Contents

1 Welcome **1**

2 Me and Redux **2**

3 Ramda **3**

4 Redux **4**
4.0.1 Crash course in Redux 4

5 Redux Reducer **4**

6 Redux Mapper **5**

7 Redux Mapper 2 **6**

8 Refactoring a real app **6**

9 Bad map fix, step 1: create selectors **7**

10 Bad map fix, step 2: Composing with ramda **8**

11 Alternative Step 2: Reselect **9**

12 Alternative Step 2.5: Ramda Reselect **9**

13 Refactoring reducer **10**

14 Reducer refactor pt1. **11**

15 Reducer refactor pt2. **12**

16 Reducer refactor pt3. - Lenses! **12**

17 Reducer refactor pt4. - Transducers! **13**

18 Keys conclusion **14**

1 Welcome

- Kia ora koutou . I'm Brooke Mitchell and this is my talk, I've got the keys.

- Thank you to jenofdoom, the conference organizers, volunteers, and other speakers. I'm sad I'm missing anna's talk, her elm workshops are part of what got me in to functional programming.
- I've a developer who has been using JavaScript in earnest for maybe three years.
- These are my experiences using lenses and transducers to simplify data manipulation in a redux app I built.
- This talk is pretty code example heavy but try not to get too caught up in following every detail about what it happening. Just know that we're trying abstractions to manage objects and evaluating their usefulness. If you get interested you can try the examples from the slides in ramda repl later
- Since long code examples make me sleepy. There are helpful talk themed breaking them up. Try to focus on those.

2 Me and Redux

- So my last year I have been pretty heavily focused on react and redux. Don't worry this isn't a talk about react. This talk is a little about my experiences using redux. Really this talk is really about how to deal with pojos (plain old JavaScript objects) (A.K.A dictionaries, hashmaps) something we all use.
- I think dictionary manipulation can seems to be one of the less glamorous corners of functional programming; which is actually handy because it skips a lot of prior knowledge expected and you should be able to use any of these techniques in your day-to-day programming. One thing you will notice when using redux, the architecture tends to channel you towards functional approaches, I think is largely due to the inspiration takes from the ELM architecture.
- So since this talk is about my experiences using objects, this talk is about how we access properties in objects, through keys. They are the items that you provide to a hash function that unlock associated values. So and when we talk about keys, there is something slightly magical and metaphorical about them. For that reason I have been thinking about keys a lot, and so it made sense to consult the master of keys, the one and only, DJ Khaled.

. . . play jay-z - the keys clip. . . `https://www.youtube.com/watch?v=SFLSOIufuhM&feature=youtu.be&t=38s`

- For those unfamiliar with the work of DJ Khaled, you could briefly describe him as a music mogul and internet celebrity. Perhaps his most impressive skill is to act like a human hash function, and I mean that in a positive way. He can make a meaningful connection between any two people. Say you need Jay-Z and Snoop Dogg to catch a plane to your studio talk to DJ Khaled. So while Khaled is an important music producer, snapchat celebrity, motivational speaker but most importantly for this talk, he also someone who talks a lot about keys.

- Khaled describes his steps to success as 'keys' that open access to other keys, our path to success will also be like a deeply nested JSON response, (lame joke), so let's get started.

3 Ramda

- I'm going to use the Ramda library and es6 style functions in the examples in this talk. Just know that every any library should have these tools.

- I know we should be wary of tool dependence here, but ramda is a useful grab bag of mostly time honoured functional programming tricks that it is handy to be aware of. So I feel any time you spend learning it will be useful in other libraries and languages. Many other libraries do the same thing and they should be able to be used interchangeably.

- These are techniques you're going to see a lot in this talk. Most of the code it's only necessary to get the gist of, but if you want to pay attention now would be a good time.

  - So the amazingly useful functional features we'll be always using are:

- auto-currying . . . look at example

- function composition

- And applicatives. Perhaps the one people are least like to have seen before. I think of applicatives most of the time when I use them as way to take a list of functions and replacing the functions with their application. There are other many other ways to use their powers but that's all we require here.

4 Redux

4.0.1 Crash course in Redux

- Anyway back to the app I was building. Sadly it was built for a closed source environment. But here is a smaller toy example that also shows some of the problems that as redux apps get larger.

. . . Show app.

Before I show some of the refactorings I think are useful for dealing with objects, I'll try and offer as quick an overview of redux as possible. Please don't be mad if you are a redux expert.

- Here is the redux app flow from 10,000 feet:
- There is a 'store' that keeps all of your state in one object
- You can subscribe to store updates to map them to your view.
- You call store.dispatch(action)
- The Redux store calls any reducer functions (called 'reducers') that will make changes to state.
- The Redux store replaces the old state tree with a new state tree.

Hopefully my overview shows you that redux is very similar to mapping and reducing in vanilla js. That's pretty much it, talk over.

- So when building a redux app, apart from the boilerplate, you are essentially doing only two things, writing reducers to hanlde updates and mapping that state to your view. Thats it, reducing and mapping an object.

5 Redux Reducer

Coincidentally a 'reducer' in redux is effectively a 'reduce' for state, I like to think of it like so. . .

```
const initialState = {loading: true, filter: all}

['ACTION 1', 'STOP_LOADING', 'ACTION 3']
  .reduce((state, action) => {
    if (action.type === 'STOP_LOADING') {
      return Object.assign({}, state, {loading: false})
    }
    else return state
    }
  },  initialState)
```

In reality that list of actions is provided by redux's dispatch function and unfolds over time, more like an observer, but I think its a good way to conceptualize it.

Take special note of the line that looks like this:

```
return Object.assign({}, oldObject, newObject)
```

- Major key alert: don't mutate your state in a reducer. It will be ignored anyway in the diff comparison. What is mutation? Here are some links properly discussing mutation on my slides which you can grab on github. link. Anyway the opposite of mutability is immutability, what is immutability, basically it means keys and values of an object are unchangeable. In practice that means we need to return a fresh new copy of the object every time.

6 Redux Mapper

- ok now for the mapper.

- People typically use libs like react-redux, but lets try a redux mapping of state to a html output, this is pretty low-level and in reality react users tend to use higher-order-components like react-redux connect, but just to show you that you could happily use redux store in any scenario.

This is a state mapper in hyperscript, a handy way to generate dom nodes directly.

```
import h from 'hyperscript'
```

```
const widgetsList = widgets => h('div', h('ul', widgets.map(w => h('li', w))))
let divWithState = h('div', h('text', 'loading...'))
store.subscribe(() => {
  const currentState = store.getState()
  divWithState = widgetsList(currentState.widgets)
})
```

I think writing functions instead of jsx is pretty cool. JSX is cool too though.

State mappers are pretty much exactly the same for all libraries, I think of them like a .map where you plucks the desired items for an object.

7 Redux Mapper 2

This is a state mapper using nanocomponent. An very cool new component createion library that runs on any framework and performs well (using the same optimizations as react fiber). It should be compatable with all the frameworks and x-to-js compilers (even elm) and frees us from writing the same components like inifite list a zillion times.

```
const component = require('nanocomponent')
const html = require('bel')

const mapStateToProps = state => ({widgets: state.widgets})
const props = mapStateToProps(store.getState())

var WidgetList = component({
  render: function (props) {
    return html`
      <ul>${props.widgets.map(e => html`<li>${e}</li>`)}</ul>
    `
  }
})
```

Anyway dispatch, reduce, map. That is my summary of redux, so enough of that.

8 Refactoring a real app

- Hopefully you can now see that setting state is the same as applying a reducing function, and getting state is similar to a mapping over state and plucking entries you care about out.

- And here is a troubled mapper. In my toy example as is often the case things weren't looking real world enough, so I decided to implement some feature creep, user management and routing. Again you dont need to read this, just get a bad feeling that all this logic shouldn't really be in a view.

```
const mapStateToProps = state => {
  return {
    user: state.users[state.routeParams.uid]
    userDetails: state.usersDetails[user.uid] userDetails,
    noUser: typeof user === 'undefined',
    name: noUser ? '' : user.info.name,
    lastUpdatedUser: user ? user.lastUpdated : 0,
    isFetching: user.isFetching || usersDetails.isFetching,
    error: users.error || usersDetails.error,
    ...
  };
};
```

- For a quick glance this looks like way too much business logic to have in a view. Also all this nested parameter access is sure to cause runtime errors if a property isn't available at a certain point in time. Key alert: use ramda/lodash 'get' instead.

9 Bad map fix, step 1: create selectors

- The first step taken is usually to get this property access out of the view and somewhere else. Usually I just make a selector file and work from there, it helps with testing, and we remove the any logic or intermediary functions from the view.

```
// selectors.js
const user = state => state.users[state.routeParams.uid]
const noUser = state =>  typeof user(state) === 'undefined'
const works$ =  state => state.works.works
const editing = state => state.works.editing
const editing$ =  R.compose(R.propOr([], 0),
                                  R.toPairs,
                                             editing)

//container.js
export const mapStateTo = (state) => {
  return {
    name: name$(state),
    userDetails : userDetails$(state),
    error: error$(state),
    editing: editing$(state)
  };
};
```

- We could go further but good enough I say, at least these are easily composable and testable now. We could take this even further and create an uber selector that combines all the selectors.

10 Bad map fix, step 2: Composing with ramda

Major key: compose selectors.

```
// selectors.js
export const stateToProps$ = R.compose(
  R.zipObj(['editing', 'works']),
  // or R.memoize(R.zipObj(['editing', 'works'])),
  R.ap([
    name$,
    userDetails$,
    error$,
    editing$
    ]),
  R.of,
)

//container.js
const mapStateToProps = stateToProps$(store.getState())
```

- Wayyy sweeter. tbh this is probably the sweet spot. Go deeper if needs require.

11 Alternative Step 2: Reselect

_ The alternative route to composing selectors is to use a selector library like reselect.

- You get the same ability to compose selectors in a library. You also get the win of createSelector memoizing the results for you. This means that if anytime the result from all the selectors is the same, createSelector doesn't bother calculating the state again, this should save a few cpu cycles but in my experience often doesn't offer major speed ups, not like a virtualdom-diff for example, but it's still nice not to create a new object every time.

```
import { createSelector } from 'reselect'
const isFetching = createSelector(
  [ user, userDetails ],
  (user, userDetails) => user.isFetching || usersDetails.isFetching,
)

export const stateToProps$ = createSelector(
  [name$, userDetails$, error$, editing$],
  (name, userDetails, error, editing) => ({name, userDetails, error, editing})
)
```

12 Alternative Step 2.5: Ramda Reselect

- My issue with reselect is it re-invents the wheel a bit when you could just take the time to learn composition and not sweat the difference when frameworks change.

- Ok so this is my version, and its very close, and probably good enough for most scenarios.

- We could just as easily use composition to create similar functionality and keep a lot of flexibility. In fact here is the same functionality as what I need from reselect, selector composition, in a ramda one-ish liner using applicatives. Try not to read this too carefully and just get a feeling that we've just composed together our own reselect library from existing pieces.

```
const createSelector = (...fns) =>
  R.compose(
    , R.apply(R.memoize(R.last(fns))))
    , R.ap(R.slice(0, -1, fns))
    R.of
```

- This does the same thing as createSelector, takes the state, runs it through a list of selectors (except the last one) then applies those values to the last function, which has been memoized.
- Now we also get memoize and we dont have to learn another library. There are other capabilities reselect has which I've never used. Like props, you'll notice I pretty much never use props.
- Thats another key I've found, focus on state for stateful components and just use props with pure components to keep things simple. Things don't always work out that way but I find that really helps keep things simple.
- I have a more fully featured version of ramda-reselect that lets you use props. It passes all vanilla reselects tests in case you ever want to use it, or hopefully just look at the source, it bumps the lines up to about 10. npm.com/ernusame/ramda-reselect
- So I feel like we've slimmed down our mapper pretty nicely.

Now lets take a look at our reducer.

13 Refactoring reducer

- Here is the real reducer for the roadworks editing app, this is the function for setting the new shape of the state called every time an 'action' is dispatched.

```
export default function works(state = initialState, action) {
  switch (action.type) {
    case WORKS_FETCH_FAILED: {
      return {
        ...state,
        appState: "error",
        error: action.message
      };
    }
    case SET_TEXT: {
      const oldItem = state.works[action.changedEntry.id];
      const newItem = action.changedEntry[action.changedEntry.id];

      const mergedEntry = {
        works: {
          ...state.works,
          [action.changedEntry.id]: {
            ...oldItem,
            ...newItem
          }
        }
      };

      return {
        ...state,
        ...mergedEntry
      };
    }
    default:
      return state;
  }
}
```

14 Reducer refactor pt1.

- I think this is a little much for one function. The advice from redux is to break functions out, and I think you can easily see how to do that. To me breaking out functions feels a little dishonest. It makes things easier to read but doesn't actually reduce complexity, now you just look in a different place, making reasoning easier is really what we want.

```
function setText (state, action) {...}

export default function works(state = initialState, action) {
    ...
    case SET_TEXT: {
      setText(state, action)
    }
    ...
}
```

15 Reducer refactor pt2.

- How about trying something else, an abstraction that allows you to target a specific part of a deeply nested object, then returns the entire object. How about something else that seems obvious but I never see. What if we could use our selectors in a reducer. This won't work. But I'm getting a feeling there is an abstraction for focusing on a section of an object for a wide range of operations.

```
export const editTextReducer = createSelector(
  state, editing$,
  (state, action) => Object.assign({}, state, {editing})
)

export default function works(state = initialState, action) {
    case SET_TEXT:
      return editTextReducer(state, action)
      };
    }
}
```

I'm talking about...

16 Reducer refactor pt3. - Lenses!

- Now we can use the lens in both places!

```
// selector
const worksItemLens = R.lensPath(["works", id, key]);
// reducer
function works(state = initialState, action) {
    case SET_TEXT: {
      const { id, key, value } = action
      const worksItemLens = R.lensPath(["works", id, key]);
      return R.set(worksItemLens, value, state);
    }
}
```

- Here is a rewrite of the SET$_{\text{TEXT}}$ actions reducing case.

```
// reducer
    case SET_TEXT: {
      const { id, key, value } = action
      const worksItemLens = R.lensPath(["works", id, key]);
      return R.set(worksItemLens, value, state);
    }
```

To me this is way cleaner. And get ready for the second win, your lenses act as both getters and setters, so you get selectors for free when you write them. Major key. I stop thinking in terms of reducers now and just think of writing a selector as a lens when I need something, compose lenses together for my mapStateToProps, and later I can use it to set the change I've dispatched in the reducer.

One of the fun things about lenses is they look like they compose left to right. It's a little confusing

```
// component container
const prefixWorks = e => R.compose(R.lensProp('works'), e)
const allViews = R.map(
  R.compose(R.view, prefixWorks),
  [errorLens, editingLens, worksLens])

const mapStateLensToProps = R.compose(
  R.zipObj(['error', 'editing', 'works']),
  R.ap(allViews),
  R.of,
)
```

- Damn and its faster. This is pretty nice to look at, although we have to be aware of the tradeoffs. Lenses don't compose as well and require a context switch in thinking. I'm not sure it's often worth it.

17 Reducer refactor pt4. - Transducers!

- One last thing to try

- Ok the title of this talk promised that there would be transducers as well. But as I was working with tranducers I was finding that they weren't quite right for my use case and I ended up reverting a large chunk of the code base and focusing on composition instead.

If you have code that is performing a large number of transformations on data. You can make good performance gains by using transducers. Transducers generalize the reducing function, passed to a reducer, so that transformations can be composed. It's also data structure agnostic. Swapping reducer functions for transducers is definitely an interesting area but I feel like I'm running out of time anyway.' Check out transducers.js or ramdas transducer function. It is excellent for situations like this

```
const t = require("transducers.js")

const xform =  t.compose(
        t.map(function(kv){return [kv[0], kv[1] + 10 ]}),
        t.map(function(kv){return [kv[0], kv[1] * 9]}),
        t.filter(function(kv){return kv[1] % 2 !== 0; }),
       )

t.seq({ one: 1, two: 2, three: 3 }, xform);
// => {one: 99, three: 117}
```

These could be super useful for something like a complex text filter. But in my case we dont have a that need yet. I started refactoring all our reducers -> transducers in the original codebase but it quickly became apparent it wasn't going to be great for ongoing maintenance.

18 Keys conclusion

- So by 'keys' I mean two things, the first, more obviously, is the meaning of keys items to access values in an object. Secondly I mean keys in DJ Khaled's sense, as ways to open doors to further success. These are some of the more abstract pieces of advice based on things that I have learned, and based on Dj Khalid's key taxonomy, can be roughly divided into minor and major keys. Minor keys are often called 'tips' and concrete examples of things you can do to improve experience, things like, 'enable redux developer tools' or 'always surround yourself with pillows' that are practical tips based that can be directly copied to similar scenarios.

- I've tried not to say that a technology or a technique is a key, I dont think lenses are better than mapping for example. Lenses have a lot of pitfalls and can be way overkill. But the approach that they represent, of simplifying by looking for similarity between mapping and reducing to refactor code is the key that I like.
    - Major Keys, tend to be concepts that are higher up on the levels of abstraction. Link Cheng Lou on the spectrum of abstraction, very important talk, (power = access to other tree levels) these have more universal principles but are are harder to describe in concrete terms, they are often highly metaphorical or strange sounding in terms of domain specific language. for example 'secure the bag' or 'keep two kitchens running'. They may sound nonsensical without the appropriate knowledge but these kinds of advice are similar to old sayings, like 'yagni' in programming and take extra effort to apply to a concrete case but have a larger amount of potential.
    - Alan Kay - Build things with knowledge and technique. Using a higher level form can remove loc in orders of magnitude, depending on choice of abstraction.
- I wish you the best of luck and may you all keep winning.

`https://www.youtube.com/watch?v=GGXzlRoNtHU`