Redux Boilerplate

A brownbag deep-dive at

MX®

by Seth House

State management?

State management?

- 🌐 Global state atom,
  👱 Composable individual state atoms.

State management?

- 🌐 Global state atom,
  👱 Composable individual state atoms.
- 📦 Encapsulated state,
  🌿 Freely accessible state.

State management?

- 🌐 Global state atom,
  👱 Composable individual state atoms.
- 📦 Encapsulated state,
  🍃 Freely accessible state.
- 🔃 Data lifecycles.

State management?

- 🌐 Global state atom,
  👱 Composable individual state atoms.
- 📦 Encapsulated state,
  🌿 Freely accessible state.
- 🔃 Data lifecycles.
- 👀 Aware of downstream consumers.

State management?

- 🌐 Global state atom,
  👱 Composable individual state atoms.
- 📦 Encapsulated state,
  🌿 Freely accessible state.
- 🔃 Data lifecycles.
- 👀 Aware of downstream consumers.
- 📐 Principled access.

State management?

- 🌐 Global state atom,
  👱 Composable individual state atoms.
- 📦 Encapsulated state,
  🌿 Freely accessible state.
- 🔃 Data lifecycles.
- 👀 Aware of downstream consumers.
- 📐 Principled access.

Mgmt	🌐 👱	📦 🌿	🔃	👀	📐
Component	👱	📦	✅	✅	⛔ *
Redux	🌐	🌿	⛔	⛔	✅
Context	👱 *	🌿	⛔	⛔ *	⛔

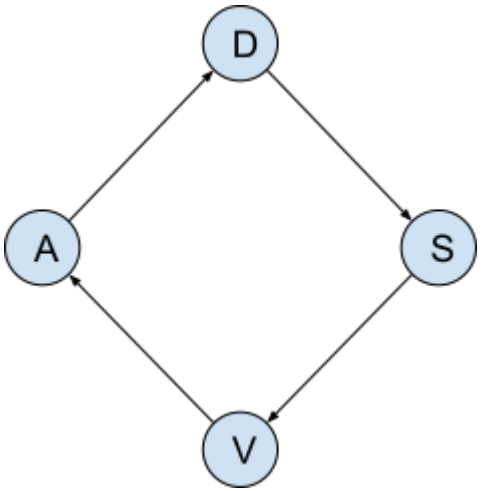
The Flux pattern

User interface architectures

- MVC
- MVP
- MVT
- MVVM
- MVI
- Flux
- BEST
- MVU

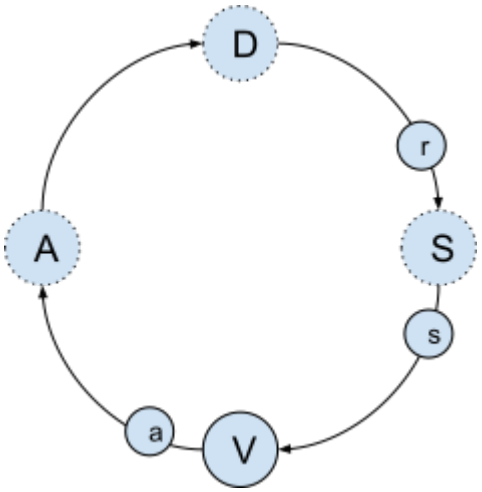
...etc.

Flux



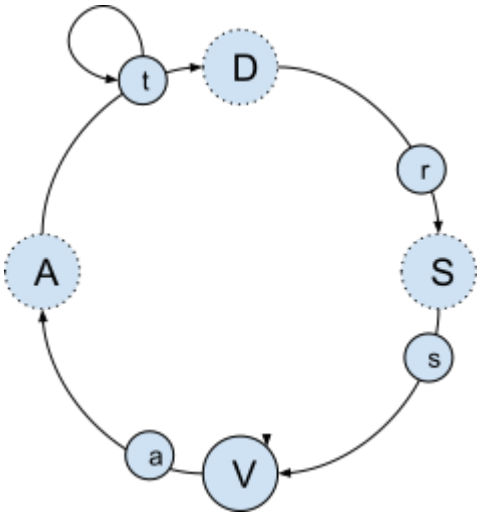
D - Dispatcher
S - Store
V - View
A - Actions

Redux



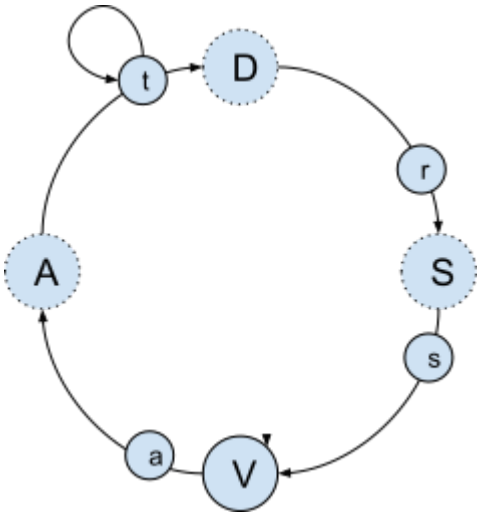
D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators

redux-thunk



D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators
t - thunks

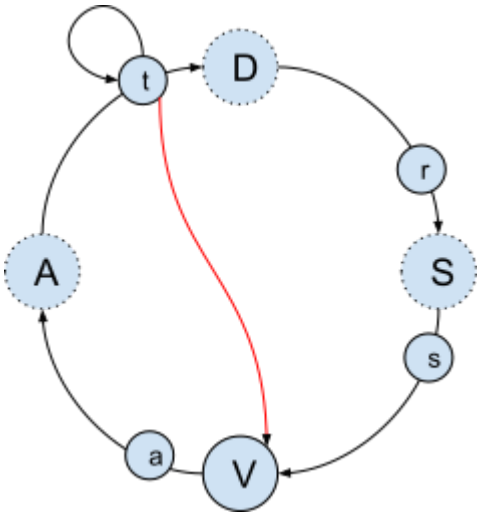
redux-thunk



D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators
t - thunks

```
const myAction = (payload) => (dispatch) =>
    MyAPI.someCall(payload).then(rep =>
        dispatch({
            type: ActionTypes.SOME_ACTION,
            payload: rep.body,
        }))
```

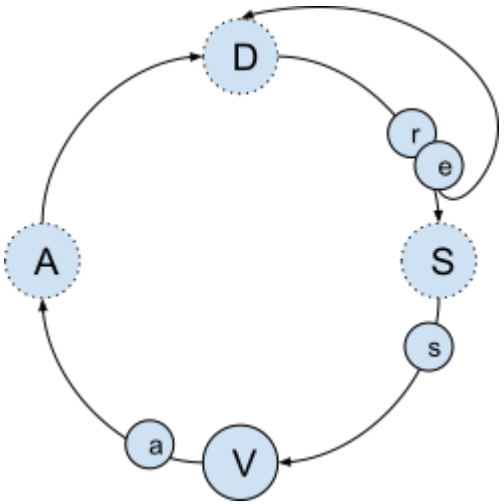
redux-thunk (warning)



D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators
t - thunks

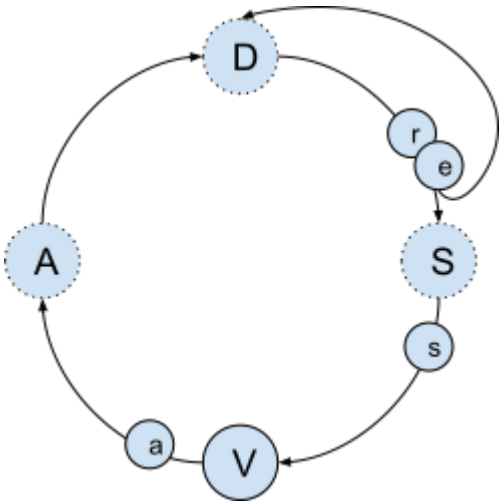
```
// Don't return a Promise from an action creator.
const myAction = (payload) => (dispatch) => {
    MyAPI.someCall(payload).then(rep =>
        dispatch({
            type: ActionTypes.SOME_ACTION,
            payload: rep.body,
        }))
}
```

redux-observable



D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators
e - epics

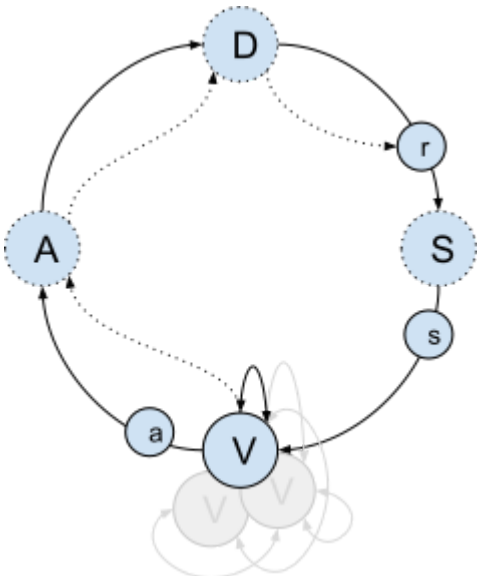
redux-observable



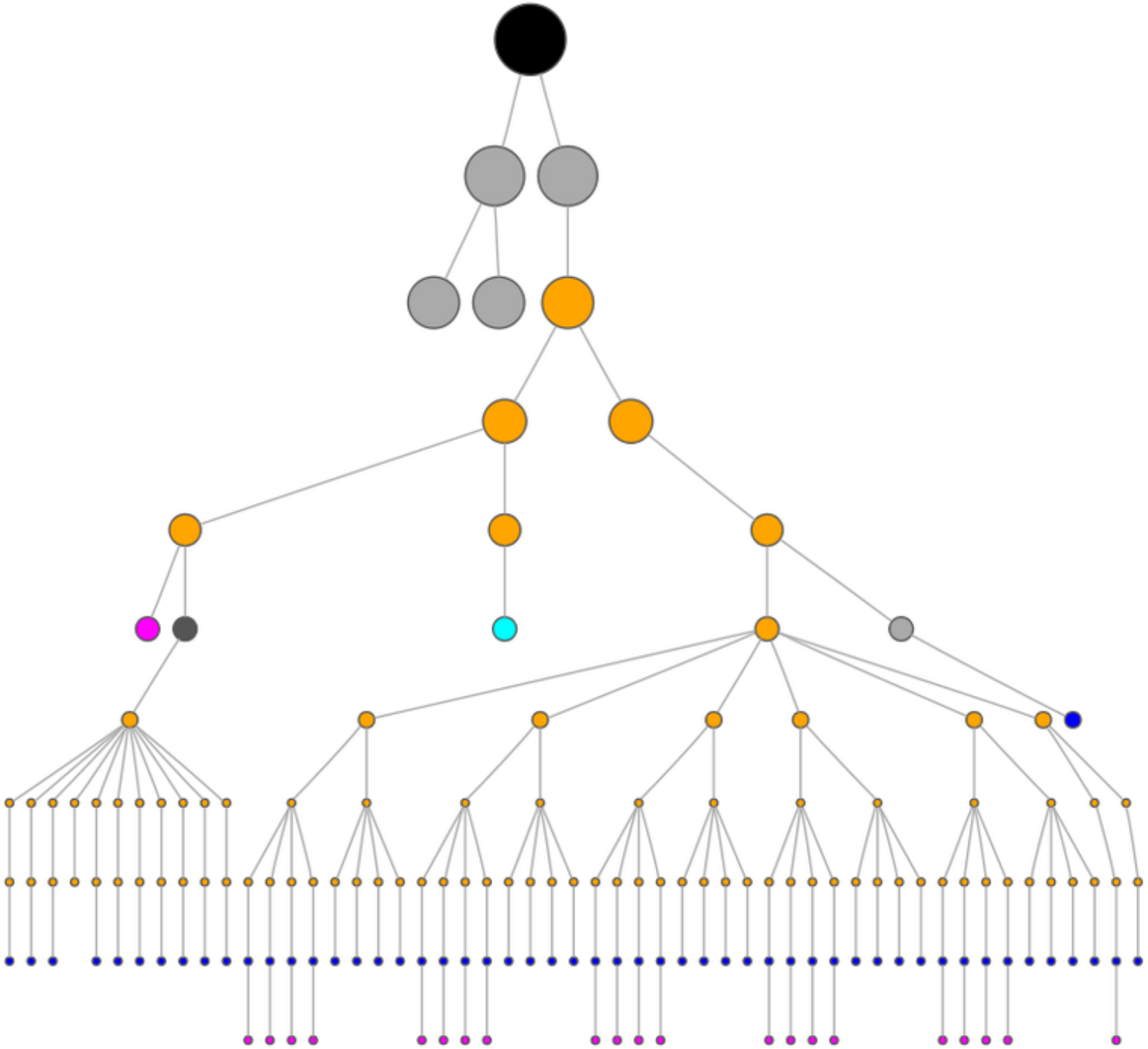
D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators
e - epics

- Reducers and Epics are peers.
- Both are appropriate for business logic. (Be consistent.)

Component state



D - Dispatcher
S - Store
V - View
A - Actions
r - reducers
s - selectors
a - action creators



Redux

Redux features

Redux

A Predictable State Container for JS Apps

Get Started

Predictable

Redux helps you write applications that **behave consistently**, run in different environments (client, server, and native), and are **easy to test**.

Centralized

Centralizing your application's state and logic enables powerful capabilities like **undo/redo**, **state persistence**, and much more.

Debuggable

The Redux DevTools make it easy to trace **when, where, why, and how your application's state changed**. Redux's architecture lets you log changes, use **"time-travel debugging"**, and even send complete error reports to a server.

Flexible

Redux **works with any UI layer**, and has **a large ecosystem of addons** to fit your needs.

Why use Redux?

Why use Redux?

- Principled state management.

Why use Redux?

- Principled state management.
- Everything else is an *implementation detail*.

Why use Redux?

- Principled state management.
- Everything else is an *implementation detail*.
- Alternatives?

Why use Redux?

- Principled state management.
- Everything else is an *implementation detail*.
- Alternatives?
  - Alt, Barracks, Delorean, disto, fluce, fluctuations, Flummox, Flux, Flux This, Fluxette, Fluxible, Fluxxor, Fluxy, Lux, Marty.js, Material Flux, McFly, microcosm, microflux, mmox, Nuclear.js, NuclearJS, OmniscientJS, Reducer, Redux, Reflux. (Flux variants circa 2015)

Why use Redux?

- Principled state management.
- Everything else is an *implementation detail*.
- Alternatives?
  - Alt, Barracks, Delorean, disto, fluce, fluctuations, Flummox, Flux, Flux This, Fluxette, Fluxible, Fluxxor, Fluxy, Lux, Marty.js, Material Flux, McFly, microcosm, microflux, mmox, Nuclear.js, NuclearJS, OmniscientJS, Reducer, Redux, Reflux. (Flux variants circa 2015)
  - Other state management libraries? *Literally thousands*.

Why use Redux?

- Principled state management.
- Everything else is an *implementation detail*.
- Alternatives?
  - Alt, Barracks, Delorean, disto, fluce, fluctuations, Flummox, Flux, Flux This, Fluxette, Fluxible, Fluxxor, Fluxy, Lux, Marty.js, Material Flux, McFly, microcosm, microflux, mmox, Nuclear.js, NuclearJS, OmniscientJS, Reducer, Redux, Reflux. (Flux variants circa 2015)
  - Other state management libraries? *Literally thousands*.
- ...often overlooked: Async support (via middleware).

"You might not need Redux"

"You might not need Redux"

"You might not need Redux"

Local state is fine.

"You might not need Redux"

Local state is fine.

Redux features vs conceptual model

Redux features vs conceptual model

vs

"When" do you need Redux?

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

- When you have cascading renders, or render race conditions, or render performance problems.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

- When you have cascading renders, or render race conditions, or render performance problems.

  No: find the errors and fix them; they will eventually show up in other contexts.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

- When you have cascading renders, or render race conditions, or render performance problems.

  No: find the errors and fix them; they will eventually show up in other contexts.

- When promises resolve after a component unmounts.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

- When you have cascading renders, or render race conditions, or render performance problems.

  No: find the errors and fix them; they will eventually show up in other contexts.

- When promises resolve after a component unmounts.

  No: this is not a React problem (vanilla DOM API too); program more defensively; use XHR instead of Fetch (cancellable); use Observables.

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

- When you have cascading renders, or render race conditions, or render performance problems.

  No: find the errors and fix them; they will eventually show up in other contexts.

- When promises resolve after a component unmounts.

  No: this is not a React problem (vanilla DOM API too); program more defensively; use XHR instead of Fetch (cancellable); use Observables.

- To make React "scale".

"When" do you need Redux?

- When multiple non-ancestor components need the same state.

  No: context; controller components; render reducers.

- When you're prop-tunneling more than `N` layers deep.

  No: context; how many layers is `N`?

- When you have cascading renders, or render race conditions, or render performance problems.

  No: find the errors and fix them; they will eventually show up in other contexts.

- When promises resolve after a component unmounts.

  No: this is not a React problem (vanilla DOM API too); program more defensively; use XHR instead of Fetch (cancellable); use Observables.

- To make React "scale".

  No: Redux has scaling considerations too...as does all of software.

"When" don't you need Redux?

"When" don't you need Redux?

- When you only need "component state" or "view state".

"When" don't you need Redux?

- When you only need "component state" or "view state".

  Define the boundary between "view state" and "application state".

"When" don't you need Redux?

- When you only need "component state" or "view state".

  Define the boundary between "view state" and "application state".
  (Hint: there isn't one.)

"When" don't you need Redux?

- When you only need "component state" or "view state".

  Define the boundary between "view state" and "application state".
  (Hint: there isn't one.)

- Start with component state and move it to Redux if/when you need to.

"When" don't you need Redux?

- When you only need "component state" or "view state".

  Define the boundary between "view state" and "application state".
  (Hint: there isn't one.)

- Start with component state and move it to Redux if/when you need to.

  Possible:

  - *Purposefully* model component state with the reducer pattern.
  - Rely on external data via actions.
  - *Never* use external data (like from other Hooks).
    - This means no API calls (the `useEffect` execution semantics are too different).

  Otherwise a refactoring nightmare.

Redux isn't needed anymore

- "Just use context."
- "Just use Hooks."
- "useState is so much simpler."
- "useReducer is the same thing."
- "Context + useReducer is Redux."

Context

Context

- Often overblown.

Context

- Often overblown.
- Simple publish/subscribe (good), buried inside a component API (bad).

Context

- Often overblown.
- Simple publish/subscribe (good), buried inside a component API (bad).
- Use as you would a module-level variable, but with React machinery instead of JavaScript machinery.

Context

- Often overblown.
- Simple publish/subscribe (good), buried inside a component API (bad).
- Use as you would a module-level variable, but with React machinery instead of JavaScript machinery.
- Principled state management is (entirely) up to you.

Context

- Often overblown.
- Simple publish/subscribe (good), buried inside a component API (bad).
- Use as you would a module-level variable, but with React machinery instead of JavaScript machinery.
- Principled state management is (entirely) up to you.
- (react-redux uses it as an implementation detail.)

Context

- Often overblown.
- Simple publish/subscribe (good), buried inside a component API (bad).
- Use as you would a module-level variable, but with React machinery instead of JavaScript machinery.
- Principled state management is (entirely) up to you.
- (react-redux uses it as an implementation detail.)
- Good for side-loading external data...when you have a small number of individual contexts.

Context

- Often overblown.
- Simple publish/subscribe (good), buried inside a component API (bad).
- Use as you would a module-level variable, but with React machinery instead of JavaScript machinery.
- Principled state management is (entirely) up to you.
- (react-redux uses it as an implementation detail.)
- Good for side-loading external data...when you have a small number of individual contexts.

```
return (
  <LocalStyle theme={lightTheme}>
    <ErrorBoundary fallback="Something went very wrong, try refreshing?">
      <LiveAnnouncer>
        <Store initialStore={initialStore}>
          <OptimizelyWrapper>
            <NotificationsProvider>
              <QueryClientProvider client={queryClient}>
                <APIContextProvider>
                  <APICacheProvider initial={apiCache}>
                    <SubscriptionProvider>
                      <FlaggedThemeProvider>
                        <HelmetProvider context={helmetContext}>
                          <OfflineNotice />
                          {children}
                        </HelmetProvider>
                      </FlaggedThemeProvider>
                    </SubscriptionProvider>
                  </APICacheProvider>
                </APIContextProvider>
              </QueryClientProvider>
            </NotificationsProvider>
          </OptimizelyWrapper>
        </Store>
      </LiveAnnouncer>
    </ErrorBoundary>
  </LocalStyle>
);
```

Hooks

- Nice, succinct API.
- Entirely dependent on very specific React internal behavior.
- Still new and unproven for long-term maintenance. Beware the hype.

`useState`

- No principled data management.
- Usually said by developers who started with React after hooks were introduced and who don't have experience with class component patterns for wrangling component state.

`useReducer`

- The reducer pattern applied to component state.
- Good for organizing data in a single component.
- Possible to reuse across multiple components (with work).
- No side-loading data.
- Async possible via action creator wrappers around `dispatch` (thunks-like usage).

Context + `useReducer`

- Contender with caveats.

Context + `useReducer`

- Contender with caveats.
- Can recreate Redux using React-specific APIs.

Context + `useReducer`

- Contender with caveats.
- Can recreate Redux using React-specific APIs.
- Small to implement; no need for react-redux addition.

Context + `useReducer`

- Contender with caveats.
- Can recreate Redux using React-specific APIs.
- Small to implement; no need for react-redux addition.
- Async possible via thunk-like "middleware" wrapper function around `dispatch`.

Context + `useReducer`

- Contender with caveats.
- Can recreate Redux using React-specific APIs.
- Small to implement; no need for react-redux addition.
- Async possible via thunk-like "middleware" wrapper function around `dispatch`.
- Entirely dependent on several independent React machineries:
  - Cannot use outside of React *context*.
  - Cannot use outside of React *components*.
  - Cannot use outside of React *hooks*.

Context + `useReducer`

- Contender with caveats.
- Can recreate Redux using React-specific APIs.
- Small to implement; no need for react-redux addition.
- Async possible via thunk-like "middleware" wrapper function around `dispatch`.
- Entirely dependent on several independent React machineries:
  - Cannot use outside of React *context*.
  - Cannot use outside of React *components*.
  - Cannot use outside of React *hooks*.
- Testing? Yes for the reducer. No for the action creators, actions, or thunks.
  (Needed parts are hidden inside the Hooks execution sandbox.)

Context + `useReducer`

- Contender with caveats.
- Can recreate Redux using React-specific APIs.
- Small to implement; no need for react-redux addition.
- Async possible via thunk-like "middleware" wrapper function around `dispatch`.
- Entirely dependent on several independent React machineries:
  - Cannot use outside of React *context*.
  - Cannot use outside of React *components*.
  - Cannot use outside of React *hooks*.
- Testing? Yes for the reducer. No for the action creators, actions, or thunks.
  (Needed parts are hidden inside the Hooks execution sandbox.)
- Worth investigating?

Why do *we* use Redux?

- Popular Flux implementation.
- Well documented.
- Lots of support options.

Why is popularity waning?

- Boilerplate.
- Hooks hype (component state revival).
- Missing features & downsides.

What are the downsides?

What are the downsides?

- Simple.

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.
  - Reducers can't differentiate between "state owner" and "interested parties". (Read/write vs read-only.)

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.
  - Reducers can't differentiate between "state owner" and "interested parties". (Read/write vs read-only.)
- No (direct) correlation between action and state-change. (Dev Tools hack around this.)

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.
  - Reducers can't differentiate between "state owner" and "interested parties". (Read/write vs read-only.)
- No (direct) correlation between action and state-change. (Dev Tools hack around this.)
- Debugging indirection. (Dev Tools action -> action creator -> action constant -> reducer or epic. Then somehow find which state is used by which component.)

What are the downsides?

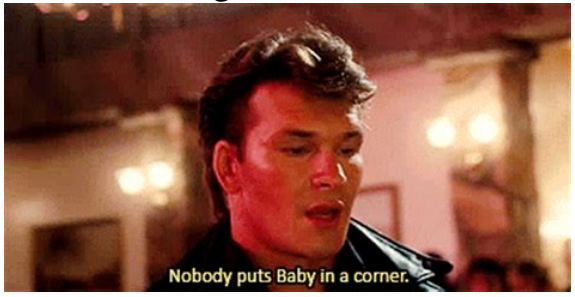
- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.
  - Reducers can't differentiate between "state owner" and "interested parties". (Read/write vs read-only.)
- No (direct) correlation between action and state-change. (Dev Tools hack around this.)
- Debugging indirection. (Dev Tools action -> action creator -> action constant -> reducer or epic. Then somehow find which state is used by which component.)
- Can't update state without triggering renders.

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.
  - Reducers can't differentiate between "state owner" and "interested parties". (Read/write vs read-only.)
- No (direct) correlation between action and state-change. (Dev Tools hack around this.)
- Debugging indirection. (Dev Tools action -> action creator -> action constant -> reducer or epic. Then somehow find which state is used by which component.)
- Can't update state without triggering renders.
- Effects management is 2nd-class.

What are the downsides?

- Simple.
- Unware of component life cycle. (No initialization. No clean-up.)
- Unware of who is using what data. (No remove unused data.)
- Global store:
  - "Normalize the store." (Hard.)
  - Can get messy if everyone can see (and change) everything.
  - Namespacing (`combineReducers`) makes a global store *optionally* global.
  - Reducers can't differentiate between "state owner" and "interested parties". (Read/write vs read-only.)
- No (direct) correlation between action and state-change. (Dev Tools hack around this.)
- Debugging indirection. (Dev Tools action -> action creator -> action constant -> reducer or epic. Then somehow find which state is used by which component.)
- Can't update state without triggering renders.
- Effects management is 2nd-class.



Deconstructing Redux

Why deconstruct?

- Redux (and/or React) often introduced via create-react-app (et al).
- Feels complex when it shouldn't.
- Reducing/removing "magic" helps you make informed decisions.

Why deconstruct?

- Redux (and/or React) often introduced via create-react-app (et al).
- Feels complex when it shouldn't.
- Reducing/removing "magic" helps you make informed decisions.
- If you don't understand the 'why' then you end up following the patterns regardless if they're helpful or not in a given situation.

`map`, `filter`, `reduce`

```
const oldArray = [0, 1, 2, 3, 4]
```

`map`, `filter`, `reduce`

```
const oldArray = [0, 1, 2, 3, 4]
```

```
const newArray = oldArray.map(x => x + 1)
// [ 1, 2, 3, 4, 5 ]
```

`map`, `filter`, `reduce`

```
const oldArray = [0, 1, 2, 3, 4]
```

```
const newArray = oldArray.map(x => x + 1)
// [ 1, 2, 3, 4, 5 ]
```

```
const evensArray = oldArray.filter(x => x % 2 === 0)
// [ 0, 2, 4 ]
```

`map`, `filter`, `reduce`

```
const oldArray = [0, 1, 2, 3, 4]
```

```
const newArray = oldArray.map(x => x + 1)
// [ 1, 2, 3, 4, 5 ]
```

```
const evensArray = oldArray.filter(x => x % 2 === 0)
// [ 0, 2, 4 ]
```

```
const totalCount = oldArray.reduce((acc, cur) => {
    acc += cur
    return acc
}, 0)
// 10
```

The 'reducer' pattern

```
const oldState = {count: 0}

const actions = [
    {type: 'INC', payload: 1},
    {type: 'DEC', payload: -1},
    {type: 'INC', payload: 1},
    {type: 'INC', payload: 1},
]

const newState = actions.reduce((acc, cur) => {
    return {...acc, count: acc.count + cur.payload }
}, oldState);

// { count: 2 }
```

```
await import('https://unpkg.com/redux@4.1.2/dist/redux.js')

// Actions
const A = { INC: 'INC', DEC: 'DEC' }

// Store
const defaultState = {count: 0}
const myReducer = (state = defaultState, action) => {
  switch (action.type) {
    case A.INC: return { ...state, count: state.count + 1 }
    case A.DEC: return { ...state, count: state.count - 1 }
    default: return state
  }
}
const Store = Redux.createStore(myReducer)

// View
const myView = (props) => `<div>My Counter: ${props.count}.<br>
  <button onclick="((ev) =>
    Store.dispatch({ type: A.INC }))(event)">+</button>
  <button onclick="((ev) =>
    Store.dispatch({ type: A.DEC }))(event)">-</button>
</div>`

// App
document.body.innerHTML = myView(Store.getState())
Store.subscribe(() =>
  document.body.innerHTML = myView(Store.getState()))
```

react-redux (classes)

```
const connect = (config) => (MyComponent) => {
    class MyWrapper extends React.Component {
        componentDidMount() {
            this.sub = Store.subscribe(() => {
                this.latestState = Store.getState()
                this.forceUpdate()
            })
        }

        componentWillUnmount() {
            this.sub.unsubscribe()
        }

        render() {
            return React.createElement(MyComponent,
                this.latestState)
        }
    }
}
```

react-redux (hooks)

```
const useSelector = fn => {
  var latestState = useRef(fn(Store.getState()))
  var [_, forceRender] = useState(0)

  useEffect(() => {
    const sub = Store.subscribe(() => {
      latestState.current = fn(Store.getState())
      forceRender((c) => (c + 1) % 10)
    }

    return () => sub.unsubscribe()
  }, [])

  return latestState.current
}
```

```
const createStore = (reducer, initialState) => {
  let _state = initialState
  let _subscribers = []

  const ret = {
    getState: () => _state,
    dispatch: (action) => {
      const newState = [action].reduce(reducer, _state)
      console.log(action.type, {action, oldState: _state, newState})
      if (newState !== _state) {
        _state = newState
        _subscribers.forEach((cb) => cb())
      }
    },
    subscribe: (cb) => {
      _subscribers.push(cb)
      return {unsubscribe: () => {
        const idx = _subscribers.indexOf(cb)
        if (idx !== -1) { _subscribers.splice(idx, 1) }
      }}
    },
  }

  // Dispatch nothing to seed initial state from reducer.
  ret.dispatch({})
  return ret
}
```

`combineReducers`

```
const foo = (state, action) => { /* ... */ }
const bar = (state, action) => { /* ... */ }

const rootReducer = combineReducers({foo, bar})
// {foo: {...fooStateHere}, bar: {...barStateHere}}
```

Our patterns

(Collected from our four biggest & oldest codebases.)

Summary of problem patterns

Summary of problem patterns

```
commit d26c0a4f1928c4be10ce426a952e786d7ab41bc7

    Add Redux boilerplate for addition Foo
---
 src/redux/actions/Foo.js                  | 14 +++++++++
 src/redux/epics/Foo.js                    | 38 ++++++++++++++
 src/redux/epics/__tests__/Foo-test.js     | 48 +++++++++++++++++++
 src/redux/epics/index.js                  |  4 +++
 src/redux/reducers/Foo.js                 |  2 ++
 5 files changed, 104 insertions(+)
```

Redux isn't fun to use.

Who should we blame?

Redux isn't fun to use.

Who should we blame?

- Organic growth.
- Shifting patterns and libraries.
- Big codebases.
- Care not to disrupt tens of millions of users.
- Deadlines.
- Customer demands.

Redux isn't fun to use.

Who should we blame?

- Organic growth.
- Shifting patterns and libraries.
- Big codebases.
- Care not to disrupt tens of millions of users.
- Deadlines.
- Customer demands.
- Nobody. These are Redux-community standard patterns.

Redux isn't fun to use.

Who should we blame?

- Organic growth.
- Shifting patterns and libraries.
- Big codebases.
- Care not to disrupt tens of millions of users.
- Deadlines.
- Customer demands.
- Nobody. These are Redux-community standard patterns.
- Sam.

Proposals

Proposals

You want to transition *how much* code?

Project	JavaScript SLOC
Biggest	143,625
Important	75,736
Big	51,774
Newcomer	33,603

Pattern: action creators

```
export const ActionTypes = {
  FOO_DO_THING: 'foo/do/thing',
  FOO_LOAD_THINGS: 'foo/load/thing',
  FOO_LOAD_THINGS_SUCCESS: 'foo/load/success',
  FOO_LOAD_THINGS_ERROR: 'foo/load/error',
  // ...etc
}

export const doThing = payload => ({
  type: ActionTypes.FOO_DO_THING,
  payload,
})
export const loadThings = () => ({
  type: ActionTypes.FOO_LOAD_THINGS,
})
// ...etc

export default dispatch => ({
  doThing = payload => dispatch(doThing),
  loadThings = payload => dispatch(loadThings),
  // ...etc
})
```

Proposal: no action creators

```
export const ActionTypes = {
  FOO_DO_THING: 'FOO_DO_THING',
  FOO_LOAD_THINGS: 'FOO_LOAD_THINGS',
  FOO_LOAD_THINGS_SUCCESS: 'FOO_LOAD_THINGS_SUCCESS',
  FOO_LOAD_THINGS_ERROR: 'FOO_LOAD_THINGS_ERROR',
  // ...etc
}
```

Proposal: no action creators

```
export const ActionTypes = {
  FOO_DO_THING: 'FOO_DO_THING',
  FOO_LOAD_THINGS: 'FOO_LOAD_THINGS',
  FOO_LOAD_THINGS_SUCCESS: 'FOO_LOAD_THINGS_SUCCESS',
  FOO_LOAD_THINGS_ERROR: 'FOO_LOAD_THINGS_ERROR',
  // ...etc
}
```

...or

```
export const ActionTypes = objMirror([
  'FOO_DO_THING',
  'FOO_LOAD_THINGS',
  'FOO_LOAD_THINGS_SUCCESS',
  'FOO_LOAD_THINGS_ERROR',
])
```

Proposal: no action creators (usage #1)

```
// src/components/MyView.js

import { useDispatch } from 'react-redux'
import { ActionTypes } from 'src/redux/actions/foo.js'

export const MyView = props => {
  const dispatch = useDispatch()

  return (
    <button
      onClick={ev =>
        dispatch({
          type: ActionTypes.DO_THING,
          payload: ev.target.value,
        })
      }
    >
      Do thing!
    </button>
  )
}
```

Proposal: no action creators (usage #2)

```
// src/components/MyView.js

import { send } from 'src/redux/store'
import { ActionTypes } from 'src/redux/actions/foo.js'

export const MyView = props => {
  return (
    <button
      onClick={ev => send(ActionTypes.DO_THING, ev.target.value)}
    >
      Do thing!
    </button>
  )
}
```

Proposal: no action creators (usage #3)

```
// src/components/MyView.js

import { send } from 'utils/Redux'
import { ActionTypes } from 'src/redux/actions/foo.js'

export const MyView = props => {
  return (
    <button onClick={send(ActionTypes.DO_THING)}>
      Do thing!
    </button>
  )
}
```

Pattern: reducers via `createReducer`

```
// src/redux/reducers/foo.js

import { ActionTypes } from 'src/redux/actions/foo.js'

const doThing = (state, action) => { /* ... */ }

const thingsLoading = (state, action) => { /* ... */ }

const thingsLoaded = (state, action) => { /* ... */ }

export const myReducer = createReducer(defaultState, {
  [ActionTypes.DO_THING]: doThing,
  [ActionTypes.THINGS_LOADING]: thingsLoading,
  [ActionTypes.THINGS_LOADED]: thingsLoaded,
})
```

Pattern: unnecessary destructuring

```
import { ActionTypes } from 'src/redux/actions/foo.js'

const {
  DO_THING,
  THINGS_LOADING,
  THINGS_LOADED,
} = ActionTypes
```

Pattern: unnecessary destructuring

```
import { ActionTypes } from 'src/redux/actions/foo.js'

const {
  DO_THING,
  THINGS_LOADING,
  THINGS_LOADED,
} = ActionTypes
```

I digress:

```
const {
  foo,
  bar,
  baz,
  qux,
  quux,
  quuz,
  corge,
  grault,
  // ...etc
} = props // or state, or Redux, or whatever
```

Proposal: only destructure with purpose

- Rename variables:

```
const { foo as MyFoo } = someObj
```

- Omit values:

```
const { foo, ...everythingButFoo } = someObj
```

- Tee up local variables for object creation using shorthand properties:

```
const { foo, bar } = someObj
const newObj = { foo, bar, otherVar }
```

Proposal: only destructure with purpose

- Rename variables:

```
const { foo as MyFoo } = someObj
```

- Omit values:

```
const { foo, ...everythingButFoo } = someObj
```

- Tee up local variables for object creation using shorthand properties:

```
const { foo, bar } = someObj
const newObj = { foo, bar, otherVar }
```

- Use aliases to reduce typing (be consistent!):

```
import { ActionTypes as A } from 'src/redux/actions/foo.js'
```

Pattern: reducers via `switch`

```
import { ActionTypes } from 'src/redux/actions/foo.js'

const thingsLoadedReducer = (state, action) => { /* ... */ }

export const myReducer = (state = defaultState, action) => {
  switch (action.type) {
    case ActionTypes.DO_THING:
      return /* ... */
    case ActionTypes.THINGS_LOADING:
      return {...state, loading: true}
    case ActionTypes.THINGS_LOADED:
      return thingsLoadedReducer(state, action)
    default:
      return state
  }
}
```

Proposal: colocate action with reducer

```
const reducers = {
    [A.INC]: (state, payload) => {
        return {...state, count: state.count + 1};
    },
    [A.DEC]: (state, payload) => {
        return {...state, count: state.count - 1};
    },
};

// or
reducers = {}
reducers[A.INC] = (state, payload) => {
    return {...state, count: state.count + 1};
}
reducers[A.DEC] = (state, payload) => {
    return {...state, count: state.count - 1};
}
```

Proposal: colocate action with reducer

```
const reducers = {
    [A.INC]: (state, payload) => {
        return {...state, count: state.count + 1};
    },
    [A.DEC]: (state, payload) => {
        return {...state, count: state.count - 1};
    },
};

// or
reducers = {}
reducers[A.INC] = (state, payload) => {
    return {...state, count: state.count + 1};
}
reducers[A.DEC] = (state, payload) => {
    return {...state, count: state.count - 1};
}
```

```
const initialState = {/* ... */}
export myReducer = makeReducer(reducers, initialState)
```

Pattern: for the sake of "the pattern"

```
import { ActionTypes } from 'src/redux/actions/foo.js'

const doThing = (state, action) => {
    ...state,
    foo: action.payload,
}

const doOtherThing = (state, action) => {
    ...state,
    bar: action.payload,
}

const doYetAnotherThing = (state, action) => {
    ...state,
    bar: action.payload,
}
```

Proposal: refactor, combine, compose

```
// {type: 'FOO', payload: {foo: {foo: 'Foo!'}}}
// {type: 'BAR', payload: {bar: {bar: 'Bar!'}}}
// {type: 'BAZ', payload: {baz: {baz: 'Baz!'}}}
const passthrough = (state, action) => {
    ...state,
    ...action.payload,
}
```

Proposal: refactor, combine, compose

```
// {type: 'FOO', payload: {foo: {foo: 'Foo!'}}}
// {type: 'BAR', payload: {bar: {bar: 'Bar!'}}}
// {type: 'BAZ', payload: {baz: {baz: 'Baz!'}}}
const passthrough = (state, action) => {
    ...state,
    ...action.payload,
}
```

```
const toggleLoading = (key) => (state, action) =>
  ({ ...state, [key]: !state[key] })

const setStateFromPayload = (state, action) =>
  ({...state, ...action.payload })

const reducers = {
    [A.LOADING]: toggleLoading('loading'),
    [A.LOADED]: composeReducers(
      toggleLoading('loading'),
      setStateFromPayload,
    ),
}
```

Pattern: same-y epics

```
export const someLoadEpic = (actions$, _, { theLoadAPIFn }) =>
  actions$.pipe(
    ofType(ActionTypes.LOAD_THING),
    switchMap(() =>
      theLoadAPIFn().pipe(
        map(rep => {
          return {
            type: ActionTypes.LOAD_THING_SUCCESS,
            payload: rep.some_value,
          }
        }),
        catchError(err => {
          return of({
            type: ActionTypes.LOAD_THING_ERROR,
            payload: err,
          })
        }),
      ),
    ),
  )
```

Proposal: `toType` operator

```
export const someLoadEpic = (actions$, _, { theLoadAPIFn }) =>
  actions$.pipe(
    ofType(ActionTypes.LOAD_THING),
    switchMap(() =>
      theLoadAPIFn().pipe(
        toType(
          ActionTypes.LOAD_THING_SUCCESS,
          ActionTypes.LOAD_THING_ERROR,
          // Optional:
          (rep) => ({ someValue: rep.some_value }),
        )
      ),
    ),
  )
```

Proposal: `toType` operator (alt)

```
epics = {}

epics[A.LOAD_THING] = (actions$, _, { theLoadAPIFn }) =>
  actions$.pipe(
    switchMap(() =>
      theLoadAPIFn().pipe(
        toType(
          ActionTypes.LOAD_THING_SUCCESS,
          ActionTypes.LOAD_THING_ERROR,
          // Optional:
          (rep) => ({ someValue: rep.some_value }),
        )
      ),
    ),
  )

export myEpic = makeEpic(epics)
```

Pattern: manual import/export list

```
import { foo, bar, baz } from 'src/redux/epics/Foo'
import { qux, quux, quz } from 'src/redux/epics/Qux'

export const rootEpic = combineEpics(
  foo,
  bar,
  baz,
  qux,
  quux,
  quz,
  // ...etc
)
```

Proposal: wildcard epic imports

```
import * as FooEpics from 'src/redux/epics/Foo'
import * as QuxEpics from 'src/redux/epics/Qux'

export const rootEpic = combineEpics(
  ...Object.values(FooEpics),
  ...Object.values(QuxEpics),
)
```

Pattern: same-y epic tests

```
it('load thing successfully', () => {
  const ctx = { loadThing: () => of('SUCCESS') }

  expectRx.toMatchObject.run(({ hot, expectObservable }) => {
    const input$ = hot('a', { a: actions.loadThing() })

    expectObservable(epics.loadThing(input$, null, ctx)).toBe('a', {
      a: { type: ActionTypes.LOAD_THING_SUCCESS },
    })
  })
})
it('load thing unsuccessfully', () => {
  const ctx = { loadThing: () => throwError('FAIL') }

  expectRx.toMatchObject.run(({ hot, expectObservable }) => {
    const input$ = hot('a', { a: actions.loadThing() })

    expectObservable(epics.loadThing(input$, null, ctx)).toBe('a', {
      a: { type: ActionTypes.LOAD_THING_ERROR },
    })
  })
})
```

Proposal: test generator for simple epics

```
describe('some module tests', () => {
  genTestToType('load the thing',
    A.LOAD_THING,
    A.LOAD_THING_SUCCESS,
    A.LOAD_THING_ERROR,
    epics.loadThing,
    [ajaxCallFoo, ajaxCallBar],
  )
})
```

Summary of proposals

- Remove action creators (entirely).
- Use `dispatch()` in views -- or `send()` helper.
- Associate a reducer function directly with action constant.
- Treat reducer functions like functions, not pattern.
- Avoid unnecessary destructuring.
- Export reducer/epic to automatically combine it.

Summary of proposals

- Remove action creators (entirely).
- Use `dispatch()` in views -- or `send()` helper.
- Associate a reducer function directly with action constant.
- Treat reducer functions like functions, not pattern.
- Avoid unnecessary destructuring.
- Export reducer/epic to automatically combine it.
- What did I miss?