Resilient React.js

Building apps with a functional approach

JazzCon.tech 2018

Jeff Barczewski

- @jeffbski
- jeff@codewinds.com
- https://codewinds.com/jazzcon2018

Jeff Barczewski

- Married, Father, Catholic
- 28 yrs (be nice to the old guy :-)
- JS (since 95, exclusive last 6 years)
- Open Source: redux-logic, pkglink, …
- Founded CodeWinds, live/online training (React, Redux, Immutable, RxJS) – I love teaching and mentoring, contact me

CodeWinds Training

- Live training (in-person or webinar)
- Self-paced video training classes (codewinds.com)
- Need training/mentoring for your team on any of these?
  - React
  - Redux
  - redux-logic
  - RxJS
  - JavaScript
  - Node.js
  - Functional approaches
- I'd love to work with you and your team

PHP CEO
@PHP_CEO

Follow

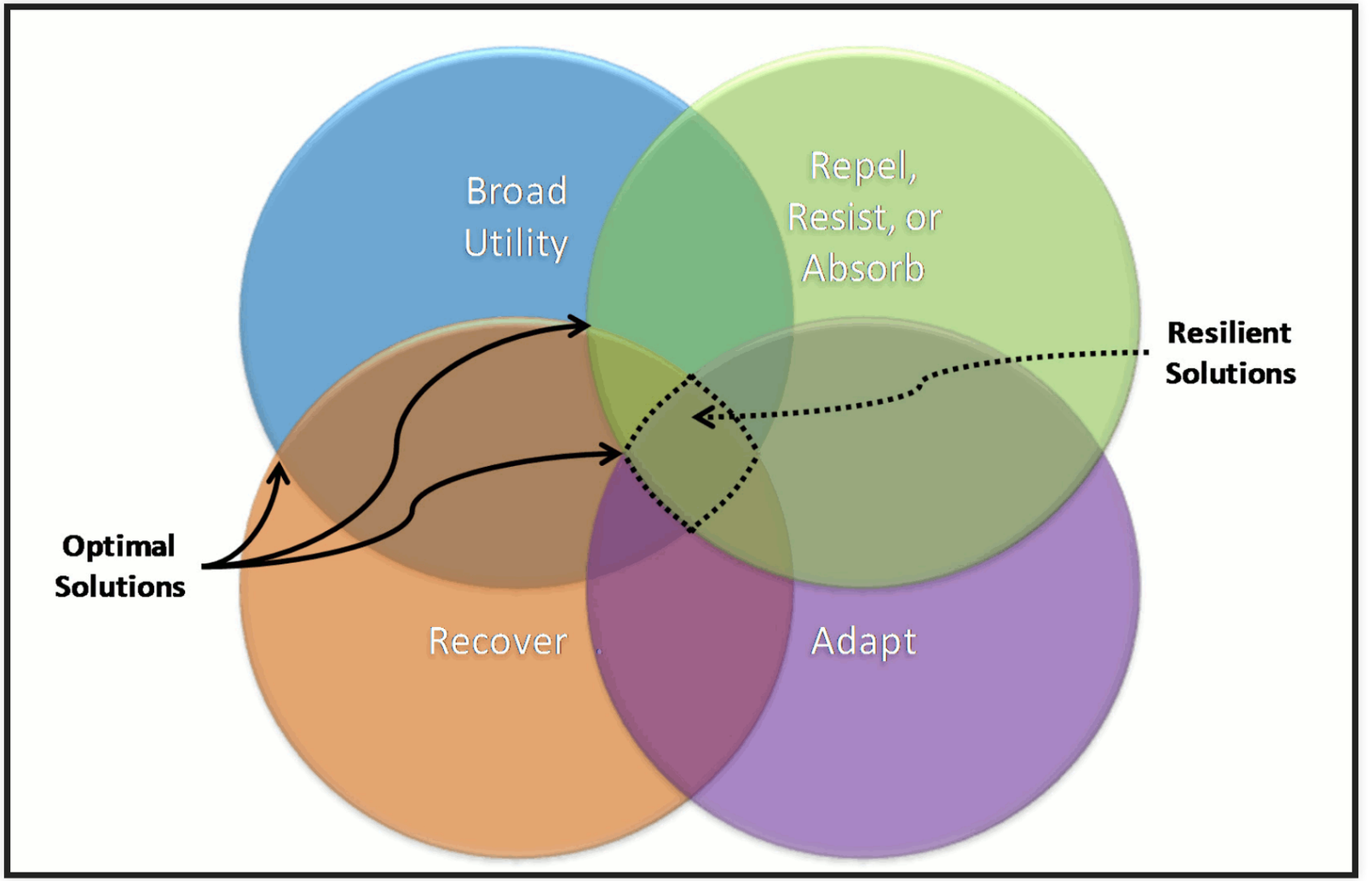
HIRING PRO TIP: HIPPIES ARE NOT THE SAME AS HIPSTERS

I HAVE MADE THIS MISTAKE AND WE NOW HAVE A FRONTEND MVC FRAMEWORK WRITTEN IN FORTRAN

RETWEETS 1,463 FAVORITES 825

6:28 PM - 6 Jun 2014

Resilient Systems




React.js Conf 2015
React.js Conf 2015 - Immutable Data and React
Rich Hickey
Creator of Clojure
Simplicity is hard work. But, there's a huge payoff. The person who has a genuinely simpler system is going to be able to affect the greatest change with the least work. He's going to kick your ass.
"Simple Made Easy" - 2011
13:35 / 31:10
CC
HD
YouTube

Simple vs. easy

"Simplicity is a prerequisite for reliability" - Edsger W. Dijkstra

The roots of "simple" are "sim" and "plex", and means "one twist". The opposite, which would be complex, is "multiple twists" or "braided together".

The central point of the talk is seeing software as being folded together or not.

People usually interchange "simple" with "easy".

Simplicity vs. easiness in software development

In software development, easy means being near to our understanding, being in our skillset, being familiar.

Easy is overrated in the software industry. "Can I get this running in 5 seconds?" It does not matter if the result is a large ball of mud. All it matters is to be done quickly.

All that is new is somewhat unfamiliar. Do not avoid it even if it is harder to grasp or do at first.

Why simplicity matters in software development

People can juggle only a few things at a time, a small number. The same with the number of things one can think of at one time. That becomes even harder when things are intertwined, because one cannot reason about them in isolation.

Intertwining raises complexity combinatorially.

Why simplicity matters in software development p2

To be able to change existing software one needs to understand it and decide how and where to apply changes. If one cannot reason about a program, he can make those decisions.

Q: What happened to every bug out there? A: it passed the type checker, and it passed all tests.

The benefits of simplicity are: ease of understanding, ease of change, ease of debugging, flexibility.

What can we do to make things simpler and less intertwined?

Functional Programming

Functional JS Advantages

- Easy to reason about especially for pure functions
- Composition is simple
- Separation of concerns – do one thing well
- Testing is easy for stateless functions
- Treating data immutably reduces surprises and enables features like undo and auditing
- Event driven system like Redux creates predictable one directional data flow and makes it easy to react in multiple domains
- Frees code to be executed in an optimal way (time shift, concurrent processing)

Functional JS

```
// simple, testable, functional component
export const App = ({todos, actions}) => (
  <div>
    <Header addTodo={actions.addTodo} />
    <MainSection todos={todos} actions={actions} />
  </div>
)

// connect returns a HOC wrapped component
export default connect(
  state => ({       // maps redux state to props
    todos: state.todos
  }),
  dispatch => ({    // binds action creators to dispatch
    actions: bindActionCreators(TodoActions, dispatch)
  })
)(App);
```

react-redux connect

- locates just the data needed from redux store
- binds action creators to dispatch
- rerenders whenever our props change
- implements shouldComponentUpdate - optimized renders

```
// returns a HOC wrapped component
export default connect(
  state => ({       // maps redux state to props
    todos: state.todos
  }),
  dispatch => ({    // binds action creators to dispatch
    actions: bindActionCreators(TodoActions, dispatch)
  })
)(App);
```

redux early

- allows you to stay functional with your React components
  - maps the state for your app
  - state is easily segmented using combineReducers
  - one directional update
  - `connect()` - maps, binds, optimized renders
  - redux dev tools - awesome for debugging

Higher Order

- **Higher Order Functions** – function that takes a function and returns a function

```
const twice = (fn, x) => fn(fn(x));
const fn = x => x + 3;
const result = twice(fn, 7);  // (7+3)+3 = 13
```

- **Higher Order Components** (HOC) – function that takes a component and returns a component

```
const Profile = ({ name }) => <div>{ name }</div>;
const PureProfile = pure(Profile);
ReactDOM.render(<PureProfile name={myName} />, div);
```

Functional Composition

```
const getCurrentDate = () => new Date();
const format = date => date.toLocaleString();
const createTodayMessage = formattedDate => `Today is ${formattedDate}`;

const str = createTodayMessage(format(getCurrentDate()));

const createFormattedTodayMessage = compose(
  createTodayMessage,
  format
);

const str2 = createFormattedTodayMessage(getCurrentDate());
```

Stateless Function Components

```
function Orders({ orders }) {
  return (
    <ul>
      { orders.map(x => (
        <li key={ x.id }>{ x.name } - { x.date }</li>
      ))}
    </ul>
  );
}
```

Stateless function components

(ES6 arrow functions)

```
const Orders = ({ orders }) => (
  <ul>
    { orders.map(x => (
      <li key={x.id}>{ x.name } - { x.date }</li>
    ))}
  </ul>
);
```

Stateless Function Components

Advantages

- Extremely simple, easy to reason about
- Easy to test, no state, pass props to test modes
- Pure functions – input + output, nothing else
- No need to use “this”, props are local variables
  - Modular and reusable
- FaceBook team will continue to create optimizations. Favors functional style.

What do I do when I need to do things outside the scope of a stateless function component?

Form Helpers

- wrap your form with a HOC, provide initialState, submit handler
- helper/HOC provides value, onChange, onBlur handlers to your fields
- helper/HOC provides onSubmit and onReset to your form
- manages temporary form state
- validates fields, provides errors after touched
- validates all, marks as touched when form is submitted
- disables your submit button once submitting
- calls your submit handler with all values when form submits and passes validation

My favorite React form helpers

- formik - Build forms in React, without the tears
  - supports yup for easy declarative object validation
  - supports displaying individual field errors or combined
- react-final-form - High performance subscription-based form state management for React
  - uses observables to selectively update fields
  - mainly geared to display errors next to fields
- both are in active development with lots of contributors
- why not redux-form? - testing is more difficult when redux is required

Formik - React form helper

```
import { Formik } from 'formik';
const page = ({ initialValues, onSubmit }) => <div>
  <Formik initialValues={initialValues} onSubmit={onSubmit} >
    { ({values, errors, touched, handleChange, handleBlur, handleSubmit,
        handleReset, isSubmitting}) =>
      <form onSubmit={handleSubmit}>
        <input type="email" name="email" onChange={handleChange}
          onBlur={handleBlur} value={values.email} />
        { errors.email && touched.email &&
          <div className="error">{errors.email}</div> }

        <input name="first" onChange={handleChange}
          onBlur={handleBlur} value={values.first} />
        { errors.first && touched.first &&
          <div className="error">{errors.first}</div> }

        <button type="submit" disabled={isSubmitting}>Submit</button>
        <button onClick={handleReset}>Reset</button>
      </form>
    }
  </Formik>
</div>;
```

Formik - Form, Field helpers

```
import { Formik, Form, Field } from 'formik';
const page = ({ initialValues, onSubmit }) => <div>
  <Formik initialValues={initialValues} onSubmit={onSubmit} >
    { ({errors, touched, handleReset, isSubmitting}) =>
      <Form>
        <Field type="email" name="email" />
        { errors.email && touched.email &&
          <div className="error">{errors.email}</div> }

        <Field name="first" />
        { errors.first && touched.first &&
          <div className="error">{errors.first}</div> }

        <button type="submit" disabled={isSubmitting}>Submit</button>
        <button onClick={handleReset}>Reset</button>
      </Form>
    }
  </Formik>
</div>;
```

Formik - custom components

```
import Select from 'react-select';
import { Formik, Form } from 'formik';
const page = ({ initialValues, onSubmit }) => <div>
  <Formik initialValues={initialValues} onSubmit={onSubmit} >
    { ({values, setFieldValue, setFieldTouched, errors, touched, isSubmitting}) =>
      <Form>
        <Select name="category" value={values.category}
          onChange={ category => setFieldValue('category', category.value) }
          onBlur={ () => setFieldTouched('category', true) }
          options={[
            { value: 'one', label: 'One' },
            { value: 'two', label: 'Two' }
          ]}
        />
        { errors.category && touched.category &&
          <div className="error">{errors.category}</div> }

        <button type="submit" disabled={isSubmitting}>Submit</button>
      </Form>
    }
  </Formik>
</div>;
```

Formik - validate

```
import { Formik, Form, Field } from 'formik';
const validate = values => {
  const errors = {};
  if (!values.email) { errors.email = 'Email is Required'; }
  if (values.first && values.first.length > 30) {
    errors.first = 'First name can be a max of 30 chars';
  }
  return errors;
};
const page = ({ initial, onSubmit }) => <div>
  <Formik initialValues={initial} onSubmit={onSubmit} validate={validate} >
    { ({errors, touched, isSubmitting}) => <Form>
        <Field type="email" name="email" />
        { errors.email && touched.email &&
          <div className="error">{errors.email}</div> }
        <Field name="first" />
        { errors.first && touched.first &&
          <div className="error">{errors.first}</div> }
        <button type="submit" disabled={isSubmitting}>Submit</button>
      </Form> }
  </Formik></div>;
```

Formik - validationSchema Yup

```
import { Formik, Form, Field } from 'formik';
import Yup from 'yup';
const schema = Yup.object({
  email: Yup.string()
    .email('Invalid email address')
    .required('Email is required'),
  first: Yup.string().max(30, 'First name can be a max of 30 chars')
});
const page = ({ initial, onSubmit }) => <div>
  <Formik initialValues={initial} onSubmit={onSubmit} validationSchema={schema}>
    { ({errors, touched, isSubmitting}) =>
      <Form>
        <Field type="email" name="email" />
        { errors.email && touched.email &&
          <div className="error">{errors.email}</div> }
        <Field name="first" />
        { errors.first && touched.first &&
          <div className="error">{errors.first}</div> }
        <button type="submit" disabled={isSubmitting}>Submit</button>
      </Form>
    }
  </Formik></div>;
```

Formik - Resources / Demos

- Formik
  - https://github.com/jaredpalmer/formik
  - Basic form demo
  - Input primatives / Yup validation
  - other demos
- Yup
  - https://github.com/jquense/yup
  - Yup runkit

Recompose

- Utility library for creating HOC's
  - "The lodash for React"
- npm install recompose
- Compose in common functionality using parameterized HOC functions
- Allows you to stay in the functional world
  - Stateless function components

https://github.com/acdlite/recompose

Solving problems with Recompose

```
import { pure, onlyUpdateForKeys, defaultProps, renameProps } from 'recompose';

const PureProfile = pure(Profile);  // optimized

const OptimProfile = onlyUpdateForKeys(['name', 'age'])(Profile);

const DefaultedComp = defaultProps({
  greeting: 'Hello'
})(Comp);

const EntComp = renameProp('first', 'firstName')(Profile);
```

withProps, mapProps

```
import { withProps, mapProps } from 'recompose';

const Comp = withProps(props => ({
  fullName: `${props.first} ${props.last}`,
  mode: 1
}))(EchoProps);
ReactDOM.render(<Comp first="John" last="Smith" />, div);

const Comp = mapProps(props => ({
  firstName: props.first,
  lastName: props.last
}))(EchoProps);
ReactDOM.render(<Comp first="John" last="Smith" />, div);
```

composing

```
import { compose, pure, withProps } from 'recompose';

const Comp = compose(
  pure,
  withProps(props => ({
    fullName: `${props.first} ${props.last}`,
    mode: 1
  }))
)(EchoProps);

const div = document.querySelector('.dynamicName');

ReactDOM.render(<Comp first="John" last="Smith" />, div); // John Smith
ReactDOM.render(<Comp first="John" last="Smith" />, div); // no render, same
ReactDOM.render(<Comp first="Amanda" last="Green" />, div); // Amanda Green
```

example form

```
const Form1 = ({ values, onChange, onSubmit }) => {
  return (
    <form onSubmit={onSubmit}>
    <input name="first" value={values.first} onChange={onChange} />
    <input name="last" value={values.last} onChange={onChange} />
    <button>Submit</button>
    </form>
  );
};
```

composing to use with our form

```
import { set } from 'lodash/fp';
import { compose, withState, withHandlers } from 'recompose';

const Comp = compose(
  withState('values', 'updateValues', { first: '', last: ''}),
  withHandlers({
    onChange: ({ updateValues }) => ev => {
      const {name, value} = ev.target;
      updateValues(x => set(name, value, x));
    },
    onSubmit: ({ values, updateValues }) => ev => {
      ev.preventDefault();
      console.log('submit', values); // do something with data
      updateValues(x => ({ first: '', last: '' }));
    }
  })
)(Form1);
```

withStateHandlers

```
import { compose, withStateHandlers } from 'recompose';

const Comp = withStateHandlers(
  props => ({   // create initial state
    first: '',
    last: ''
  }),
  {  // state handlers
    onChange: props => ev => {
      const {name, value} = ev.target;
      return {  // return the state changes to merge
        [name]: value
      };
    },
    onSubmit: props => ev => {
      ev.preventDefault();
      console.log('submit', props)); // do something with data
      // return the state changes to merge
      return { first: '', last: '' };
    }
  }
)(Form1);
```

branch, renderComponent, renderNothing

```
import { branch, renderComponent, renderNothing } from 'recompose';
const Loading = props => <div>Loading...</div>;
const OurComp = ({ content }) => <div>Content: { content }</div>;

// if props.content is not truthy like when undefined
// render a Loading component instead of normal
const BranchedComp = branch(
  props => !props.content,  // predicate function, tests if we have content
  renderComponent(Loading)  // alt component to use if that was true
)(OurComp);  // otherwise fall through and render OurComp like normal

const div = document.querySelector('.app');
ReactDOM.render(<BranchedComp />, div);  // Loading...
ReactDOM.render(<BranchedComp content="hello" />, div);  // Content: hello
```

```
const BranchNothingComp = branch(
  props => !props.content,  // predicate function, tests if we have content
  renderNothing  // just renders nothing until we have content
)(OurComp);
ReactDOM.render(<BranchedNothingComp />, div);  // nothing to see here
ReactDOM.render(<BranchedNothingComp content="hello" />, div);  // Content: hello
```

composing branch, renderComponent, and lifecycle

```
import { compose, lifecycle, branch } from 'recompose';
const Loading = props => <div>Loading...</div>;
const MainContent = ({ items }) => (
  <ul>
   { items.map(x => <li key={x.id}>{x.name}</li> ) }
  </ul>
);
```

```
const DynamicContent = compose(
  // after mount, we will fetch and setState which passes down as props
  // so inner component will receive props containing 'items'
  lifecycle({
    componentDidMount() {
      axios.get('http://yourserver.com')
        .then(res => { this.setState({ items: res.data.result}); });
    }
  }),
  branch( // until we have 'items', show Loading component
    props => !props.items,
    renderComponent(Loading))
)(MainContent);
```

immutable data

- treat your data as immutable
  - prevents surprises
  - helps you reason about your code
  - problems due to mutations are hard to track down
- doesn't require immutable.js
  - map, filter, reduce rather than forEach
  - object spread or use helpers - lodash/fp, timm, updeep, ramda
- eslint plugins to help you prevent mutations
  - eslint-plugin-immutable - no-let, no-this, no-mutation
  - eslint-plugin-fp - additional fp style rules

lodash/fp

- already built-in to lodash
- alternate functional style interface to lodash commands
  - treats data immutably and curries
  - switches main data parameter last (in most commands) to enable composition
  - curries
- names are familiar to many so adoption is pretty easy
  - often already included in many projects
- needs better docs for fp usage (see jfmengels/lodash-fp-docs)

lodash/fp usage

```
npm install --save lodash
```

```
import fp from 'lodash/fp'; // get all lodash/fp commands as object
import get from 'lodash/fp/get'; // single method import
import { get, set } from 'lodash/fp'; // import named methods

// can use babel-plugin-lodash to transform into modularized imports
```

lodash/fp immutable helpers

```
import { set } from 'lodash/fp';

const state0 = {
  a: 1,
  b: {
    bb: [10, 20, 30]
  },
  c: {
    cc: {
      ccc: 'hello'
    }
  }
};

// dotted path or array of paths
const state1a = set('c.cc.ccc', 'hi', state0);
const state1b = set(['c', 'cc', 'ccc'], 'hi', state0)

// can also take advantage of currying
const changeCCCToHiFn = set('cc.ccc.ccc', 'hi');
const state1c = changeCCCToHiFn(state0);
```

lodash/fp p2

```
import {set, update} from 'lodash/fp';

// can also use array paths using [n]
const state2a = set('b.bb[1]', 200, state1a);
const state2b = set(['b', 'bb', 0], 200, state1a);

// setting a path that doesn't exist creates necessary objects
const state2 = set('d.dd.ddd', 'I am new', state1c);

// increment a by `
const state3 = update('a', x => x + 1, state2);

// append an item to b.bb
const state3 = update(
  'b.bb',
  arr => arr.concat(40),
  state2);
```

lodash/fp p3

```
import {compose, pipe, set, update} from 'lodash/fp';

// compose together updates
const state4 = compose(
  set('e.ee', 'Hey'),             // second
  update('b.bb[0]', x => x + 1) // first
)(state3);

// pipe allows you to compose from left to right
const state5 = pipe(
  set('f.ff', 'This happens first'),
  set('g.gg', 'This happens second')
)(state4);
```

lodash/fp p4

```
import {get, getOr} from 'lodash/fp';
const state0 = {
  b: {
    bb: [10, 20, 30]
  },
  c: {
    cc: { ccc: 'hello' }
  }
};
// use get with a path to select data in an object
// if any starting object or any path does not exist, it returns undefined
// (state0 && state0.c && state0.c.cc) ? state0.c.cc.ccc : undefined
const greeting = get('c.cc.ccc', state0); // use dotted path
const alsoGreeting = get(['c', 'cc', 'ccc'], state0); // or array path
const thirdValOfBB = get('b.bb[2]', state0);  // 30

// leave off the final param to create a selector function
const greetingSelector = get('c.cc.ccc');
const greeting = greetingSelector(state0);   // hello

// can use a default if not found/undefined
const fooDefaulted = getOr('mydefault', 'e.ee.eee', state0);
```

lodash/fp p5

```
import {get, pipe} from 'lodash/fp';
const greetingSelector = get('foo.bar.baz');
const appendYear = x => `${x} ${(new Date()).getFullYear()}`;
const wrapInQuotes = x => `"${x}"`;

// pipe (alias of flow) is like compose but
// invokes and passes outputs to fns from left to right
const quotedGreetingWithDateSelector = pipe(
  greetingSelector,
  appendYear,
  wrapInQuotes
);

const state = {
  foo: {
    bar: {
      baz: 'Hello JazzCon.tech'
    }
  }
};
const quotedGreet = quotedGreetingWithDateSelector(state);
// "Hello JazzCon.tech 2018"
```

devtools react/redux

- learn and embrace the tools
- easily learn what components are used and their props and state
- quickly learn about redux state changes and what is occuring in the system
- community continues to improve

Summary

- Functional JS is easy to understand, test, compose, and augment
- Use stateless function components and redux
- Treat your data immutably to prevent surprises
- Use form helpers like formik or react-final-form to make forms easy
- Yup object validation makes it easy to validate and cast objects
- Useful tools: recompose, lodash/fp

Thanks

- https://codewinds.com/jazzcon2018 (slides, resources)
- https://codewinds.com/ (newsletter tips/training)
- jeff@codewinds.com
- @jeffbski