Lessons from the Trenches

Designing Resilient Bulletproof React/Redux Apps Part 1

Connect.tech 2017

Jeff Barczewski

- @jeffbski
- jeff@codewinds.com
- https://codewinds.com/connect2017

PHP CEO
@PHP_CEO

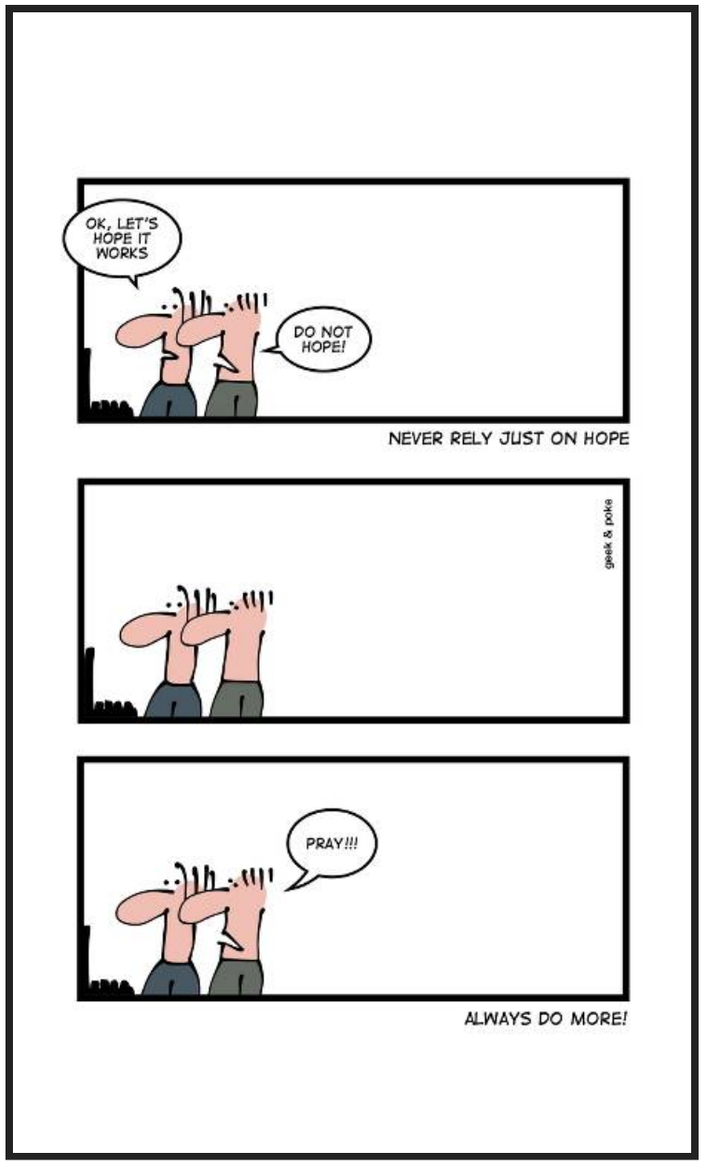
Follow

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



Jeff Barczewski

- Married, Father, Catholic
- 27 yrs (be nice to the old guy :-)
- JS (since 95, exclusive last 5 years)
- Open Source: redux-logic, pkglink, …
- Founded CodeWinds, live/online training (React, Redux, Immutable, RxJS) – I love teaching, contact me

CodeWinds Training

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

My early career

- Aerospace Engineer – US Air Force, ASD/XR Wright Patterson AFB, Ohio
  - Fortran 77 with extensions
  - C
  - C++ / Interviews
- Consulting
  - C++
  - Java

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

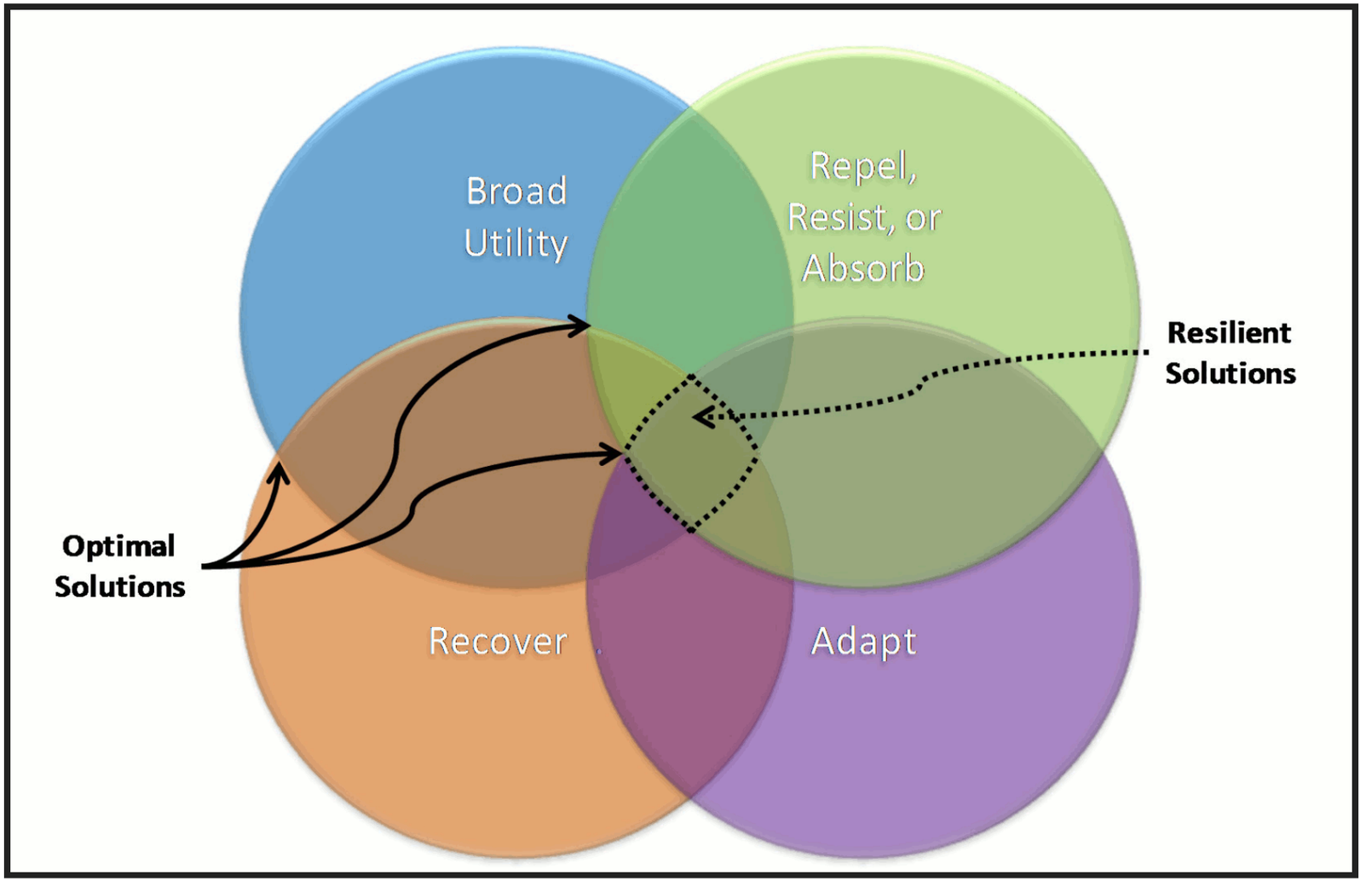
F-15 Single wing by Israeli Defense Force

F-15 Single wing landing by History Channel

Traits that inspire me

- Performant
- Adaptability
- Durability
- Trusted

Resilience


Broad
Utility
Repel,
Resist, or
Absorb
Resilient
Solutions
Optimal
Solutions
Recover
Adapt

Designing Resilient React/Redux Apps P1

- Use an intuitive code structure
- Modularize your code
- Embrace a functional style
- Preventing surprises
- Tools and libraries that will help
- Planning for the journey

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

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

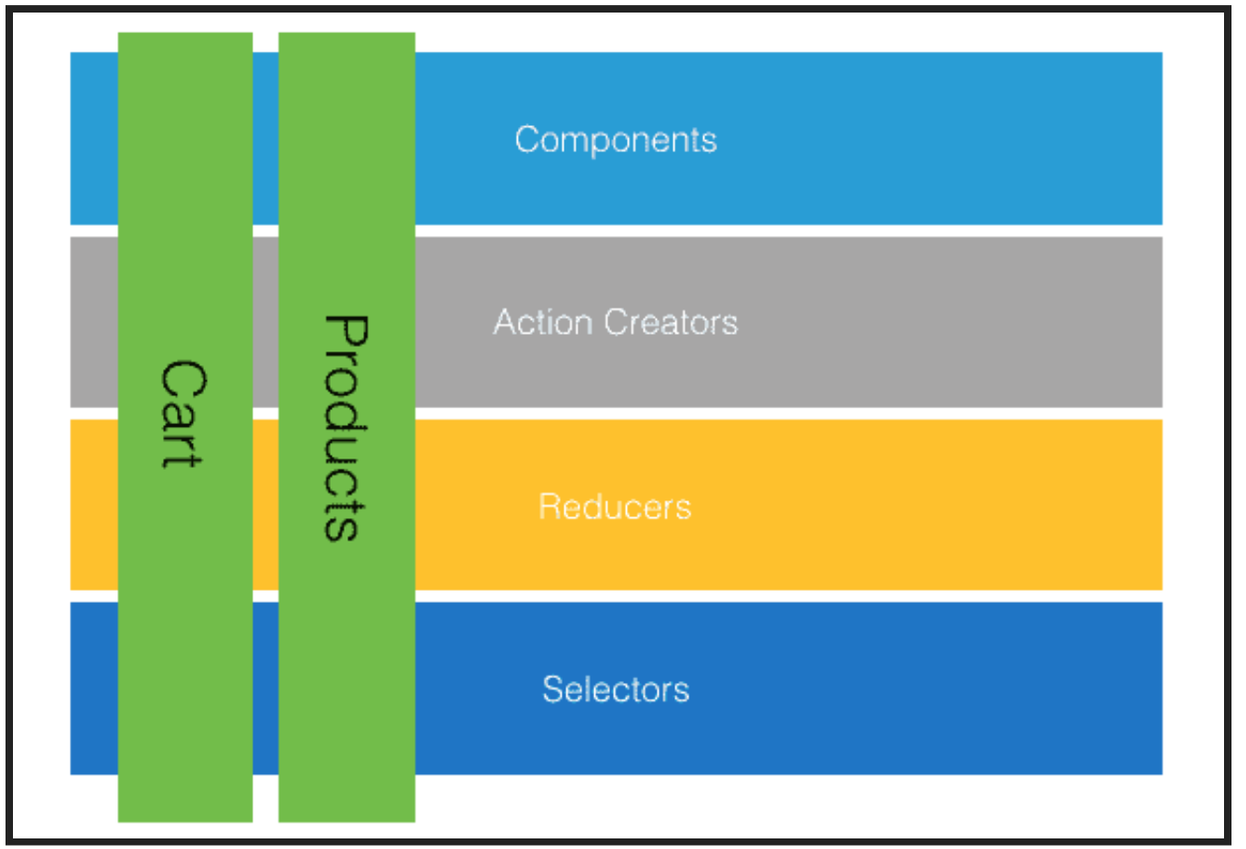
```
actions/
    UserActions.js
components/
    Header.js
    Sidebar.js
    User.js
    UserProfile.js
    UserAvatar.js
containers/
    App.js
    User.js
reducers/
    index.js
    user.js
routes.js
```

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
```

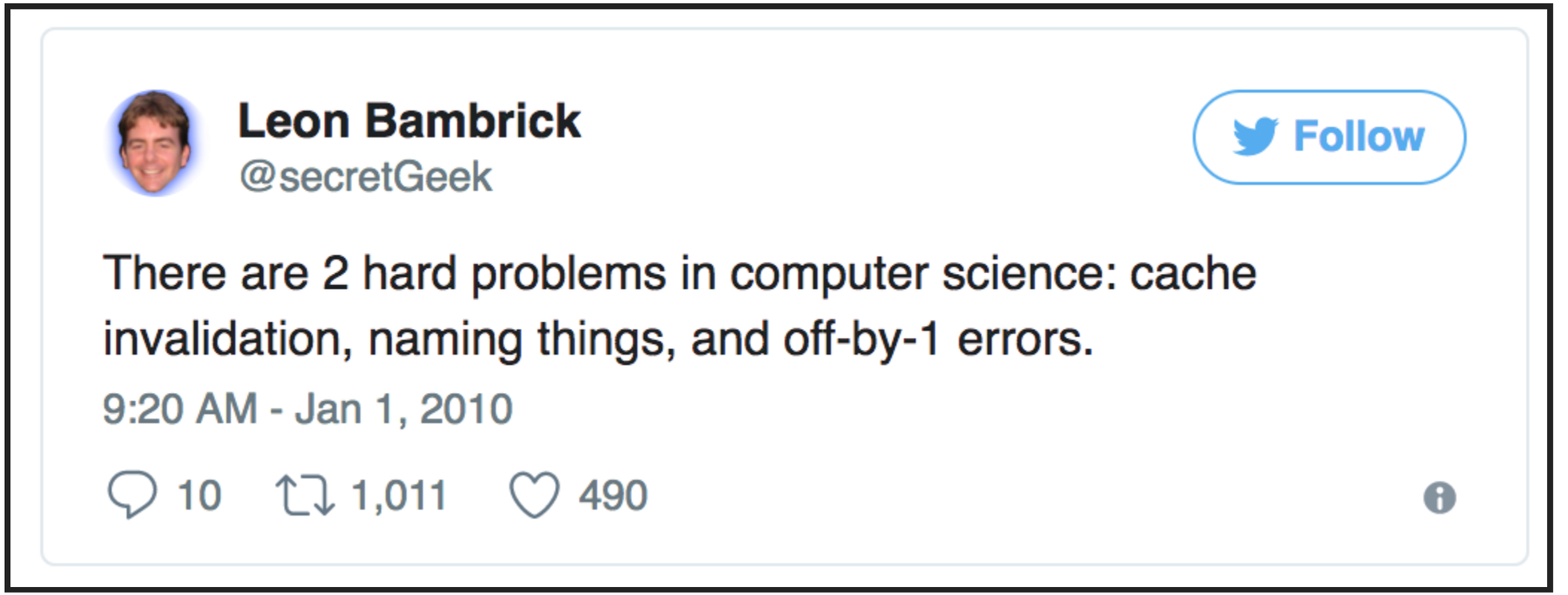
Tests live near the code

```
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
    ProductImage.test.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
}
```


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;
```

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 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

OO JavaScript

```
class App extends Component {
  constructor(props) {
    super(props)
    this.state = { todos: [] };
  }
  addTodo = (text) => {
    const todos = [
      {
        id: createUniqueID(),
        completed: false,
        text: text
      },
      ...this.state.todos
    ]
    this.setState({todos});
  }
  render() {
    return (
      <div>
        <Header addTodo={this.actions.addTodo} />
        <MainSection todos={this.state.todos} actions={this.actions} />
      </div>
```

OOP Advantages

- Methods live close to the data they work with
- Reuse through inheritance
- Encapsulation - protect data – changes performed by methods
- Flexibility through Polymorphism – obj.draw()

Object Oriented Programming

OOP Shortcomings

- Clutter – classes quickly grow in any real usage, everything is commingled - Hard to reason about
- Inheritance spreads implementation over many classes/files
- OOP favors mutation
- Cross domain sharing is difficult

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

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.

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

```
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

```
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

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

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';

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

```
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

```
const Loading = props => <div>Loading...</div>;
const Loaded = ({ content }) => <div>Loaded { content }</div>;

const Comp = branch(
  props => !props.content,
  renderComponent(Loading)
)(Loaded);

ReactDOM.render(<Comp />, div, () => {
  console.log(div.innerHTML);
  ReactDOM.render(<Comp content="hello" />, div, () => {
    console.log(div.innerHTML);

  });
});
```

composing branch, renderComponent, and lifecycle

```
const Loading = props => <div>Loading...</div>;
const MainContent = ({ items }) => <ul>
  { items.map(x => <li key={x.id}>{x.name}</li> )}
</ul>;

const DynamicContent = compose(
  lifecycle({
    componentDidMount() {
      axios.get('http://yourserver.com')
        .then(res => { this.setState({ items: res.data.result}); });
    }
  }),
  branch(props => !props.content, 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)

lodash/fp usage

```
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 { compose, get, getOr, set, pipe, update } 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

```
// 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);

// use get with a path to select data in an object
// if any starting object or any path does not exist, it returns undefined
const greeting = get('c.cc.ccc', state0);

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

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

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

lodash/fp p3

```
// 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);
```

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

- Design your apps to be resilient and maintainable so they will have a long useful life
- Structure, modularization, and naming matters
- RDD, PR early, npm scripts, eslint
- Functional JS is easy to understand, test, compose, and augment
- Use stateless function components and redux
- Treat your data immutably to prevent surprises
- Useful tools: redux-actions, reselect, recompose, lodash/fp

Thanks

- https://codewinds.com/connect2017 (slides, resources)
- https://codewinds.com/ (newsletter tips/training)
- jeff@codewinds.com
- @jeffbski