports **snapshot tests**
- Lets Facebook maintain the build process
  - future benefits from future improvements
- Produces small production deploys

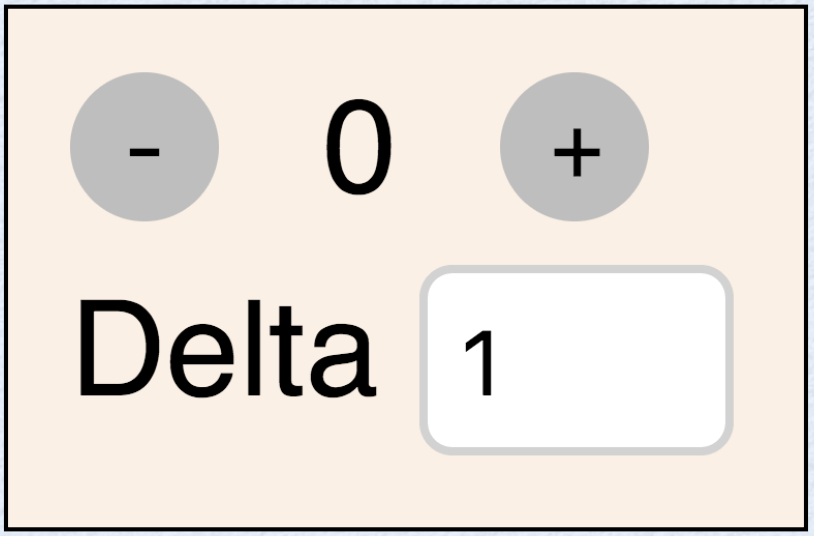
Notable Packages Installed

- **Babel** - JavaScript transpiler (ES6+ to ES5) and more
- **ESLint** - pluggable JavaScript linter
- **Istanbul** - code coverage tool
- **Jest** - JavaScript test framework supporting snapshot tests
- **Lodash** - JavaScript utility library
- **PostCSS** - tool for transforming styles with plugins
  - "can lint CSS, support variables and mixins, transpile future CSS syntax, inline images, and more"
- **React** - of course
- **ReactDOM** - provides DOM-specific methods
- **react-scripts** - scripts and configuration used by create-react-app
  - source of future benefits
- **SockJS** - WebSocket emulation (tries to use native WebSockets first)
- **UglifyJS** — JavaScript parser/compressor/beautifier
- **Webpack** - module and asset bundler
- **webpack-dev-server** - an Express server that server a webpack bundle
- **whatwg-fetch** - polyfill for Fetch API used to make REST calls

Example App

https://github.com/mvolkmann/react-redux-demo

- Keeping it simple so we can focus on the tools
- Demo time!
  - `cd redux-demo`
  - `npm start`



Sass

http://sass-lang.com

- Syntactically Awesome Style Sheets
- A very popular CSS preprocessor
- Supports variables, nested rules, mixins, and more
- Integrating Sass with an app based on create-react-app requires some setup described on the next slide

Using Sass with create-react-app

- Install **`node-sass`** and **`npm-run-all`**
  - **`npm install --save-dev node-sass npm-run-all`**
- Add these npm scripts to **`package.json`**
  - **`"build-css": "node-sass src/ -o src/",`**
  - **`"watch-css": "npm run build-css && node-sass src/ -o src/ --watch",`**
  - **`"start-js": "react-scripts start",`**
- Replace existing npm scripts in **`package.json`** with these
  - **`"start": "npm-run-all -p watch-css start-js",`**
  - **`"build": "npm run build-css && react-scripts build",`**
- Add to **`.gitignore`**
  - **`src/**/*.css`**
- If there are existing **`.css`** files,
  rename them to **`.scss`** and remove **`.css`** files from git

  ex. **`git mv src/App.css src/App.scss`**

CSS Recommendations ...

- Top element of every component should have
  a CSS class whose name matches the component
- Create a separate CSS file for each component
  that specifies its default styling
  and import it into the component
- Use a CSS preprocessor like Sass that supports nested rules
- Have one rule in component CSS files that
  matches class of top element and wraps all other rules
  - greatly reduces rule conflicts
- Create one application-wide CSS file
  that provides global styling and
  can override component styles when needed

… CSS Recommendations

my-component.js

```
import './my-component.css';

class MyComponent extends Component {
  render() {
    return (
      <div className="my-component">
        ...
      </div>
    );
  }
}
```

class-based component

```
const MyComponent = props => (
  <div className="my-component">
    ...
  </div>
);
```

function-based component

my-component.css

```
.my-component {
  .some-nested-class {
    color: blue;
  }
}
```

This approach makes it easy for designers to style the app.

Class Public Fields

- Avoids "bind" issue for event handling functions
- TC39 Stage 2 proposal
- Supported by Babel and create-react-app now

```
class Counter extends Component {

  onDecrement = () =>
    this.props.dispatch({type: 'decrement'});
  onIncrement = () =>
    this.props.dispatch({type: 'increment'});

  render() {
    const {counter} = this.props;
    return (
      <div className="counter">
        <div className="button-row">
          <button className="dec-btn"
            onClick={this.onDecrement}>
            -
          </button>
          {counter}
          <button className="inc-btn"
            onClick={this.onIncrement}>
            +
          </button>
        </div>
        <Delta />
      </div>
    );
  }
}
```

ESLint

http://eslint.org/

- "The pluggable linting utility for JavaScript and JSX"
- Reports many syntax errors and potential run-time errors
- Reports deviations from specified coding guidelines
- Error messages identify violated rules,
  making it easy to adjust them if you disagree
- Has **`--fix`** mode that can fix violations of many rules
  - modifies source files
- **`npm install -D eslint`**
- To use from an npm script, add following to **`package.json`**

```
"lint": "eslint --quiet src -ext .js",
```

`--quiet` only reports errors

- Editor/IDE integrations available
  - Atom, Eclipse, emacs, Intellij IDEA, Sublime, Visual Studio Code, Vim, WebStorm

ESLint Rules

- No rules are enforced by default
- Desired rules must be configured
- See list of current rules at http://eslint.org/docs/rules/
- Configuration file formats supported
  - JSON - **`.eslintrc.json`**; can include JavaScript comments; most popular
  - JavaScript - **`.eslintrc.js`**
  - YAML - **`.eslintrc.yaml`**
  - inside **`package.json`** using **`eslintConfig`** property
  - use of **`.eslintrc`** containing JSON or YAML is deprecated
- Searches upward from current directory for these files
  - combines settings in all configuration files found with settings in closest taking precedence
  - configuration file in home directory is only used if no other configuration files are found

see mine at https://github.com/mvolkmann/MyUnixEnv/blob/master/.eslintrc.json

ESLint Demo

- See **`lint`** script in **`package.json`**
- Modify **`counter.js`**
  - remove semicolon from end of **`PropsType`** definition
  - remove "**`t`**" at end of "**`extends Component`**"
- **`npm run lint`**

Prettier

https://github.com/prettier/prettier

- "An opinionated JavaScript formatter ...
  with advanced support for language features
  from ES2017, JSX, Flow, TypeScript, CSS, LESS, and SCSS"
- "Parses your JavaScript into an AST and pretty-prints the AST, completely ignoring any of the original formatting"
- **`npm install -D prettier`**
- To use from an npm script, add following to **`package.json`**

```
"format": "prettier --no-bracket-spacing --single-quote --write src/**/*.{css,js}",
```

  - to format all `.js` files under `src` directory, enter **`npm run format`**
  - overwrites existing files with formatted versions
- Doesn't run on files under **`node_modules`** by default
- Editor/IDE integrations available
  - Atom, Emacs, JetBrains, Sublime, Vim, Visual Studio Code

Prettier Options

- **`--jsx-bracket-same-line`**
  - puts closing > of JSX start tags on last line instead of on new line

```
<something
  prop1="value1"
  prop2="value1"
  prop3="value1"
  prop4="value1"
>
  content
</something>
```

VS.

```
<something
  prop1="value1"
  prop2="value1"
  prop3="value1"
  prop4="value1">
  content
</something>
```

- **`--no-bracket-spacing`**
  - omits spaces between brackets in object literals

```
{ foo='1' bar=true }
```

VS.

```
{foo='1' bar=true}
```

- **`--no-semi`** - omits semicolons
- **`--print-width n`** - defaults to 80
- **`--single-quote`**
  - uses single quotes instead of double quotes for string delimiters
- **`--tab-width n`** - defaults to 2
- **`--trailing-comma`**
  - adds trailing commas wherever possible; defaults to none
- **`--use-tabs`** - uses tabs instead of spaces for indentation
- and more lesser used options

prettier-eslint-cli

https://github.com/prettier/prettier-eslint-cli

- Command-line interface to prettier-eslint
- "Formats your JavaScript using **`prettier`** followed by **`eslint --fix`**"
- "Get the benefits of Prettier's superior formatting capabilities, but also benefit from the configuration capabilities of ESLint"
- **`npm install -D prettier-eslint-cli`**
- To use from an npm script, add following to **`package.json`**

```
"format": "prettier-eslint --no-bracket-spacing --single-quote --write src/**/*.js",
```

  - to format all `.js` files under `src` directory, enter **`npm run format`**

Prettier and CSS

- While Prettier can process CSS files, ESLint cannot
- So it doesn't make sense to run prettier-eslint-cli on CSS files
- Consider adding a separate npm script like

```
"format-css": "prettier --write src/**/*.css",
```

- If using Sass
  - no need to format generated CSS files

Prettier Demo

- See **`format`** script in **`package.json`**
- Modify **`counter.js`**
  - remove all semicolons
  - mess up lots of indentation
  - change "**`dec-btn`**" to be defined on one line
  - break an arrow function after the arrow so it is on two lines
  - remove parens from **`return`** statement in **`render`** method
- **`npm run format`** and reload file in editor to see changes or trigger from editor/IDE plugin

Why Use Types?

- Can find type errors before runtime
  - more convenient than waiting until runtime
- Types document expectations about code
  - types of variables, object properties, function parameters, and function return types
  - comments can be used instead, but those
    - are more verbose
    - tend to be applied inconsistently
    - easily go out of date when code is updated
- Increases refactoring confidence
  - don't have to wonder what assumptions callers made about supported types
- Removes need to write ...
  - error checking code for type violations
  - type-related unit tests
- Editor/IDE plugins can use types to highlight issues and provide code completion

Why Avoid Types?

- Takes time to ...
  - learn type syntax
  - master applying them
- Makes code more verbose
- Can hamper prototyping and rapid development
  - developers can lose focus when distracted by having to satisfy a compiler or type checker

When to Use Types

- Use types when
  - application is large, complex, or critical
  - expected lifetime of code is long and refactoring is likely
  - code will be written and maintained by a team of developers
- Avoid types when
  - the conditions above are not present

Flow

https://flow.org/

- "A static type checker, designed to find type errors in JavaScript programs"
- Open source tool from Facebook
- Catches many errors without types
  - using **type inference** and **flow analysis**
  - "precisely tracks the types of variables as they flow through the program"
- Can gradually add types
- Most ES6+ features are supported
  - for a list, see https://github.com/facebook/flow/issues/560
- Supports React and JSX
- Editor/IDE integrations available
  - Atom, emacs, Sublime, Visual Studio Code, Vim, WebStorm
- Too much to say about this
  - see slides at https://github.com/mvolkmann/flow-material

Flow Demo

- See **`flow`** script in **`package.json`**
- Modify **`counter.js`**
  - comment out declaration of **`counter`** in **`PropsType`**
  - see definitions of **`DispatchType`** and **`StateType`** in **`types.js`**
  - change all occurrences of "**`counter`**" to "**`count`**"
  - in **`onDecrement`** method, change "**`type`**" to "**`kind`**"
- **`npm run flow`**
  or see errors provided by editor/IDE plugin

Jest

https://facebook.github.io/jest/

- A JavaScript test framework "built on top of Jasmine"
- "Runs your tests with a fake DOM implementation (via jsdom) so that your tests can run on the command line"
- Watches source and test files and automatically reruns tests when they change
  - can run all tests or only those that failed in last run
- Support snapshot tests
  - more on next slide
- Can use to test React components
  - but isn't specific to React
- Default test framework of apps created with create-react-app

Jest Snapshot Tests

- Snapshot tests assert that ...
  - a component will render same content as last successful test
- The first time snapshot tests are run ...
  - `toMatchSnapshot` matchers save a representation of the rendered output in a subdirectory of the test file named `__snapshots__`
- In subsequent runs ...
  - the same representation is generated again and compared to what was saved in last successful run
- When snapshot tests fail ...
  - scroll back to review differences in rendered output
  - if changes are correct, press "u" to accept them
    - overwrites previous snapshot files with new ones
  - if changes are incorrect, fix code and run tests again
- Requires react-test-renderer
  - `npm install -D react-test-renderer`

`__snapshot__` directories should be checked into version control

Jest Watch Mode

- Can iteratively change code being tested and tests and have tests rerun automatically on save from any editor/IDE
- Can filter tests to run on filenames
  - press "p" and enter a regex pattern to filter
  - press "a" to return to running all tests

Enzyme

http://airbnb.io/enzyme/

- Great for testing user interactions with components
- **`npm install -D enzyme`**
- Steps
  - render a component with **`mount`**, **`render`**, or **`shallow`**
    - these return a wrapper object representing what was rendered
  - **`find`** an input element whose interaction will be tested
    - by calling **`find`** on wrapper object
    - supports a subset of CSS selectors
  - **`simulate`** an event on it
    - by calling **`simulate`** on wrapper returned by **`find`**
  - make assertions about changes that should occur
    - can use **`expect`** from Jest

`render` performs static rendering.
This generates static HTML.
Assertions can only test
what is rendered.

`shallow` performs shallow rendering.
The component and its
top-level children are rendered,
but not their descendants.
Assertions can test what
the parent renders and can
simulate events on those elements.

`mount` performs full rendering.
The top component and
all its ancestors are rendered.
Assertions can test everything
that is rendered and
simulate events on everything.

Jest/Enzyme Example

This example assumes a React application that uses Redux.

```
// @flow
import React from 'react';
import Counter from './counter';
import {Provider} from 'react-redux';
import configureStore from 'redux-mock-store';
import {mount} from 'enzyme';
import renderer from 'react-test-renderer';
import './types';

describe('Counter', () => {
  let store;

  beforeEach(() => {
    const mockStore = configureStore();
    const initialState =
      {counter: 0, delta: 1};
    store = mockStore(initialState);
  });

  test('should match snapshot', () => {
    const tree = renderer
      .create(
        <Provider store={store}>
          <Counter />
        </Provider>
      )
      .toJSON();
    expect(tree).toMatchSnapshot();
  });

  test('should decrement', () => {
    const wrapper = mount(
      <Provider store={store}>
        <Counter />
      </Provider>
    );
    const btn = wrapper.find('.dec-btn');
    btn.simulate('click');
    const actions = store.getActions();
    expect(actions[0])
      .toEqual({type: 'decrement'});
  });

  test('should increment', () => {
    const wrapper = mount(
      <Provider store={store}>
        <Counter />
      </Provider>
    );
    const btn = wrapper.find('.inc-btn');
    btn.simulate('click');
    const actions = store.getActions();
    expect(actions[0])
      .toEqual({type: 'increment'});
  });
});
```

verifies that when the user interacts with the UI in a certain way, the expected Redux actions are dispatched

Jest/Enzyme Demo

- See **`test`** script in **`package.json`**
- **`npm t`**
  - runs tests in watch mode
- Modify **`counter.js`**
  - change **`dec-btn`** to render "**`decrement`**" instead of "**`-`**"
  - in **`onDecrement`** method, change value of **`type`** to '**`minus`**'
  - change **`onIncrement`** method to just output "**`incrementing`**" and not dispatch an action
  - note errors when tests run
  - fix errors one at a time
  - press "**`w`**" to see options
  - press "**`q`**" to quit

Code Coverage

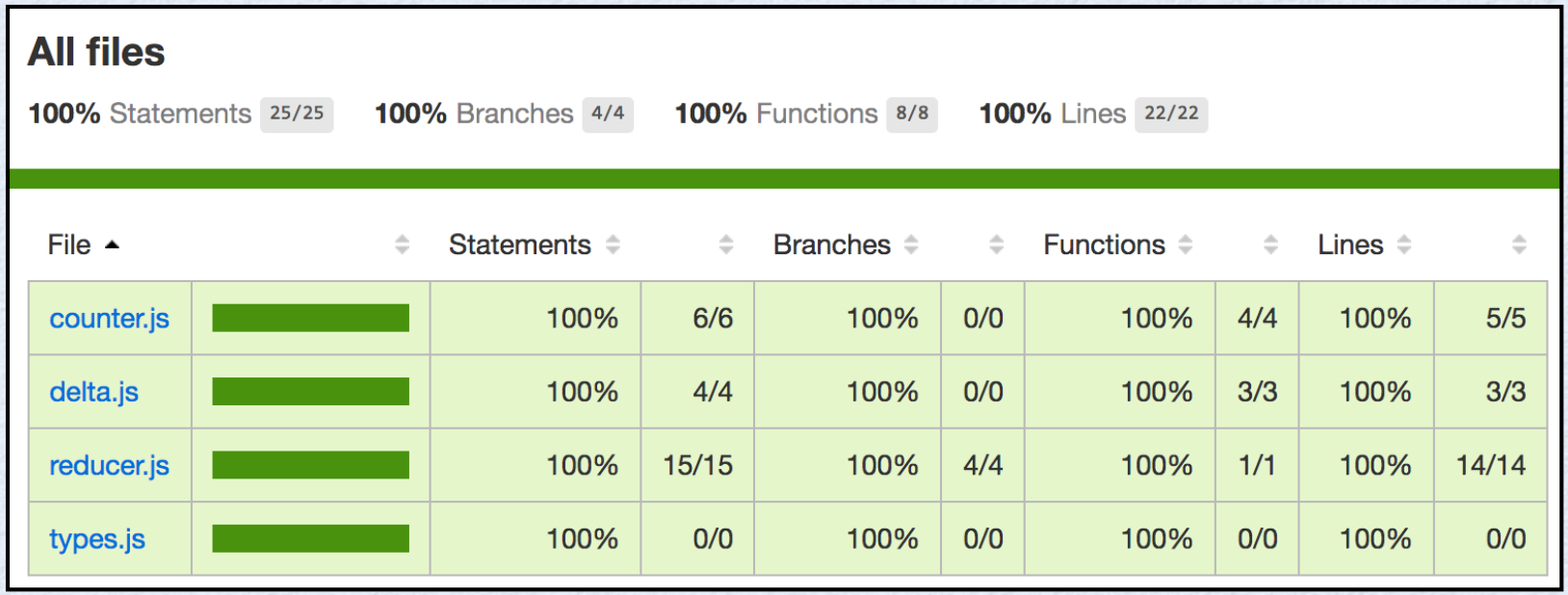
- Jest can report on code coverage of tests using Istanbul
- Can configure to fail if coverage is below specified thresholds
- **`package.json`** changes

```
"jest": {
  "collectCoverageFrom": ["src/**/*.js", "!src/index.js"],
  "coverageThreshold": {
    "global": {
      "branches": 100,
      "functions": 100,
      "lines": 100,
      "statements": 100
    }
  }
},

"scripts": {
  ...
  "cover": "cross-env CI=true npm test -- --coverage",
  "cover-open": "open coverage/lcov-report/index.html",
  "verify": "npm-run-all lint flow format cover",
  ...
}
```

Coverage Demo ...

- See **`cover`**, **`cover`**, **`cover-open`**, and **`verify`** scripts in **`package.json`**
- **`npm run cover`**



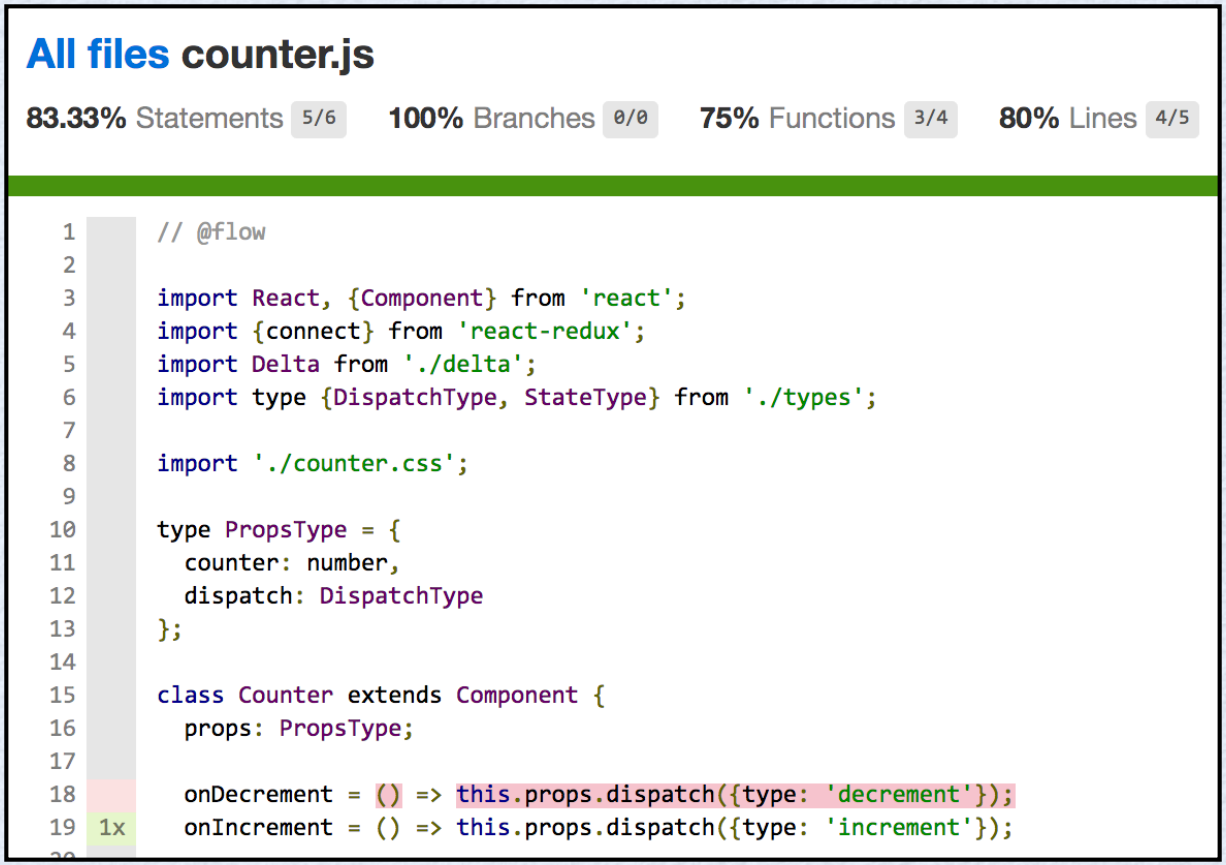
All files

100% Statements 25/25 **100%** Branches 4/4 **100%** Functions 8/8 **100%** Lines 22/22

File	Statements		Branches		Functions		Lines	
counter.js	100%	6/6	100%	0/0	100%	4/4	100%	5/5
delta.js	100%	4/4	100%	0/0	100%	3/3	100%	3/3
reducer.js	100%	15/15	100%	4/4	100%	1/1	100%	14/14
types.js	100%	0/0	100%	0/0	100%	0/0	100%	0/0

… Coverage Demo

- Change "should decrement" **`test`** in **`counter.test.js`** to **`test.skip`**
- **`npm run cover`**
- **`npm run cover-open`**
- Click **`counter.js`** to see detail



Husky

https://github.com/typicode/husky

- "Git hooks made easy"
  - `npm install -D husky`
- One use is to configure a Git hook for push that runs ESLint, Flow, Prettier, and tests and doesn't push if any of those fail
- In `package.json`

```
"scripts": {
  ...
  "prepush": "npm run verify",
  "test-no-watch": "cross-env CI=true npm test -- --verbose",
  "verify": "npm-run-all lint flow format test-no-watch",
  ...
 }
```

- Can bypass
  - `git push --no-verify`
  - mostly useful to push to own branch rather than `master`

```
alias pushn='git push --no-verify origin `git rev-parse --abbrev-ref HEAD`'
```

Husky Demo

- See **`prepush`** and **`verify`** scripts in **`package.json`**
- **`git push`**
  - runs the **`lint`**, **`flow`**, **`format`**, and **`test-no-watch`** scripts
  - if any of these fail, the push is not performed

CircleCI

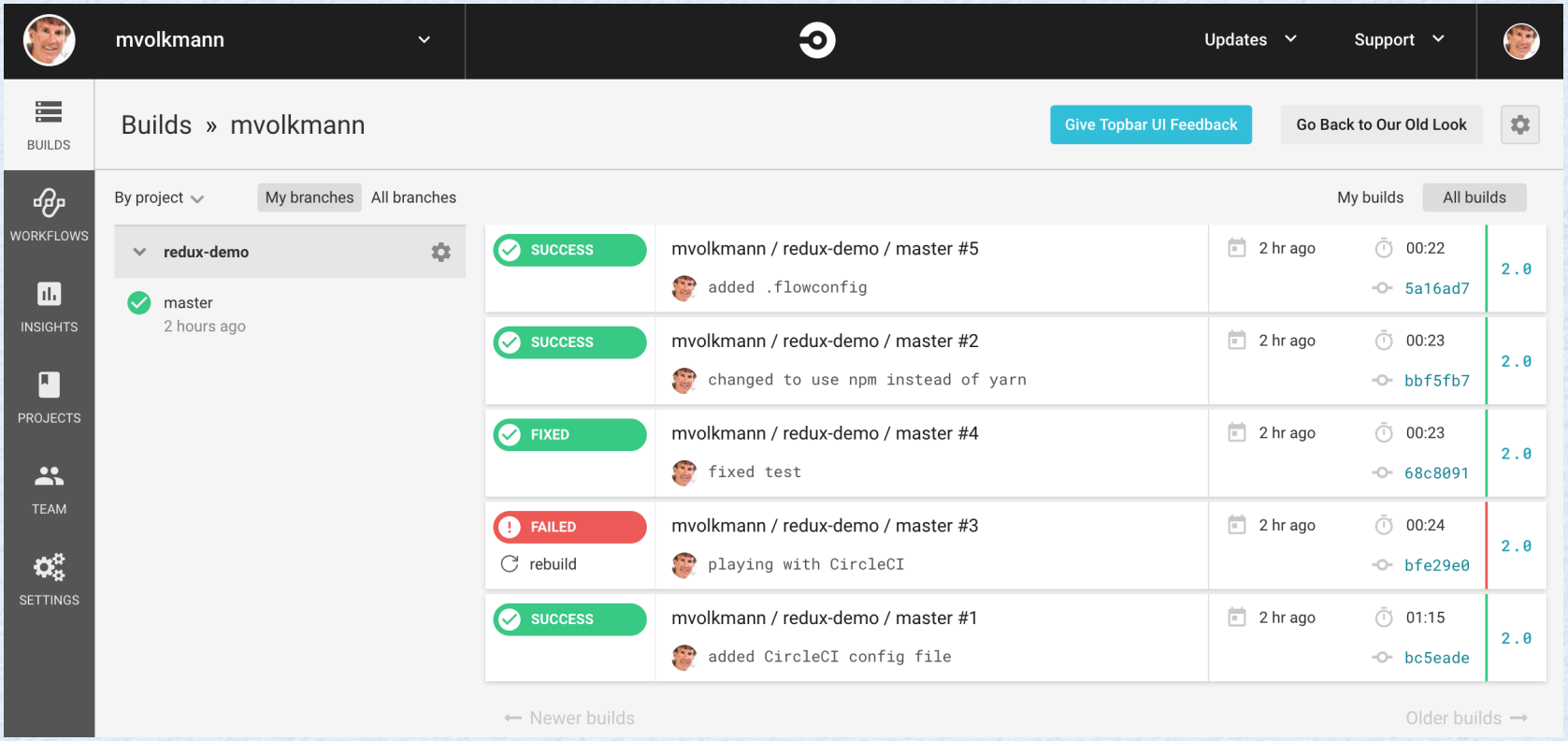
https://circleci.com

- Continuous Integration/Deployment in the cloud
- Benefits
  - verifies that builds are not only working in local environments due to unique setup
  - verifies that builds from teammates
- Free for public repos and a single container
  - additional containers are $50/month
- Configure in minutes!
  - sign up
  - select repos to manage
  - create `.circleci/config.yml`
  - start pushing changes
- Fun Fun Function video
  - https://www.youtube.com/watch?v=7VxBn_ZgOek

Minimal `config.yml`

```
version: 2
jobs:
  build:
    docker:
      - image: circleci/node:7.10
    working_directory: ~/repo
    steps:
      - checkout
      - restore_cache:
          keys:
          - v1-dependencies-{{ checksum "package.json" }}
          # fallback to using latest cache
          # if no exact match is found
          - v1-dependencies-
      - run: npm install
      - save_cache:
          paths:
            - node_modules
          key: v1-dependencies-{{ checksum "package.json" }}
      - run: npm cover
```

CircleCI Web Dashboard



`async` and `await`

- New keywords added to JavaScript in ES2017
- Make it much easier to work with functions that return promises
- Makes writing asynchronous code look similar to writing synchronous code
- Can be used today in browsers by utilizing Babel
- Enabled by default in Node v7.6 and above
- Example on next slide uses Fetch API to make REST calls
  - same technique would apply to any functions that return promises
- **`await`** keyword can be applied to any function call,
  even ones that do not explicitly return promises
  - will return a promise that resolves to the returned value
    or rejects if an error is thrown

Side-by-Side Example

https://github.com/mvolkmann/async-await-screencast

```
const fetch = require('node-fetch');

function demo() {
  const urlPrefix = 'http://localhost:3000';
  const username = 'mvolkmann';
  const storeName = 'Taco Bell';

  let url = `${urlPrefix}/people/${username}/zip`;
  let zip;
  fetch(url)
    .then(res => res.text())
    .then(zipCode => {
      zip = zipCode;
      console.log('zip =', zip);

      url = `${urlPrefix}/stores/locations` +
        `?zip=${zip}&name=${storeName}`;
      return fetch(url);
    })
    .then(res => {
      if (res.status === 404) {
        throw new Error(
          `There are no ${storeName} stores in ${zip}.`);
      }
      return res.json();
    })
    .then(locations => {
      console.log(`${storeName} locations are:`);
      for (const location of locations) {
        console.log(location);
      }
    })
    .catch(e => console.error(e.message));
}

demo();
```

```
const fetch = require('node-fetch');

async function demo() {
  const urlPrefix = 'http://localhost:3000';
  const username = 'mvolkmann';
  const storeName = 'Taco Bell';

  try {
    let url = `${urlPrefix}/people/${username}/zip`;
    let res = await fetch(url);
    const zip = await res.text();
    console.log('zip =', zip);

    url = `${urlPrefix}/stores/locations` +
      `?zip=${zip}&name=${storeName}`;
    res = await fetch(url);
    if (res.status === 404) {
      throw new Error(
        `There are no ${storeName} stores in ${zip}.`);
    }

    const locations = await res.json();
    console.log(`${storeName} locations are:`);
    for (const location of locations) {
      console.log(location);
    }
  } catch (e) {
    console.error(e.message);
  }
}

demo();
```

`async` and `await` Questions

- What happens if **`await`** is used inside a function that is not marked as **`async`**?
  - you'll get a **`SyntaxError`**
- What happens if you call a function marked as **`async`** that returns a promise without using **`await`**?
  - it just returns the promise object without waiting for it to resolve or reject
- What happens if you call a function using **`await`**, but the function is not marked as **`async`**?
  - it returns its value immediately

CSS

- Christopher Chedeau (vjeux)
  - laid out issues with CSS and the case for CSS in JS
    in slides at https://speakerdeck.com/vjeux/react-css-in-js
- Claim
  - placing CSS in JS solves most issues with plain CSS
- Counterclaim
  - most of these issues are only serious in apps that use large amounts of CSS
  - many apps do not
- We will review each of his points and discuss pros and cons

CSS Issue #1: Global Namespace

- CSS rules are global
- Conflicts are resolved by specificity rules
- In ties, the last one wins
- We've agreed that globals are bad in programming languages

- **CSS in JS**: CSS is specified in component source files or imported into it, so it's local rather than global
- **Plain CSS**: rule scoping conventions greatly reduces conflicts
  - we'll show this soon

CSS Issue #2: Dependencies

- Components can include only their CSS (via ES module import)
  to indicate dependencies,
  but CSS from other components can still effect them

- **CSS in JS**: CSS is scoped to components
  and isn't effected by CSS in other components
- **Plain CSS**: rule scoping conventions prevent conflicts
  between component-specific CSS rules

CSS Issue #3: Dead Code Elimination

- No easy way to remove unused CSS rules
  from what is served at runtime,
  or even know which rules are unused

- **CSS in JS**: can use JS approaches to find and eliminate unused rules
- **Plain CSS**: no good answer here, but
  having a reasonable amount of CSS means
  less opportunities for dead code
  - there are tools that do this for CSS, but perhaps not mature yet

CSS Issue #4: Minification

- No easy way to minify CSS rules,
  replacing element/class/property names and property values
  with shorter alternatives

- **CSS is JS**: can use JS approaches such as uglify
- **Plain CSS**: no good answer here, but
  having a reasonable amount of CSS means
  less need for minification

CSS Issue #5: Sharing Constants

- CSS now has variables and CSS preprocessors support them, but constants in JavaScript code can't be accessed in CSS

- **CSS in JS**: can easily share access to JS variables
- **Plain CSS**: having common JavaScript and CSS constants is rare

CSS Issue #6: Non-deterministic Resolution

- If CSS files are loaded asynchronously
  in the order they are needed based on user interaction,
  results are unpredictable

- **CSS in JS**: components can control how non-local CSS is used through imports
- **Plain CSS**: can load all CSS upfront to make it predictable
  which is okay if there is a reasonable amount,
  but not otherwise

CSS Issue #7: Isolation

- Component styling can be overridden from outside components
- Can be based on element nesting and class names used
- Changes to components can break this styling

- **CSS in JS**: colocating CSS and JS addresses this because changes are made in the same file where CSS resides
- **Plain CSS**: relying on CSS classes over element nesting to match elements in components reduces this issue; still depends on class names not being changed or removed

Avoid writing global CSS selectors that make assumptions about element nesting in components!

CSS in JS Issues

- Makes it difficult for designers to style apps
  - often not comfortable with JavaScript
- Styles in JS are a distraction when implementing components
  - clutters code
  - best to think about styles after components are working, not encourage thinking about them too early

like how when building a house, cabinet finishes, paint colors, and flooring can be selected near the end of construction

- Can't use CSS tools like Sass and stylelint
- What needs to be dynamic?
  - HTML in JS is good because it is very common to dynamically control what is rendered
  - not as common to need dynamic CSS, but when needed can dynamically assign CSS classes and `style` properties
- Requires manual cascading
  - accepting style objects as props that are mixed into defaults using `Object.assign` or object spread

Following the conventions recommended earlier for Sass greatly reduces the impact of these issues!

Glamor

- One of the simpler approaches to "CSS in JS"
  - stays closer to the spirit of CSS
  - doesn't force creating new components for every tag that needs styling
  - **DOES GLAMOROUS DO THIS?**
- https://github.com/threepointone/glamor
- `css` function takes an object describing CSS properties and creates a CSS class on the fly
  - class names are "`css-generated-name`" (ex. `css-gnati0`)
  - this is a **downside** because mapping classes in the browser inspector to their source is more difficult - HOW DO YOU DEBUG CSS LIKE THIS?
- Let's convert CSS for default create-react-app to use Glamor!

App.css vs. app-css.js

CSS syntax

```
.App {
  text-align: center;
}

.App-header {
  background-color: #222;
  height: 150px;
  padding: 20px;
  color: white;
}

.App-intro {
  font-size: large;
}

.App-logo {
  animation:
    App-logo-spin infinite 20s linear;
  height: 80px;
}

@keyframes App-logo-spin {
  from { transform: rotate(0deg); }
  to { transform: rotate(360deg); }
}
```

JavaScript syntax

```
import {css} from 'glamor';

export const app = css({
  textAlign: 'center'
});

export const appHeader = css({
  backgroundColor: '#222',
  height: 150,
  padding: 20,
  color: 'white'
});

export const appIntro = css({
  fontSize: 'large'
});

const appLogoSpin = css.keyframes({
  from: {transform: 'rotate(0deg)'},
  to: {transform: 'rotate(360deg)'}
});
export const appLogo = css({
  animation:
    `${appLogoSpin} infinite 20s linear`,
  height: 80
});
```

using JavaScript syntax solves the CSS issues

unit defaults to `px`

note camelCased CSS property names

App.js - Old and New

```
import React, {Component} from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo}
            className="App-logo"
            alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit
          <code>src/App.js</code>
          and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

can't tell which CSS classes are being used

```
import React, {Component} from 'react';
import {
  app, appHeader, appIntro, appLogo
} from './app-css';
import logo from './logo.svg';

class App extends Component {
  render() {
    return (
      <div className={app}>
        <div className={appHeader}>
          <img src={logo}
            className={appLogo}
            alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className={appIntro}>
          To get started, edit
          <code>src/App.js</code>
          and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

dependencies are explicit

alternate syntax

```
<div {...app}>
```

CSS-in-JS Questions

- Do we really want snapshot tests to fail when CSS changes?
- Is there a tool that parses CSS and output equivalent CSS-in-JS code?
- These solutions seem to give up on the possibility that people who don't know JS and React will be able to style an app. Is that acceptable?

Routes

- At their most basic, routes map URLs to views
- react-router is the most popular way to manage routes in React applications
  - https://reacttraining.com/react-router/
  - specifies routes using JSX
  - provides many powerful features
    - server-side rendering
    - code-splitting - only loads imports of a route when it is visited
    - redirects for routes that require authentication
    - animated transitions

many apps don't need these

- Hash-based routing is simpler

Hash-based Routing

- In constructor of top component, listen for **`hashchange`** events generated any time the URL hash changes

```
window.addListener('hashchange', () => this.forceUpdate());
```

  - `forceUpdate` causes `render` to be called when no props or state have changed

- Add `router` method to top component

```
router = () => {
  const {hash} =
    getLocationParts(window.location);
  switch (hash) {
    case 'page1':
      return <Page1 />;
    case 'page2':
      return <Page2 />;
    default:
      return null;
  }
};
```

can also use `path` and `query` to select a component and pass data to it

```
function getLocationParts(loc) {
  return {
    hash: loc.hash.substring(1),
    path: loc.pathname,
    query: new URLSeachParams(loc.search)
  };
}
```

- Call `router` method in `render` method

```
render = () => <div className="app">{this.router()}</div>;
```

Changing Routes

- Using hyperlinks

```
<a href="#page2">Page 2</a>
```

- Using code

```
document.location.href = '#page2';
```

- What features are sacrificed by not using react-router?
  - talk to Jeff Barczewski about this

State

- Redux is the most popular way to manage state in React applications
  - http://redux.js.org/
  - many variations: react-redux, redux-logic, redux-saga, redux-thunk, ...
- Redux adds complexity and libraries on top of it add more
  - action objects, action type constants, action creator functions, dispatching actions, reducers, creating the store, listening for store changes and re-rendering, providers, connected components, sagas, thunks, ...
- "You Might Not Need Redux" article by Dan Abramov
  - https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367
- This complexity can be avoided by just using React `setState`
  - called on an instance of a component to update its state
  - after updating the state, the component and all components it renders are re-rendered
  - done asynchronously and the virtual DOM makes it very efficient

Calling `setState`

- Two ways to call
- 1) With an object
  - `this.setState(someObject);`
  - properties in ***`someObject`*** replace properties in current state via a "shallow merge"
  - any properties in ***`someObject`*** that are not already in the state are added
  - any properties in ***`someObject`*** that are already in the state replace them

```
this.setState({score: 10});
```

- 2) With a function
  - `this.setState(someFunction);`
  - ***`someFunction`*** is passed the current state as an object
  - it must return an object that will be shallow merged into the current state, just like in the first approach

```
this.setState(state => {
  const score = state.score + 1;
  return {score};
});
```

- Choosing
  - if any new state values need to be computed based on current state values, use function approach
  - otherwise use object approach

Using `setState` instead of Redux

- One store
  - Redux holds all application state in one place, called the "store"
  - can instead do this in state of top component
- Dispatching actions
  - in Redux, "actions" can be dispatched from anywhere
  - these typically result in updates to the store
  - to mimic this without Redux,
    make the top component `setState` method available everywhere
  - one approach: `React.setTopState = this.setState.bind(this);`
  - do this in constructor of top component
  - now any component can call `React.setTopState`
- What do we lose?
  - ability to use the Redux Chrome plugin and time travel debugging
  - alternate ways to get state to nested components besides using props

With this approach, components are truly functions of their props (and their own state, if any) which makes them easier to understand.

Redux

- Do use react-redux
  - "Official React bindings for Redux"
  - https://github.com/reactjs/react-redux
- Do use `mapStateToProps`
  - extracts specific state properties and passes them to the component through props
- Don't use `mapDispatchToProps`
  - by default, the `dispatch` function is passed to the component in a prop
  - component will have access to all state properties specified in `mapStateToProps`, so its event handling functions can use them to create payloads needed in calls to `dispatch`
  - a downside is that this makes it explicit that components are using Redux, but it's highly unlikely you'll use them outside of Redux later
- Don't use `mergeProps`
  - unless you enjoy complicated approaches

Redux Example - `index.js`

```
// @flow

import React from 'react';
import ReactDOM from 'react-dom';
import {createStore} from 'redux';
import {Provider} from 'react-redux';

import Counter from './counter';
import reducer from './reducer';

// The only part of this that is application-specific
// is the use of the Counter component.
// Note how no props are passed to Counter.
// It gets all its props from the store using
// mapStateToProps at the bottom of counter.js.
function render(): void {
  ReactDOM.render(
    <Provider store={store}>
      <Counter />
    </Provider>,
    document.getElementById('root')
  );
}

const store = createStore(reducer);
store.subscribe(render);
render();
```

Redux Example - types.js

```
// @flow

export type ActionType = {
  type: string,
  payload?: mixed
};

export type DispatchType =
  (action: ActionType) => void;

export type StateType = {
  counter: number,
  delta: number
};

// This is a copy of StateType
// with all properties optional.
// It is useful for the return
// type of our reducer functions.
export type SubstateType = {
  counter?: number,
  delta?: number
};
```

Redux Example - counter.js

```
// @flow

import React, {Component} from 'react';
import {connect} from 'react-redux';
import Delta from './delta';
import type {DispatchType, StateType}
  from './types';

type PropsType = {
  counter: number,
  dispatch: DispatchType
};

class Counter extends Component {
  props: PropsType;

  onDecrement = () =>
    this.props.dispatch({type: 'decrement'});
  onIncrement = () =>
    this.props.dispatch({type: 'increment'});
```

```
 render() {
    const {counter} = this.props;
    return (
      <div>
        <div>
          <label>Counter = </label>
          {counter}
        </div>
        <div>
          <button className="inc-btn"
            onClick={this.onIncrement}>
            Increment
          </button>
          <button className="dec-btn"
            onClick={this.onDecrement}>
            Decrement
          </button>
        </div>
        <Delta />
      </div>
    );
  }
}

const mapState =
  ({counter}: StateType) => ({counter});
export default connect(mapState)(Counter);
```

Redux Example - `delta.js`

```
// @flow

import React, {Component} from 'react';
import {connect} from 'react-redux';
import type {DispatchType, StateType} from './types';

type PropsType = {
  delta: number,
  dispatch: DispatchType
};

class Delta extends Component {
  props: PropsType;

  onDeltaChange = (e: SyntheticInputEvent) =>
    this.props.dispatch({
      type: 'deltaChange',
      payload: Number(e.target.value)
    });
```

```
 render() {
    return (
      <div>
        <label>Delta</label>
        <input
          type="number"
          onChange={this.onDeltaChange}
          value={this.props.delta}
        />
      </div>
    );
  }
}

const mapState =
  ({delta}: StateType) => ({delta});
export default connect(mapState)(Delta);
```

Redux Example - `reducer.js`

```
// @flow

import type {
  ActionType, StateType, SubstateType
} from './types';

const initialState: StateType = {
  counter: 0,
  delta: 1
};

// In this example, all reducer functions
// are in one file, but we could mix in
// functions from other files here.
const functions = {
  decrement(state: StateType): SubstateType {
    const {counter, delta} = state;
    return {counter: counter - delta};
  },
  deltaChange(
    state: StateType,
    delta: number): SubstateType {
    return {delta};
  },
  increment(state: StateType): SubstateType {
    const {counter, delta} = state;
    return {counter: counter + delta};
  }
};
```

```
function reducer(
  state: StateType,
  action: ActionType): StateType {
  const {payload, type} = action;
  if (type === '@@redux/INIT') {
    return initialState;
  }

  const fn = functions[type];
  if (!fn) {
    console.error(
      `unsupported action type "${type}"`);
  }
  const changes =
    fn ? fn(state, payload) : state;
  return {...state, ...changes};
}

export default reducer;
```

These reducer functions only return state changes to be made, not entire new state.

This approach looks up reducers functions by name rather than using action type constants and a switch statement.

Redux Example - `reducer.test.js`

```
// @flow

import reducer from './reducer';
import type {StateType} from './types';

describe('reducer', () => {
  it('should decrement', () => {
    const state: StateType = {counter: 5, delta: 2};
    const action = {type: 'decrement'};
    const newState = reducer(state, action);
    expect(newState.counter).toBe(3);
  });

  it('should increment', () => {
    const state: StateType = {counter: 5, delta: 2};
    const action = {type: 'increment'};
    const newState = reducer(state, action);
    expect(newState.counter).toBe(7);
  });

  it('should change delta', () => {
    const state: StateType = {counter: 0, delta: 2};
    const action = {type: 'deltaChange', payload: 3};
    const newState = reducer(state, action);
    expect(newState.delta).toBe(3);
  });
});
```

Redux Example - `counter.test.js`

```
// @flow

import React from 'react';
import {Provider} from 'react-redux';
import configureStore
  from 'redux-mock-store';
import {mount} from 'enzyme';
import Counter from './counter';
import './types';

/**
 * These tests just verify that when
 * the user interacts with the UI
 * in a certain way, the expected
 * Redux actions are dispatched.
 */
describe('Counter', () => {
  let store;

  beforeEach(() => {
    const mockStore = configureStore();
    const initialState =
      {counter: 0, delta: 1};
    store = mockStore(initialState);
  });
```

```
  it('should decrement', () => {
    const wrapper = mount(
      <Provider store={store}>
        <Counter />
      </Provider>
    );
    const btn = wrapper.find('.dec-btn');
    btn.simulate('click');

    const actions = store.getActions();
    expect(actions[0])
      .toEqual({type: 'decrement'});
  });

  it('should increment', () => {
    const wrapper = mount(
      <Provider store={store}>
        <Counter />
      </Provider>
    );
    const btn = wrapper.find('.inc-btn');
    btn.simulate('click');

    const actions = store.getActions();
    expect(actions[0])
      .toEqual({type: 'increment'});
  });
});
```

Redux Example - `delta.test.js`

```
// @flow

import React from 'react';
import {Provider} from 'react-redux';
import configureStore
  from 'redux-mock-store';
import {mount} from 'enzyme';
import Delta from './delta';

/**
 * These tests just verify that when
 * the user interacts with the UI
 * in a certain way, the expected
 * Redux actions are dispatched.
 */
describe('Delta', () => {
  let store;

  beforeEach(() => {
    const mockStore = configureStore();
    const initialState = {delta: 1};
    store = mockStore(initialState);
  });
```

```
  it('should dispatch', () => {
    const wrapper = mount(
      <Provider store={store}>
        <Delta />
      </Provider>
    );
    const input = wrapper.find('input');
    input.simulate(
      'change',
      {target: {value: 2}});

    const actions = store.getActions();
    expect(actions[0]).toEqual({
      type: 'deltaChange',
      payload: 2
    });
  });
});
```

Wrap Up

- Configuring tools requires a bit of work,
  but the automation they provide
  is well worth the effort
- Tools can reduce time spent performing tedious tasks
  - like finding bugs, formatting code, and running tests
- Utilizing language features like
  class public fields and async/await
  can simplify code, making it easier to read
- Go forth and automate!