Lessons from the trenches

Designing Bulletproof React/Redux Apps

JazzCon.tech 2018

Jeff Barczewski

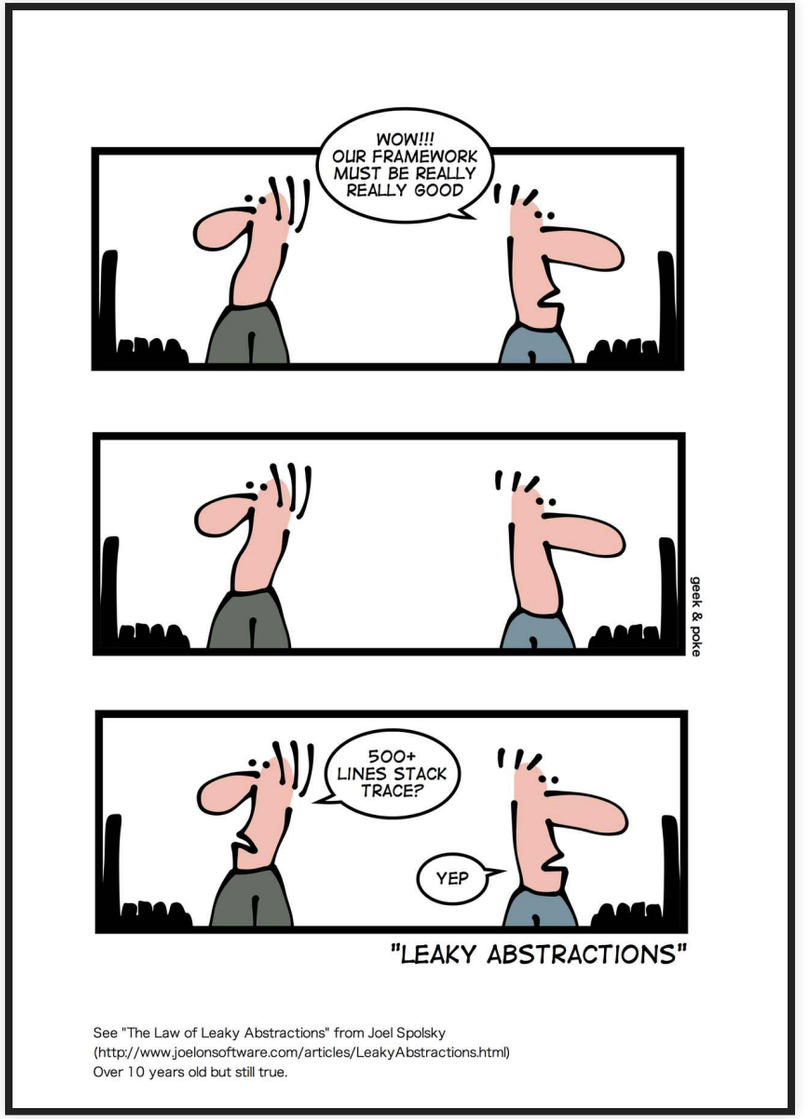
- @jeffbski
- jeff@codewinds.com
- https://codewinds.com/jazzcon2018

Jeff Barczewski

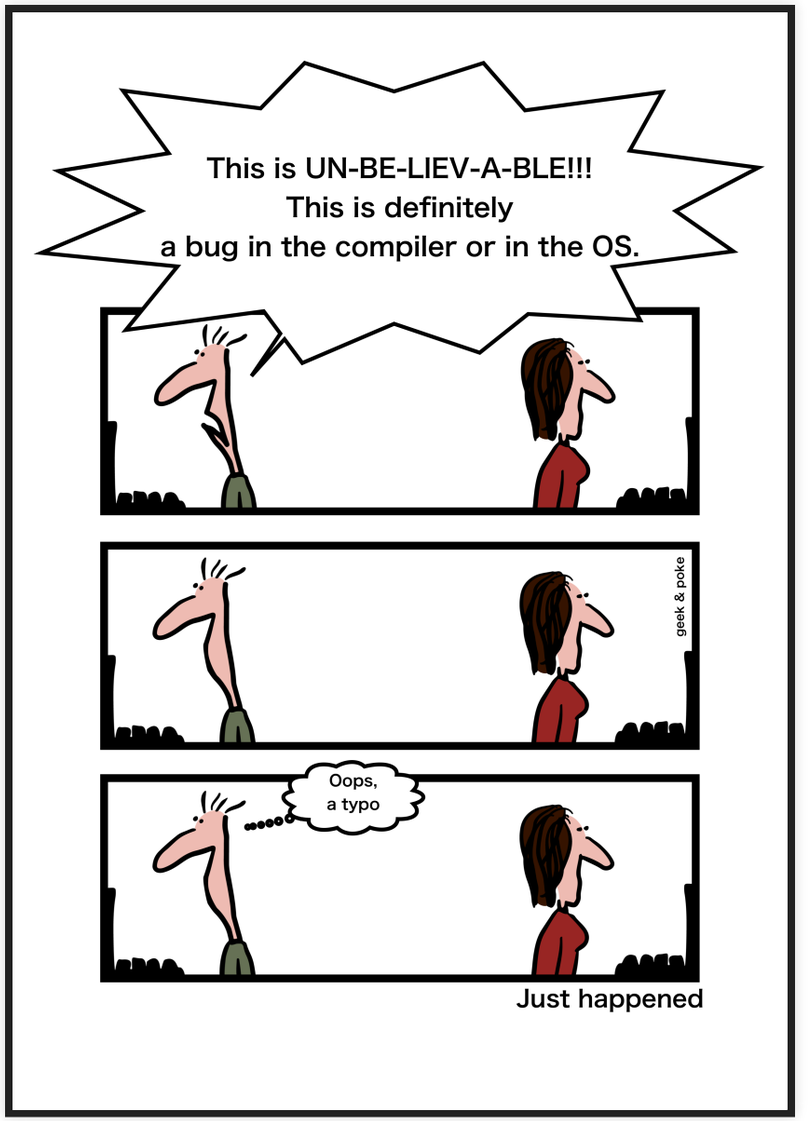
- Married, Father, Catholic
- 28 yrs (be nice to the old guy :-)
- JS (since 95, exclusive last 6 years)
- Open Source: redux-logic, pkglink, …
- Founded CodeWinds, live/online training (React, Redux, Immutable, RxJS) – I love teaching and mentoring, contact me

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



Good Framework by Geek & Poke Licensed CC BY 3.0



What inspires you?

F15 Eagle

F-15E Strike Eagle by Gerry Metzler - IMG_214 Licensed CC BY-SA 2.0

Afghanistan, F-15E 391st" by Staff Sgt. Aaron Allmon (USAF) - Src. Public domain

Life happens

F15 Single Wing Landing

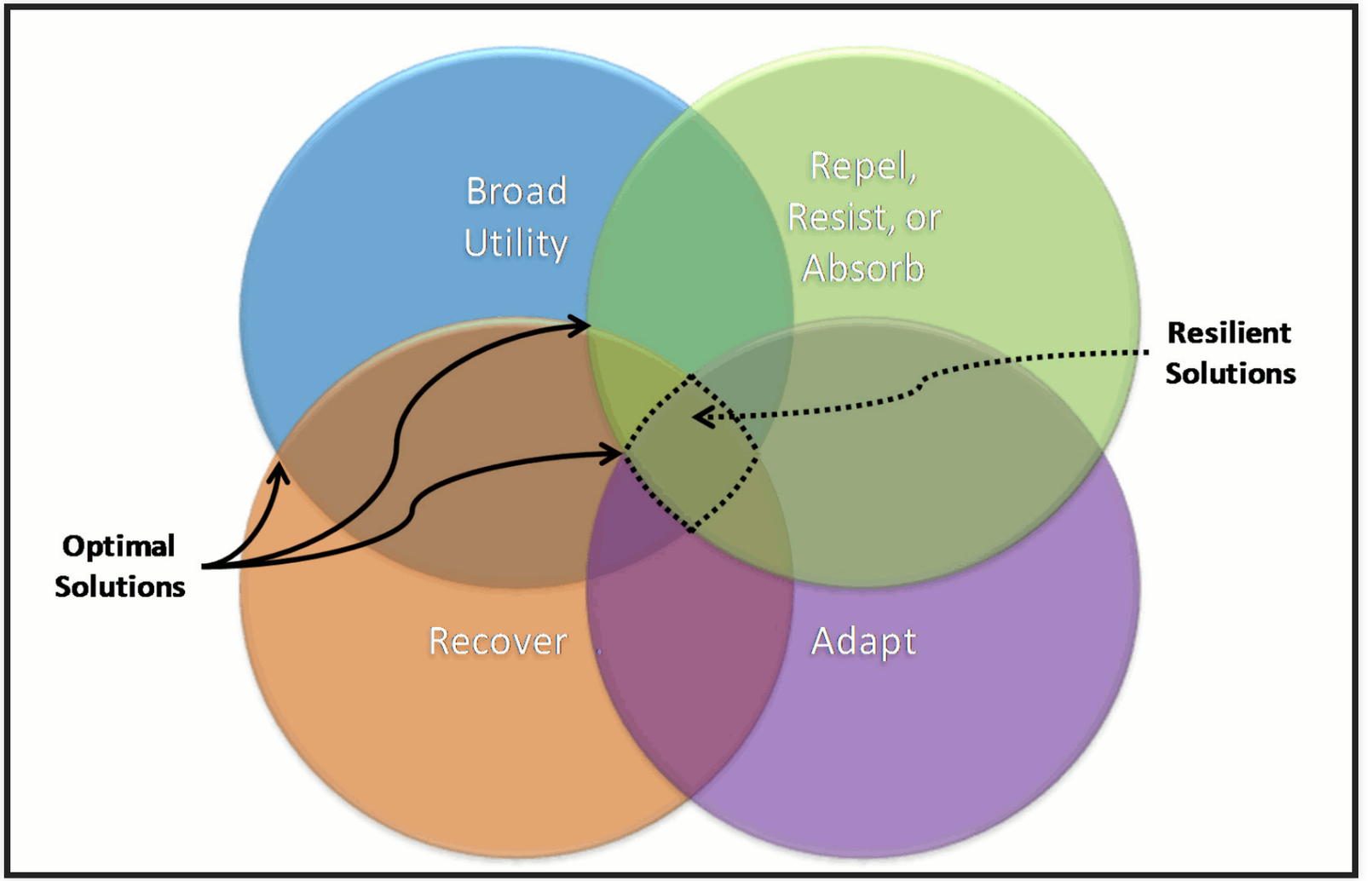
F-15 Single wing by Israeli Defense Force

F-15 Single wing landing by History Channel

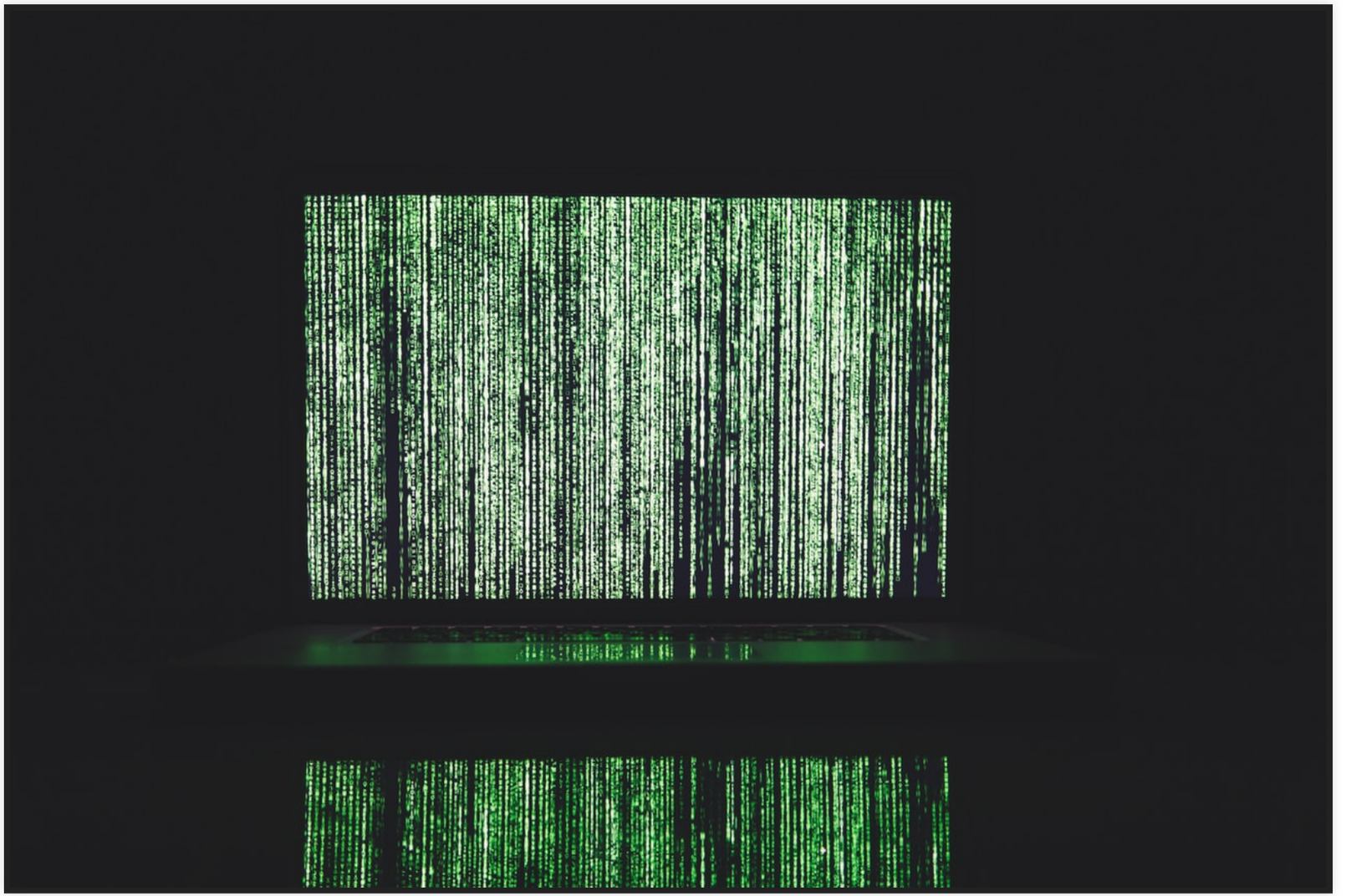
Traits that inspire me

- Performant
- Adaptability
- Durability
- Trusted

Resilient Systems



In general, how resilient is today's software?

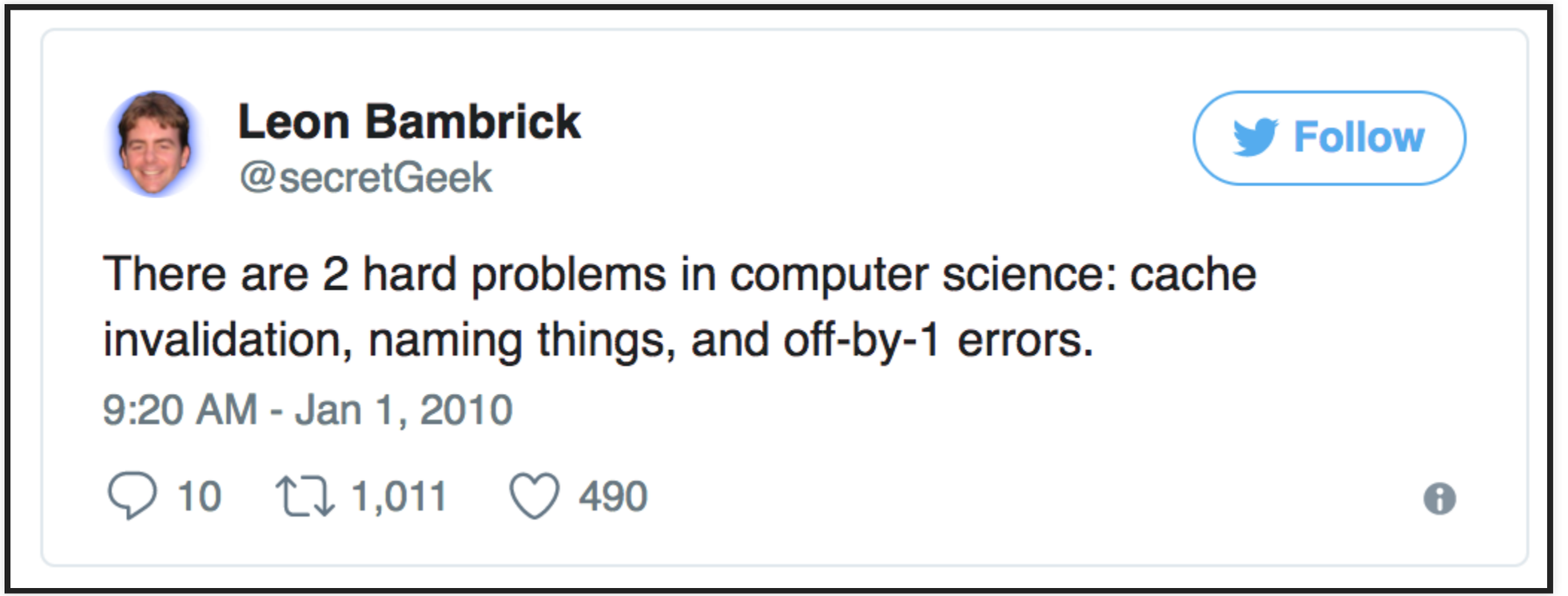


How do we build rock solid apps?

Designing Bulletproof Resilient React/Redux Apps

- Modularize your code
- Embrace a functional style
- Preventing surprises
- Business Logic
- Unit and Full end to end testing
- Tools and libraries that will help
- Use an intuitive code structure
- Planning for the journey

Programs are meant to be read by humans and only incidentally for computers to execute. - Donald Knuth


Leon Bambrick
@secretGeek
Follow
There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.
9:20 AM - Jan 1, 2010
10
1,011
490

There are two problems in computer science: there's only one joke, and it isn't funny. - Phil Karlton

Modularize

- Break large files into smaller manageable ones
  - Hard to do once it is cluttered
- Break large functions into smaller ones
  - Easy to reason about small pure functions
- Giving naming extra thought
  - Be consistent
  - Helps everyone find things
  - Refactoring is a pain

combine reducers (optionally nested) to separate state

```
// app/reducer.js
import product from '../product';
import user from '../user';

export default combineReducers({
  product: product.reducer,
  user: user.reducer
});
```

Use same structure for action types & state

```
// from product/actions.js
// Product actions are namespaced with product/
export const PRODUCT_ADD = 'product/ADD';
export const PRODUCT_DELETE = 'product/DELETE';

// from user/actions.js
export const USER_ADD = 'user/ADD';

// from app/reducer.js
import product from '../product';
import user from '../user';

export default combineReducers({
  product: product.reducer,
  user: user.reducer
});
```

redux-actions - AC / reducer helpers

```
// product/actions.js
import { createAction } from 'redux-actions';
// actionCreators that also return the action type when toString()'d
export const add = createAction('product/ADD');
export const remove = createAction('product/REMOVE');
export const foo = createAction(
  'product/FOO',
  (a, b) => ({ a, b }) // payload will be { a, b }
);
export const bar = createAction(
  'product/BAR',
  (a, b, c) => a, // payload will be a
  (a, b, c) => ({ b, c }) // meta will be { b, c }
);
const addAction = add({ id: 1, name: 'widget' });
// { type: 'product/ADD', payload: { id: 1, name: 'widget' }}
const removeAction = remove(1);
// { type: 'product/REMOVE, payload: 1 }
const fooAction = foo(10, 20);
// { type: 'product/FOO', payload: { a: 10, b: 20 }}
 const barAction = bar(1, 2, 3);
// { type: 'product/BAR', payload: 1, meta: { b: 2, c: 3 }}
```

redux-actions - (p2)

```
// product/actions.js
import { createActions } from 'redux-actions';

const actions = createActions({
  product: {
    ADD: null, // use identity fn for payload
    REMOVE: undefined, // use identity for payload
    FOO: (a, b) => ({ a, b }), // payload will be { a, b }
    BAR_BAZ: [
        (a, b, c) => a, // payload will be a
        (a, b, c) => ({ b, c }) // meta will be { b, c }
    ]
  }
});

// It will still convert the action object structure to camel case
const barAction = actions.product.barBaz(1, 2, 3);
// { type: 'product/BAR_BAZ', payload: 1, meta: { b: 2, c: 3 }}

export default actions.product; // exports { add, remove, foo, bar }
```

redux-actions - (p3)

```
// product/reducer.js
import { handleActions } from 'redux-actions';
import productActions from './actions';

const initialState = { a: null, b: null };

// create a reducer to handle all the actions
export default handleActions({
  [productActions.add]: (state, action) => {    // product/ADD
    // add reducer code
  },
  [productActions.remove]: (state, action) => { // product/REMOVE
    // remove reducer code
  },
  [productActions.foo]: (state, action) => {    // product/FOO
    // add reducer code
  },
  [productActions.bar]: (state, action) => { // product/BAR
    // remove reducer code
  }
}, initialState);
```

greppable code

Make it easy on yourself, so you can instantly find

- constants
- action types
- functions
- components

Selectors free our state structure

```
const state = {
 items: [
   { id: 1, name: 'Foo', catId: 20 },
   { id: 2, name: 'Bar', catId: 30 }
 ],
 categories: [
   { catId: 20, name: 'Games' },
   { catId: 30, name: 'Business' }
 ]
};
const itemsSelector = state => state.items;
const categoriesSelector = state => state.categories;
```

```
// using lodash/fp to create selectors
import {get} from 'lodash/fp';

const itemsSelector = get('items');
const categoriesSelector = get('categories');

// use dotted paths to go deep and [] to access arrays
const firstCategoryNameSelector = get('categories[0].name');
```

Combining (Naively)

```
const itemsWithCategoriesSelector = state => {
  const items = itemsSelector(state);
  const categories = categoriesSelector(state);
  const findCategory = catId => find(c => c.catId === catId)(categories);

  const itemsWithCategories = items.map(i => {
    const category = findCategory(i.catId);
    return compose(
      set('category', category),
      unset('catId')
    )(i);
  });

  return itemsWithCategories;
};
```

Reselect - memoized selectors

```
import { createSelector } from 'reselect';

const itemsWithCategoriesSelector = createSelector(
  /* simple light selectors */
  itemsSelector,
  categoriesSelector,
  /* complex expensive computation to memoize */
  (items, categories) => {
    const findCategory = catId => find(c => c.catId === catId)(categories);
    return items.map(i => {
      const category = findCategory(i.catId);
      return compose(
        set('category', category),
        unset('catId')
      )(i);
    });
  }
);
```

What does your fs structure tell you?

Typical react-redux-project/src

```
actions/
components/
containers/
reducers/
routes.js
```

What does your fs structure tell you? (p2)

Typical react-redux-project/src

```
actions/
components/
    Header.js
    Sidebar.js
    User.js
    UserProfile.js
    UserAvatar.js
containers/
reducers/
routes.js
```

What does your fs structure tell you? (p3)

Typical react-redux-project/src

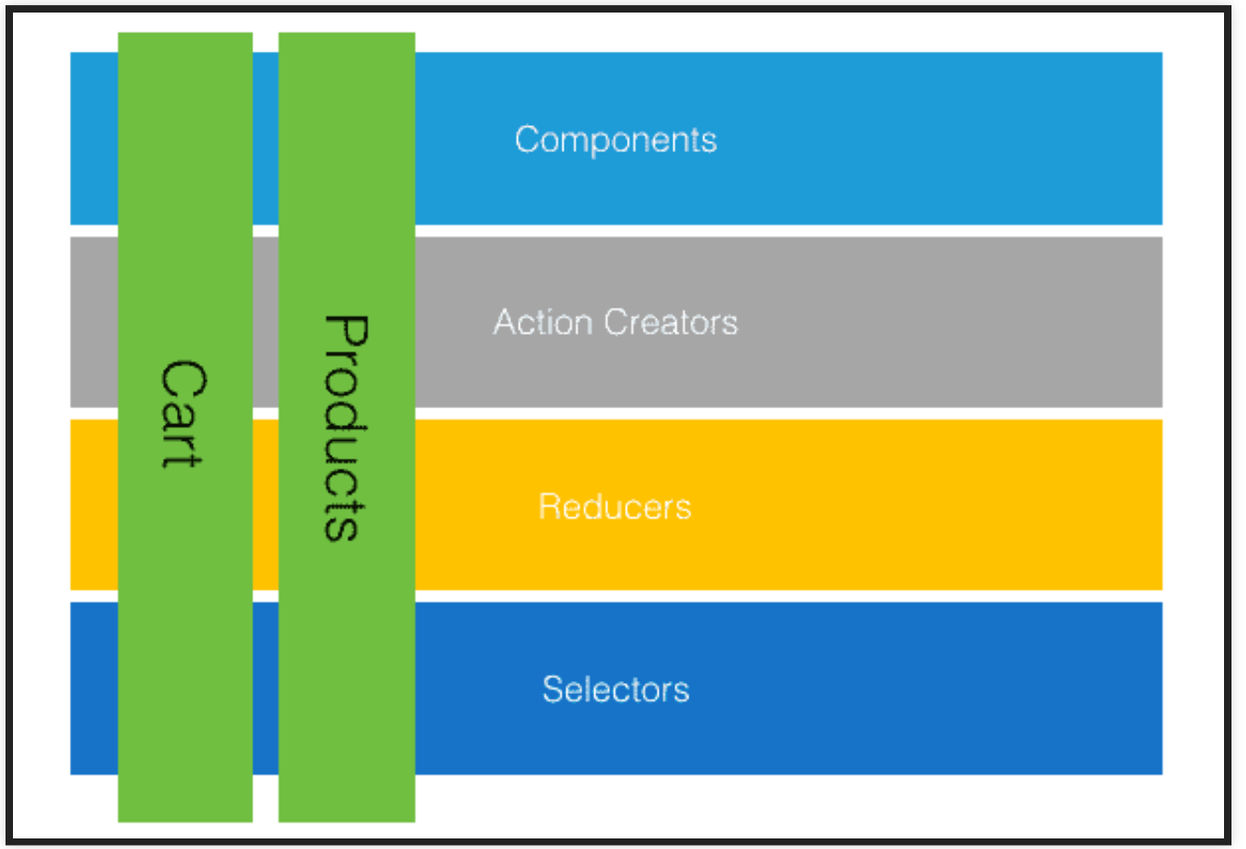
```
actions/
    UserActions.js
components/
    Header.js
    Sidebar.js
    User.js
    UserProfile.js
    UserAvatar.js
containers/
    App.js
    User.js
reducers/
    index.js
    user.js
routes.js
```

Adapted from @francoisz

Adding a feature

```
actions/
    ProductActions.js   <= Here
    UserActions.js
components/
    Header.js
    Sidebar.js
    Product.js          <= Here
    ProductList.js      <= Here
    ProductItem.js      <= Here
    ProductImage.js     <= Here
    User.js
    UserProfile.js
    UserAvatar.js
containers/
    App.js
    Product.js          <= Here
    User.js
reducers/
    index.js
    foo.js
    bar.js
    product.js          <= Here
routes.js
```

Features cross concerns



from Cristiano Rastelli's "Let There Be Peace on CSS" talk

Organize by feature or domain

```
app/
    Header.js
    Sidebar.js
    App.js
    index.js
    reducer.js
    routes.js
product/
    Product.js
    ProductContainer.js
    ProductList.js
    ProductItem.js
    ProductImage.js
    actions.js
    index.js
    reducer.js
user/
    User.js
    UserContainer.js
    UserProfile.js
    UserAvatar.js
    actions.js
    index.js
```

Tests live near the code

```
app/
    Header.js
    Header.test.js
    Sidebar.js
    Sidebar.test.js
    App.js
    App.test.js
    index.js
    reducer.js
    reducer.test.js
    routes.js
    routes.test.js
product/
    Product.js
    Product.test.js
    ProductContainer.js
    ProductContainer.test.js
    ProductList.js
    ProductList.test.js
    ProductItem.js
    ProductItem.test.js
    ProductImage.js
```

product/index.js

```
import * as actions from './actions';
import reducer from './reducer';
import Product from './Product';
import ProductList from './ProductList';
import ProductItem from './ProductItem';
import ProductImage from './ProductImage';

export default {
  actions,
  reducer,
  Product,
  ProductContainer,
  ProductList,
  ProductItem,
  ProductImage
}
```

feature-u

feature-u is a utility library that facilitates feature-based project organization in your react project. It assists in organizing your project by individual features.

https://feature-u.js.org/

feature-u revealed

- Concepts - https://feature-u.js.org/cur/concepts.html
- Benefits - https://feature-u.js.org/cur/benefits.html
- Usage - https://feature-u.js.org/cur/usage.html

feature-u example

- Basic expense tracking app
- Add expense amount and description
- Display List of expenses

feature-u example - bootstrapping

```
import ReactDOM           from 'react-dom';
import {launchApp}        from 'feature-u';
import {reducerAspect}    from 'feature-redux';
import {routeAspect}      from 'feature-router'; // basic state router

const features = [];

export default launchApp({

  aspects: [
    reducerAspect, // enables redux
    routeAspect,   // enables a simple state router
  ],

  features,

  registerRootAppElm(rootAppElm) {
    // where do we want to render this?
    ReactDOM.render(rootAppElm, document.getElementById('myAppRoot'));
  }
});
```

feature-u example - expenses feature

```
import React              from 'react';
import {createFeature}    from 'feature-u';
import {featureRoute}     from 'feature-router';
import {slicedReducer}    from 'feature-redux';
import expensesReducer    from './reducer'; // traditional reducer for expenses
import ExpenseList        from './list-ui'; // simple comp ({list}) => <ul>...</ul>
const KEY = 'expenses';

// define where I want this state mapped in redux, feature-redux ensures unique
const reducer = slicedReducer(`my.feature.${KEY}`, expensesReducer);
export const expenseListSelector = appState => reducer.getSlicedState(appState).list

export default createFeature({
  name:  KEY,
  route: featureRoute({
    content({app, appState}) { // render content if expenseList is defined
      const expenseList = expenseListSelector(appState);
      if (expenseList) return <ExpenseList list={expenseList} />;
    },
  }),
  reducer
});
```

feature-u example - dashboard feature

```
import React              from 'react';
import {createFeature}  from 'feature-u';
import {featureRoute, PRIORITY}   from 'feature-router';
import {slicedReducer}  from 'feature-redux';
import dashboardReducer  from './reducer'; // traditional reducer for expenses
import Dashboard      from './dashboard-ui'; // simple comp ({dashboard, expenses})
import expenselistSelector from '../expenses';
const KEY = 'dashboard';
const reducer = slicedReducer(`my.feature.${KEY}`, expensesReducer);
export const dashboardSelector = appState => reducer.getSlicedState(appState).list;

export default createFeature({
  name:  KEY,
  route: featureRoute({
    priority: PRIORITY.HIGH, // run this one the others
    content({app, appState}) { // render content if dashboard is defined
      const dashboard = dashboardSelector(appState);
      const expenses = expenselistSelector(appState);
      if (dashboard) return <Dashboard data={dashboard} expenses={expenses}  />;
    },
  }),
  reducer });
```

feature-u example - adding redux-logic

```
import ReactDOM           from 'react-dom';
import {launchApp}        from 'feature-u';
import {reducerAspect}    from 'feature-redux';
import {logicAspect}      from 'feature-redux-logic'; // <--- new import
import {routeAspect}      from 'feature-router'; // basic state router

const features = [];

export default launchApp({

  aspects: [
    reducerAspect, // enables redux
    logicAspect,    // <--- enables redux-logic, hooked into middleware
    routeAspect,   // enables a simple state router
  ],

  features,

  registerRootAppElm(rootAppElm) {
    // where do we want to render this?
    ReactDOM.render(rootAppElm, document.getElementById('myAppRoot'));
  }});
```

feature-u example - expenses feature w/redux-logic

```
import React              from 'react';
import {createFeature}    from 'feature-u';
import {featureRoute}     from 'feature-router';
import {slicedReducer}    from 'feature-redux';
import expensesReducer    from './reducer'; // traditional reducer for expenses
import ExpenseList        from './list-ui'; // simple comp ({list}) => <ul>...</ul>
import logic              from './logic';   // array of redux-logic items for expenses
const KEY = 'expenses';
const reducer = slicedReducer(`my.feature.${KEY}`, expensesReducer);
export const expenseListSelector = appState => reducer.getSlicedState(appState).list

export default createFeature({
  name:  KEY,
  route: featureRoute({
    content({app, appState}) { // render content if expenseList is defined
      const expenseList = expenseListSelector(appState);
      if (expenseList) return <ExpenseList list={expenseList} />;
    },
  }),
  reducer,
  logic  // <---- feature-redux-logic aspect added new property for logic
});
```

Where to learn more about feature-u

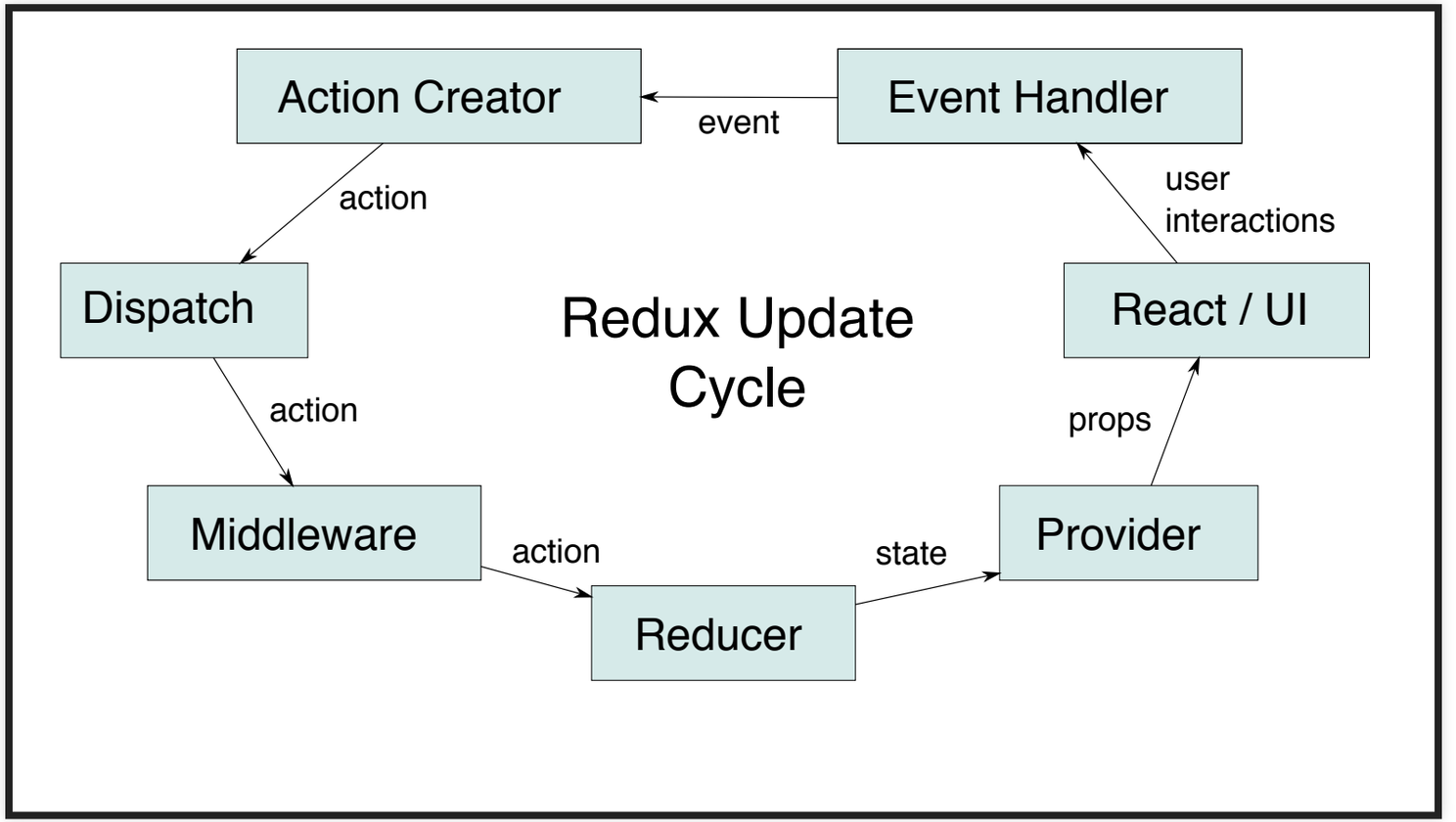
- https://feature-u.js.org/ - Home
- http://bit.ly/feature-u - Blog article introducing
- https://github.com/KevinAst - repos for feature-u, feature-redux, feature-redux-logic, feature-router
- https://github.com/KevinAst/eatery-nod - react-native (expo) example app using feature-u, redux, redux-logic
- More to come

Where can we implement business logic in react/redux?

Goals for our business logic

- Full state
- Dispatch
- Intercept (validate/transform)
- Async Processing
- Cancellation / Latest
- Filter / debounce / throttle
- Apply across many types
- One place for all logic
- Simple
- Load from split bundles

Where can we implement business logic in react/redux?



Common ways to implement business logic

- fat action creators
- reducers
- thunks
- sagas - redux-saga
- epics - redux-observable
- effects - redux-loop
- custom middleware
- redux-logic

My CodeWinds blog article discussing each method

My current preferences on business logic

- redux-logic - https://github.com/jeffbski/redux-logic
  - declarable
  - supports interception, async processing, cancellation
  - use promises, async/await, observables
- redux-observable or redux-most - second choice
  - requires more knowledge of observables or most's monadic streams
- custom middleware - powerful but DIY

redux-logic example

```
import { createLogic } from 'redux-logic';

const fetchPollsLogic = createLogic({
  // declarative built-in functionality wraps your code
  type: FETCH_POLLS, // only apply this logic to this type
  cancelType: CANCEL_FETCH_POLLS, // cancel on this type
  latest: true, // only take latest

  // your code here, hook into one or more of these execution
  // phases: validate, transform, and/or process
  process({ getState, action }, dispatch, done) {
    axios.get('https://survey.codewinds.com/polls')
      .then(resp => resp.data.polls)
      .then(polls => dispatch({ type: FETCH_POLLS_SUCCESS, payload: polls }))
      .catch(err => {
        console.error(err); // log since could be render err
        dispatch({ type: FETCH_POLLS_FAILED, payload: err, error: true })
      })
      .then(() => done()); // call done when finished dispatching
  }
});
```

redux-logic example 2

```
import { createLogic } from 'redux-logic';

const fetchPollsLogic = createLogic({
  // declarative built-in functionality wraps your code
  type: FETCH_POLLS, // only apply this logic to this type
  cancelType: CANCEL_FETCH_POLLS, // cancel on this type
  latest: true, // only take latest

  processOptions: {
    // provide action types or action creator functions to be used
    // with the resolved/rejected values from promise/observable returned
    successType: FETCH_POLLS_SUCCESS, // dispatch this success act type
    failType: FETCH_POLLS_FAILED, // dispatch this failed action type
  },

  // Omitting dispatch from the signature allows you to simply
  // return obj, promise, obs not needing to use dispatch directly
  process({ getState, action }) {
    return axios.get('https://survey.codewinds.com/polls')
      .then(resp => resp.data.polls);
  }
});
```

Testing - Unit / Integration Testing

- jest - https://facebook.github.io/jest/
- mocha - works with Node.js also - https://mochajs.org/
- expect style
  - built-in to jest
  - https://github.com/mjackson/expect
  - chai also has an expect style
- enzyme - http://airbnb.io/enzyme/
  - primarily using mount
  - css selectors

Testing - End to End Testing

- selenium web driver - https://webdriver.io and many others
- nightmarejs - http://www.nightmarejs.org/
  - electron based (selenium not required)
  - node.js code and API available
- headless chrome
  - exciting to go direct to the browser
  - Google supporting directly
  - I'm excited about this space!
  - https://github.com/GoogleChrome/puppeteer and many other libs

Testing - export testable

- make your private functions testable
  - export as testable
  - communicates privacy but still usable for testing

```
function foo(a, b) {
  // great private code here
}

function bar(c, d) {
  // more private code
}

const mainThing = { ... };
export default mainThing; // main function or object you are exporting

// exporting private functions for testing purposes
export const testable = {
  foo,
  bar
};
```

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);
```

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)

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

Stateless function components

```
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.

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);
```

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);
```

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 - 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

What does your README say?

RDD - Readme Driven Development

- Write the README first
- Start with description and goals
- Create example of using the API
- Share early to get feedback
- This is probably the most important doc to write

When do you create a PR?

Create your PR early

- Before you are writing the code
- Share the README, example, notes, or docs first
- Can use labels to indicate the stage of development
- Use the PR to discuss ideas, API, and implementation details before you even begin to code
- Even if you decide against building the feature, the PR stays around to capture the coversations that took place

npm scripts

Create npm run scripts for your common bash commands for the project.

- `npm run` will give a list of all available commands
- `npm start` - start up your developer auto build environment
- `npm test` - run your test suite
- Make it a goal to be able to just `npm install` for complete setup
- Add others as necessary
- pre/post scripts - `pretest` will run before test
- npm-run-all package provides `run-p` and `run-s` commands
  - `run-p watch:* foo bar` - starts all these in parallel
  - `run-s cat dog` - runs these sequentially

eslint - your early warning radar

- catch problems even before saving
- configure for your teams' style
- prevent the most common mistakes
- works in most editors or from command line

Reproduceable JS builds

- Lock down the versions of everything in your project
  - use package-lock.json - newer npm
  - or yarn.lock with yarn
  - or npm-shrinkwrap.json from older npm
- Can tar/zip project with node_modules
  - will need to rebuild if using different architecture
- Might even consider creating Debian packages

Reproduceable Environments

- For Mac, homebrew's `brew bundle Brewfile` - checks, installs, updates
- Ansible - only requires SSH on the server
- docker-compose
- other container/image tools

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

- Modularize and simplify your code
- Use combineReducers to separate state, selectors to find
- Structure your code by feature, try feature-u
- Use functional JS and treat data immutably using immutable helpers or object spread

- For business logic, try redux-logic, redux-observables, custom mw
- Tools: recompose, lodash/fp, formik, yup, redux-actions
- Use consistent structure for action types and state

Thanks

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