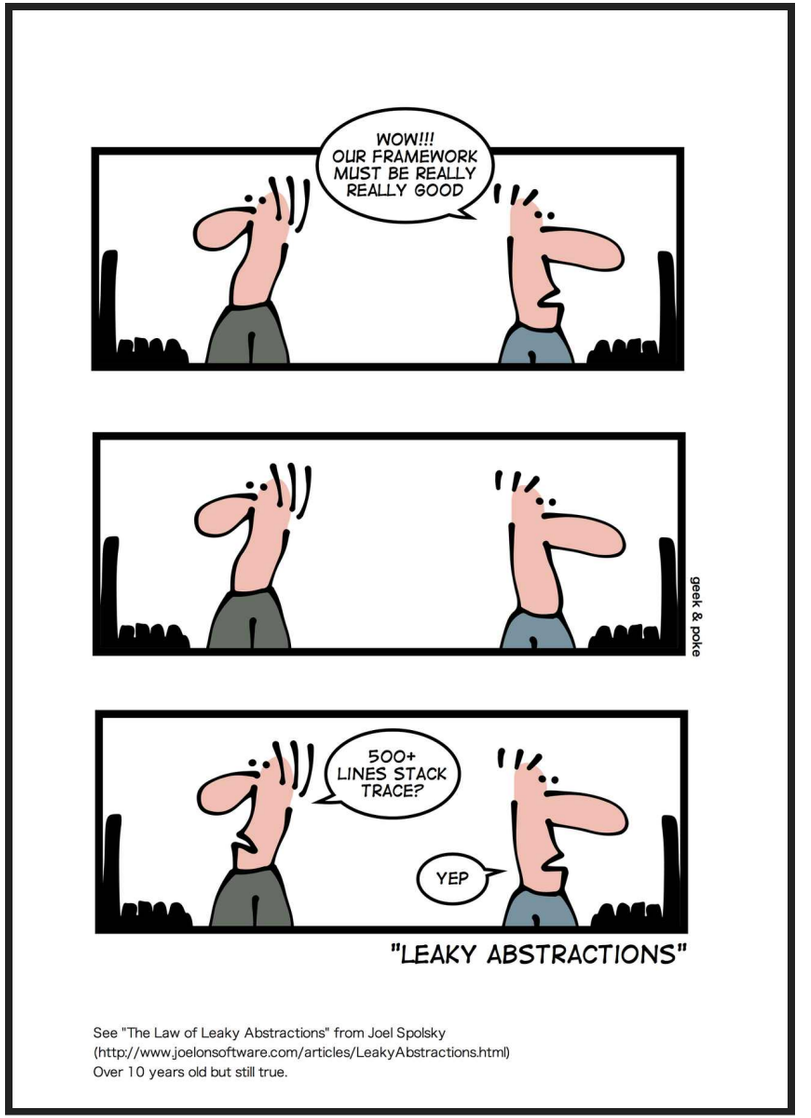
Lessons from the Trenches

Designing Resilient Bulletproof React/Redux Apps Part 2

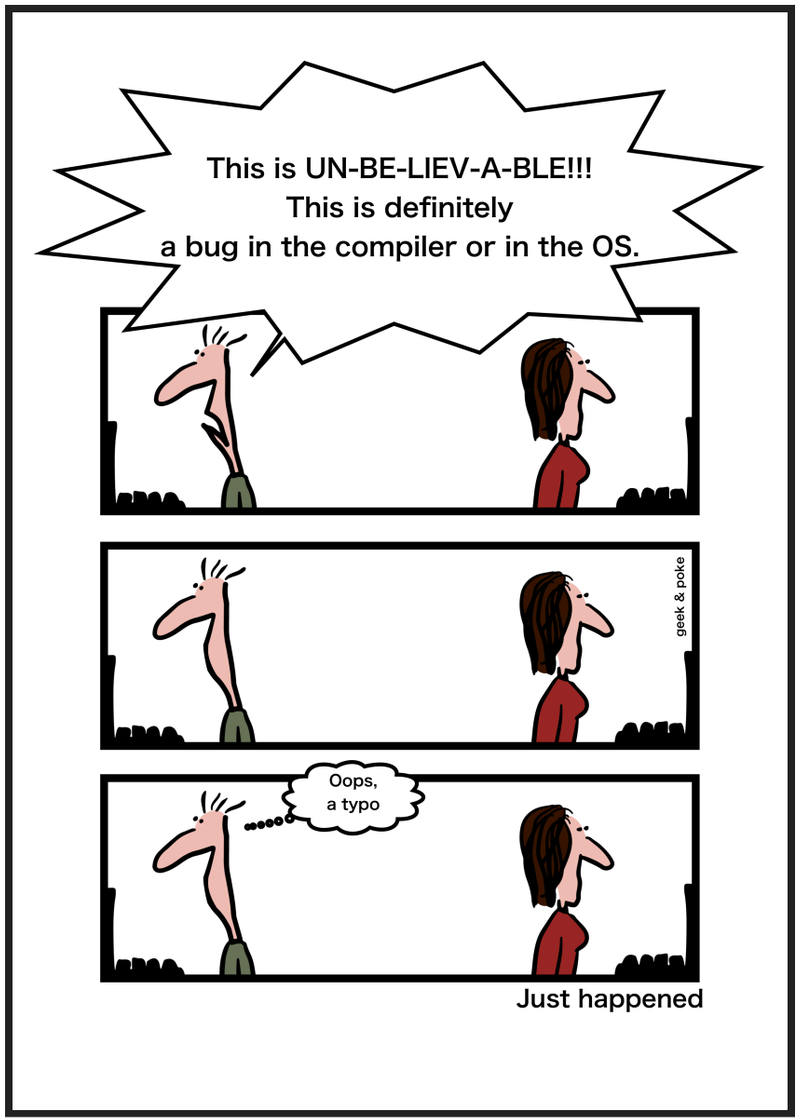
Connect.tech 2017

Jeff Barczewski

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



Good Framework by Geek & Poke Licensed CC BY 3.0



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.

What is business logic?

Types of business logic

- Validation – name and email required, name < 40 chars
- Verification – Do you have sufficient credits for this transaction?
- Authorization – Does your membership allow access to this?
- Transformation – Unauthorized action converted to upsell popup action
- A/B Logic – Half users get X, half get Y
- Augment – Add unique ID; Fill in user profile using uid
- Silence/filter – Resource is at max, ignore further increases
- Async Processing – Fetch data, post order, hide notification after delay, subscribe to updates
- Cancellation – Navigate to different page or user, change search
- Debounce - Wait till pauses before search, Throttle - only every 10ms

Foundational Questions

- What are reactive apps?
- What is asynchronous I/O?
- Why should I care?

Back in 1995

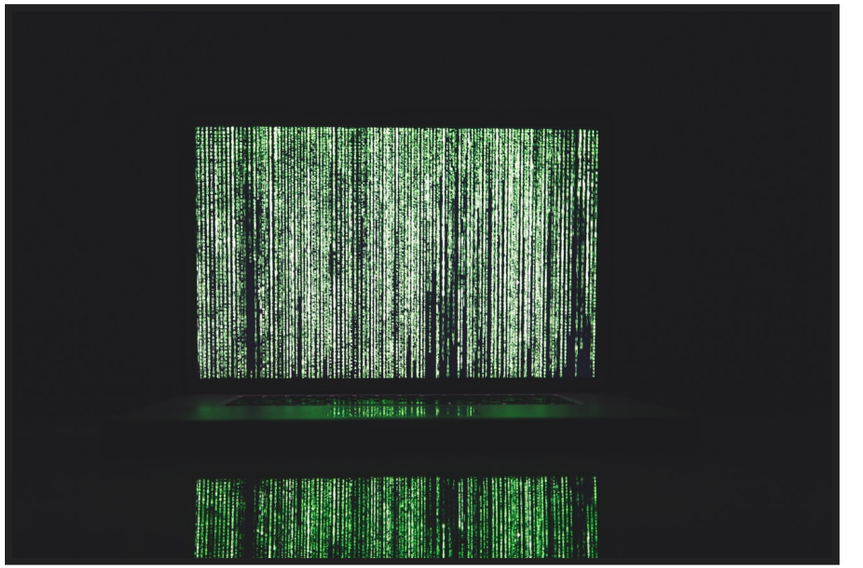
1. **Browser** makes **single request** to server
2. **Server** responds with full page of **HTML**
3. **Browser** renders the **HTML**
4. User clicks link, repeat starting with 1

Today (w/o HTTP/2)

1. **Browser** makes **initial** request to server
2. **Server** responds with **partial** page
3. **Browser** fetches lots of **js** and **data**
4. **Server(s)** respond with the **js** and **data**
5. **Browser** runs **js** and might fetch **more data**
6. **Browser** renders incomplete page
7. **Server(s)** responds with additional data
8. **Browser** upgrades connection to a **web socket**
9. **Server** pushes data over web socket as it is available
10. **Browser** updates page as **new data** comes in

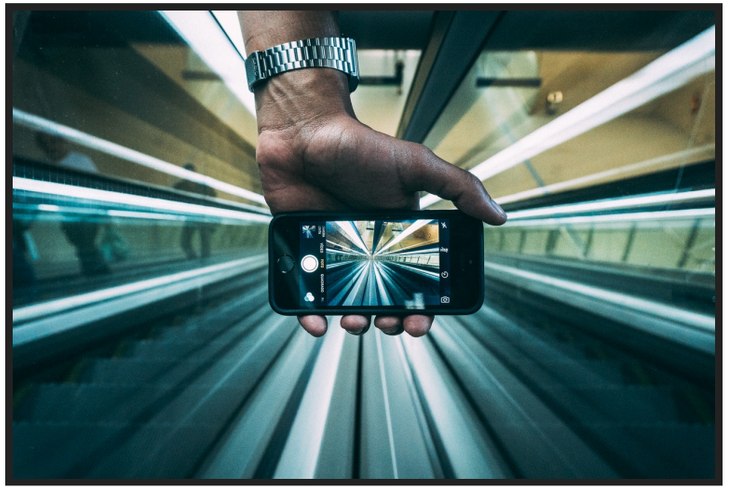
Today with HTTP/2

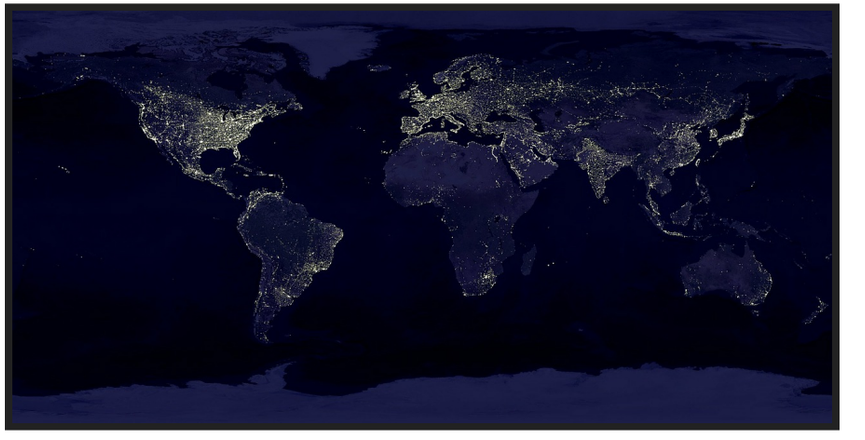
1. **Browser** makes **initial request** to server
2. **Server** responds with partial page and **predicted js/data**
3. **Browser** makes **N concurrent** requests using same connection to get additional data
4. **Server** responds to the concurrent requests over the same connection
5. **Browser** upgrades connection to a **web socket**
6. **Server** pushes data as it is available
7. **Browser** updates page as **new data** comes in



Today's Challenges

- Data needs to be pushed in real time
- Apps and pages hold state to improve response times and even provide offline experiences
- Data no longer lives on one server but is distributed
- Each server may only have a piece of the data
- Server's need to be predictive and resilient
- Browser is orchestrating lots of asynchronous activity
- New I/O - voice, location, motion, VR, AR





Orchestrating async tasks and I/O

There are many ways to manage async operations in your code.

What are your favorites?

Ways to deal with async tasks and I/O

- callbacks
- promises
- async/await
- observables
- other functional approaches (futures, most.js, monads)

Promises

- ES6 added Promises A+ to the standard
- One of the most popular techniques for dealing with async work
- Run some code and eventually return a value or an error
- fetch and axios HTTP clients return promises

```
axios.get('https://survey.codewinds.com/polls')
    .then(resp => {
      // work with the data
      const polls = resp.data.polls;
      const filteredPolls = polls.filter(filterFn);
      return filteredPolls;
    })
    .then(filteredPolls => dispatch({
      type: FETCH_POLLS_SUCCESS,
      payload: filteredPolls
    }));
```

Promises p2

```
const prom = new Promise((resolve, reject) => {
 // do something and later resolve/reject
 resolve(123); // or reject(new Error('Bad...'))
});

const resolvingProm = Promise.resolve(42);
const rejectingProm = Promise.reject(new Error('foo'));

const p2 = prom.then(successFn, errorFn); // work with success and error

const p3 = prom.catch(err => {
  // do something regarding the error, log or other then
  // can return something else or could just re-throw
  return []; // no todo items were found to continue the flow
});
```

Promises p3

```
const pollsAndUsersProm = Promise.all([
  axios.get('https://survey.codewinds.com/polls')
    .then(resp => resp.data.polls)
  axios.get('https://server2/users')
    .then(resp => resp.data.users)
]).then(([polls, users]) => joinPollsUsers(polls, users));

const firstResponderPromise = Promise.race([
  axios.get('https://survey.codewinds.com/polls'),
  axios.get('https://survey2.codewinds.com/polls')
]);
```

Promises strengths

- Common / Popular
- Fairly simple - few concepts to learn
- Easy to convert from others
  - from callbacks (promisify)
- chain operations together working on a piece at a time
- `then()` and `catch()` wrap returned non-promise values with a promise
- Satisfies a decent number of use cases

Promise limitations

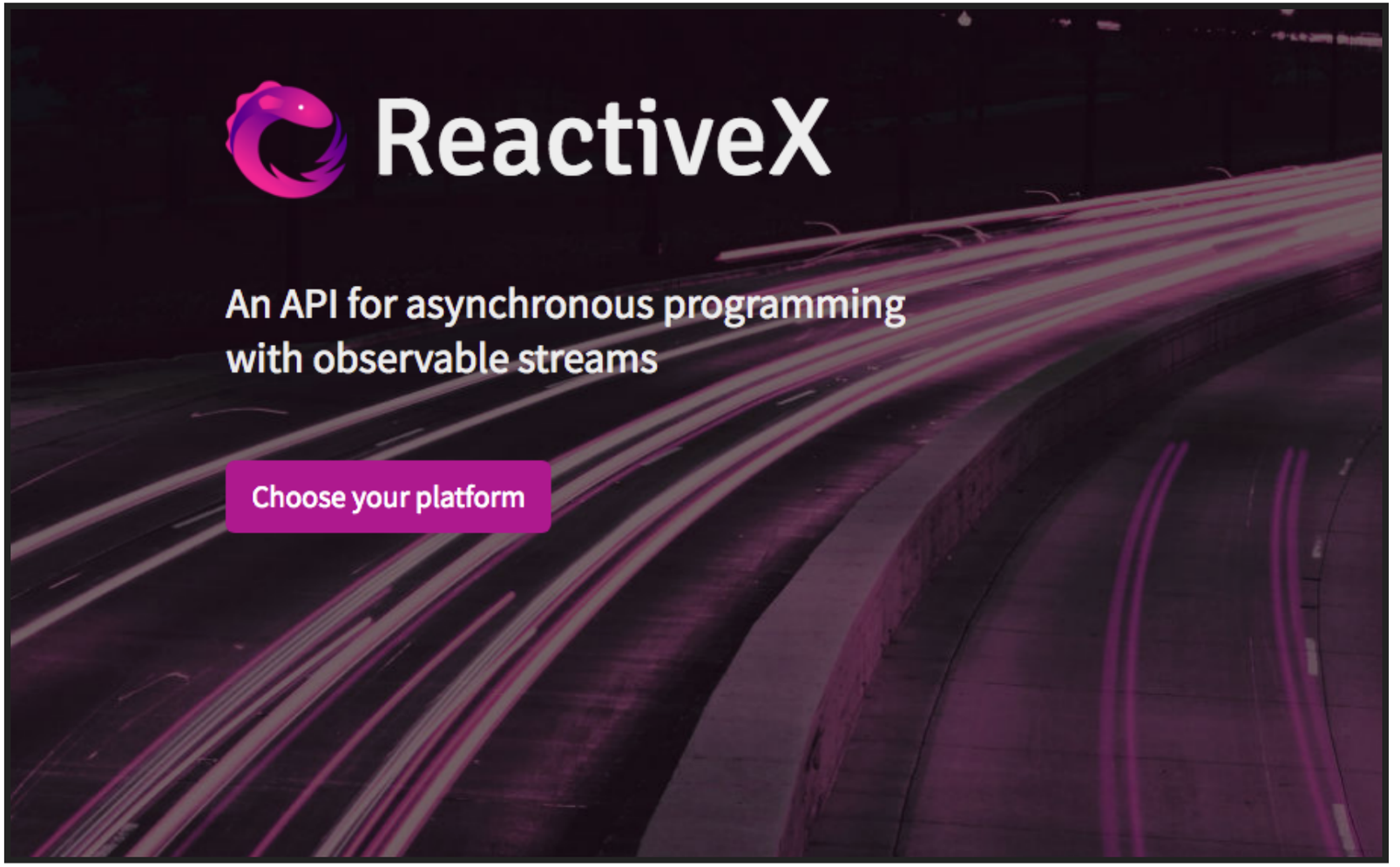
- Only resolves once (single value)
- Always resolves or rejects, no cancellation
- Always asynchronous
- Starts running code immediately, can't defer
- Not the easiest to access data between `then()` and `catch()` blocks, usually create a variable outside of the scope to share access

async/await (ES7)

- Eliminates much of the cermony with using promises
- Adoption in browsers is good, can polyfill
- Nice abstraction, looks synchronous (async / await informs)
- Still boils down to promises
  - single resolve or reject
  - no cancellation

```
async function fetchMyData() {
  const pollResp = await axios.get('https://survey.codewinds.com/polls')
  const polls = pollResp.data.polls;
  const topPoll = findTopPoll(polls);
  const topPollDataResp = await axios.get(`https://survey.codewinds.com/polls/topPol
  const topPollData = topPollDataResp.data.poll;
  // errors that are thrown are wrapped in a rejecting promise
  return topPollData; // will be wrapped in a promise
}
```

Observables



Remember ES5 array methods?

```
const arr = [10, 20, 30];
arr.forEach(x => console.log(x)); // 10, 20, 30
const bar = arr.filter(x => x < 25); // [10, 20]
const cat = arr.map(x => x * 10); // [100, 200, 300]

// chaining them together
arr
  .filter(x => x < 25)
  .map(x => x * 10)
  .forEach(x => console.log(x));
```

The Big Idea

Observables

What if we could operate on data in a similar composable fashion but it could arrive over time?

Observables (vs Promises)

- Zero to many values over time
- Cancellable
- Synchronous or Asynchronous
- Composable streams
- Observables can wait for subscriptions to start
- RxJS provides large set of operators

Observables

```
const ob$ = Rx.Observable.create(obs => {
  obs.next(10); // send first value
  obs.next(20); // 2nd
  obs.next(30); // 3rd
  obs.complete(); // we are done
});

ob$.subscribe(x => console.log(x));
// 10, 20, 30
```

demo

Observable Errors

```
const ob$ = Rx.Observable.create(obs => {
  obs.next(10); // send first value
  obs.next(20); // 2nd
  obs.error(new Error('something bad'));
});

ob$.subscribe(
 x => console.log(x),           // next
 err => console.error(err),     // errored
 () => console.log('complete') // complete
);
```

demo

ob$.subscribe()

Two forms are supported: separate fn args, or passing an object.

```
ob$.subscribe(
 x => console.log(x),           // next
 err => console.error(err),     // errored
 () => console.log('complete') // complete
);

// OR using object
ob$.subscribe({
 next: x => console.log(x),              // next
 error: err => console.error(err),       // errored
 complete: () => console.log('complete') // complete
});
```

You don't have to define all, just subscribe to what you care about.

Observable.of()

Simple way to create an observable with a known number of values.

```
// emits two values: foo, bar and then completes
const ob$ = Rx.Observable.of('foo', 'bar');

ob$.subscribe({
  next: x => console.log(x),
  error: err => console.error(err),
  complete: () => console.log('complete')
});

// foo, bar, complete
```

demo marbles

ob$.do()

For side effects, still needs subscribe to exec

```
const ob$ = Rx.Observable.of('foo', 'bar')
  .do(
    x => console.log(x),
    err => console.error(err),
    () => console.log('complete')
  );

ob$.subscribe();
// foo, bar, complete
```

demo

Observable.throw()

Create an observable which raises an error.

```
const ob$ = Rx.Observable.throw(new Error('my error'));

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// error my error
```

demo

Observable.from()

Create an observable from an array, promise, or another observable.

```
const prom = new Promise((resolve, reject) => {
  resolve('foo');
  /* or reject(new Error('my error')) */
});

const ob$ = Rx.Observable.from(prom);

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// next foo
// complete
```

demo marbles

Observable.interval()

```
const int$ = Rx.Observable.interval(1000);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});

// next 0
// next 1
// ...
```

demo marbles

ob$.take(N)

```
const int$ = Rx.Observable.interval(1000)
  .take(2);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0
// next 1
// complete
```

demo marbles

ob$.debounceTime()

```
const int$ = Rx.Observable.interval(100)
  .take(5)
  .debounceTime(200);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 4
// complete
```

demo marbles

ob$.throttleTime()

```
const int$ = Rx.Observable.interval(100)
  .take(5)
  .throttleTime(200);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0
// next 3
// complete
```

demo marbles

ob$.filter()

```
const int$ = Rx.Observable.interval(1000)
  .take(5)
  .filter(x => x % 2);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 1
// next 3
// complete
```

demo marbles

ob$.map()

```
const int$ = Rx.Observable.interval(1000)
  .take(5)
  .map(x => `${x} banana`);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0 banana
// next 1 banana
// next 2 banana
// next 3 banana
// next 4 banana
// complete
```

demo marbles

chaining

```
const int$ = Rx.Observable.interval(1000)
  .take(5)
  .filter(x => x % 2)
  .map(x => `${x} banana`);

int$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 1 banana
// next 3 banana
// complete
```

demo

Observable.merge()

```
const ob$ = Rx.Observable.merge(
  Rx.Observable.interval(1000)
    .map(x => `${x}s*****`),
  Rx.Observable.interval(200)
);

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next 0
// next 1
// next 2
// next 3
// next 0s*****
// next 4
// ...
```

demo marbles

.combineLatest()

```
const a$ = Rx.Observable.interval(2000).map(x => `${x}s`);
const b$ = Rx.Observable.interval(1200);
const ob$ = Rx.Observable.combineLatest(
  a$,
  b$,
  (a, b) => ({
    a: a,
    b: b
  })
);

ob$.subscribe(x => console.log('next', x.a, x.b));
// next 0s 3
// next 0s 4
// next 0s 5
// next 0s 6
```

demo marbles

ob$.catch()

```
const ob$ = Rx.Observable.throw(new Error('my error'))
  .catch(err => Rx.Observable.of({ type: 'UNCAUGHT',
    payload: err,
    error: true }));

ob$.subscribe({
  next: x => console.log('next', x),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next Object: {type: 'UNCAUGHT', payload: 'Error: my error at...'}
// complete
```

demo

Observable.ajax

```
const ob$ = Rx.Observable.ajax.getJSON('https://reqres.in/api/users')
  .map(payload => payload.data); /* use data prop */

ob$.subscribe({
  next: x => console.log('next', JSON.stringify(x, null, 2)),
  error: err => console.log('error', err),
  complete: () => console.log('complete')
});
// next [{ id: 1, first_name: 'george'...
//   { id: 2, first_name: 'lucille'...
//   ...
// complete
```

demo

ob$.mergeMap

```
Rx.Observable.of('redux', 'rxjs')
  .mergeMap(x => Rx.Observable.ajax({
      url: `https:npmsearch.com/query?q=${x}&fields=name,description`,
      crossDomain: true,
      responseType: 'json'
  })
  .map(ret => ret.response.results))  /* use results prop of payload */
  .subscribe({
    next: x => console.log('next', JSON.stringify(x, null, 2)),
    error: err => console.log('error', err),
    complete: () => console.log('complete')
  });
// next [{ name: 'react-redux-confirm-modal' ... ]
// next [{ name: 'rxjs-serial-subscription' ... ]
// complete
```

demo

Other functional approaches

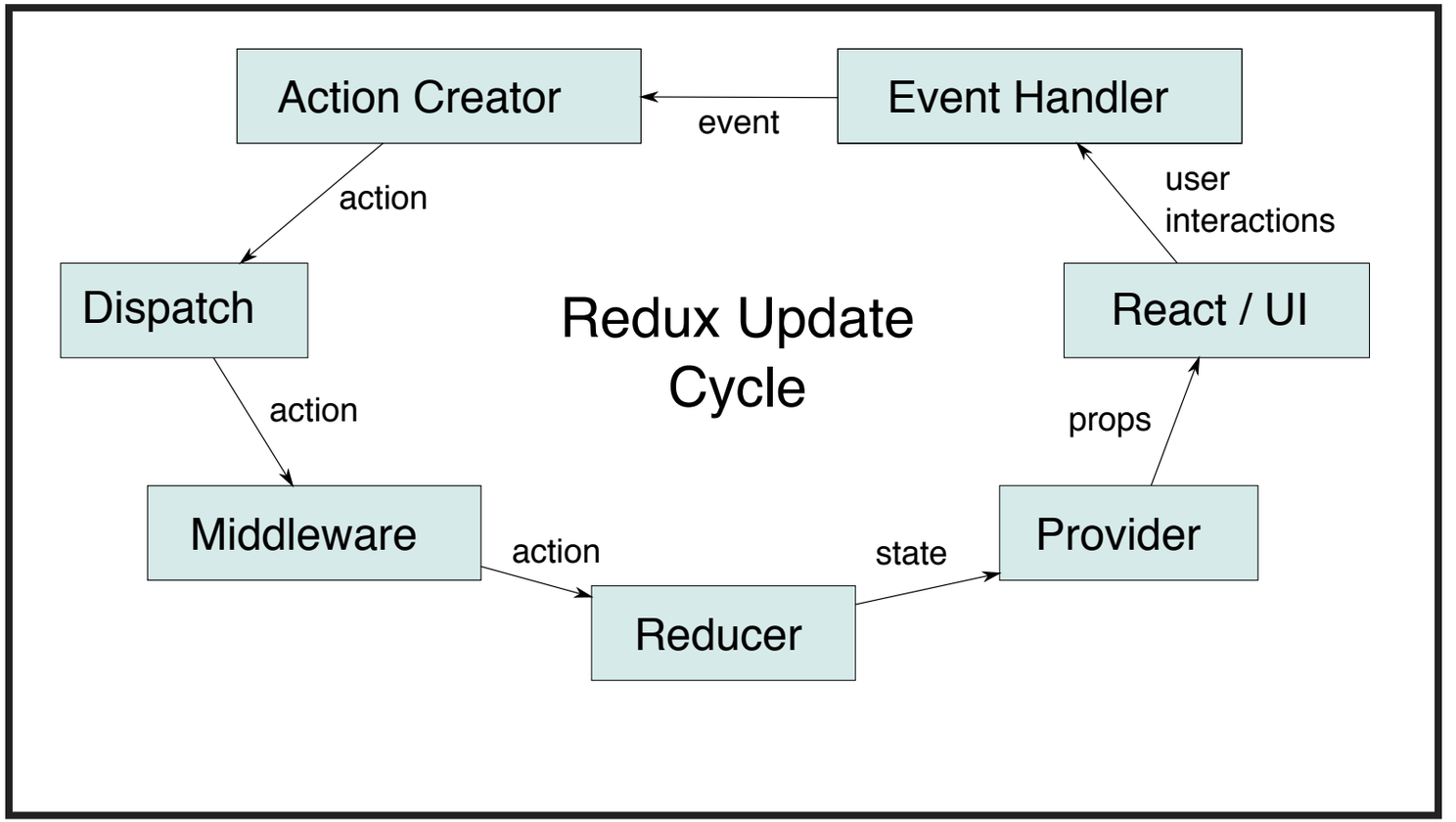
- futures are like promises but they don't run until you tell them to
- most.js - monadic streams https://github.com/cujojs/most
  - interop/convert to and from ES Observables and promises
  - performant
  - compatible with fantasy land spec
- monads - the grandparent of much of this work
  - fantasy land spec - https://github.com/fantasyland/fantasy-land/
    - compatible implementations - implementations.md in repo
  - monet.js - https://monet.github.io/monet.js/
  - folktale - http://folktalejs.org/

Where can we implement business logic in react/redux?

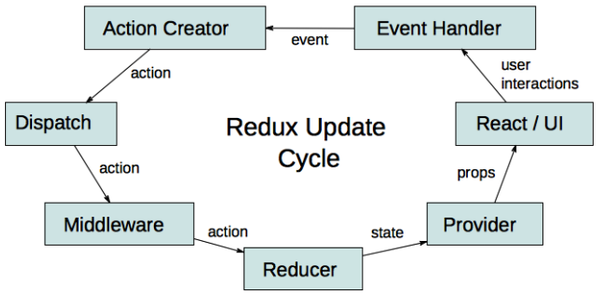
Goals for our business logic

- 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?

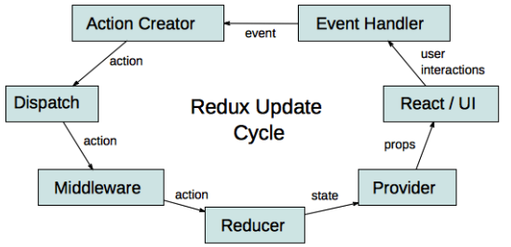


Reducer




```
function r(state, action) {
    // calc new state
    return newState;
}
```

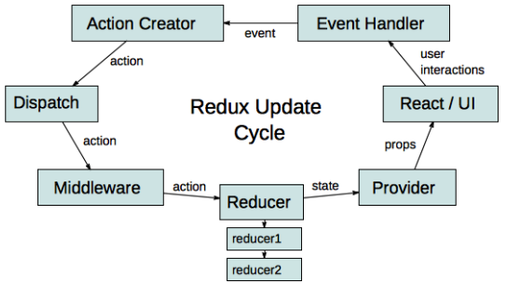
Reducer 2

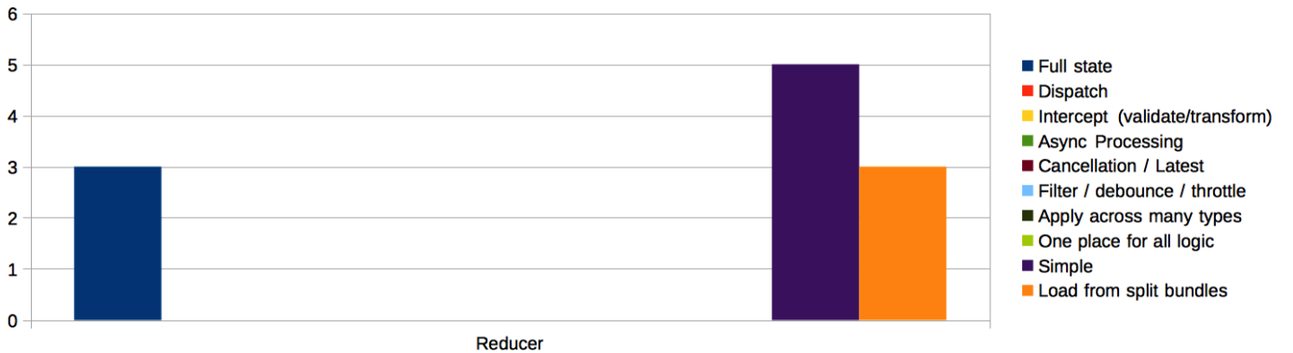



```
// full state
const state = {
    foo: {…},
    bar: {...}
};
```

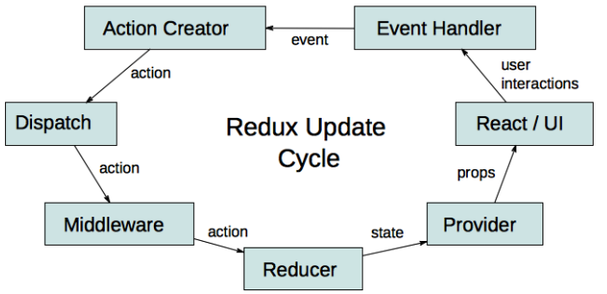
```
const reducer =
    combineReducers({
        foo: fooRed,
        bar: barRed
    });

function fooRed(state, action) {
    // calc new state
    return newState;
}
```

Reducer 3


Action Creator
event
Event Handler
action
user interactions
Dispatch
Redux Update Cycle
React / UI
action
props
Middleware
action
Reducer
state
Provider
reducer1
reducer2

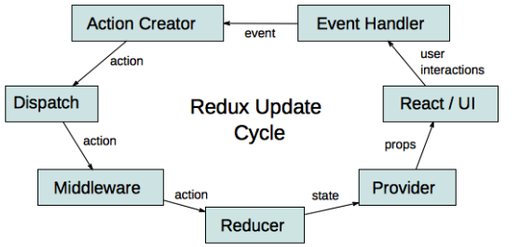

6
5
4
3
2
1
0
Reducer
Full state
Dispatch
Intercept (validate/transform)
Async Processing
Cancellation / Latest
Filter / debounce / throttle
Apply across many types
One place for all logic
Simple
Load from split bundles

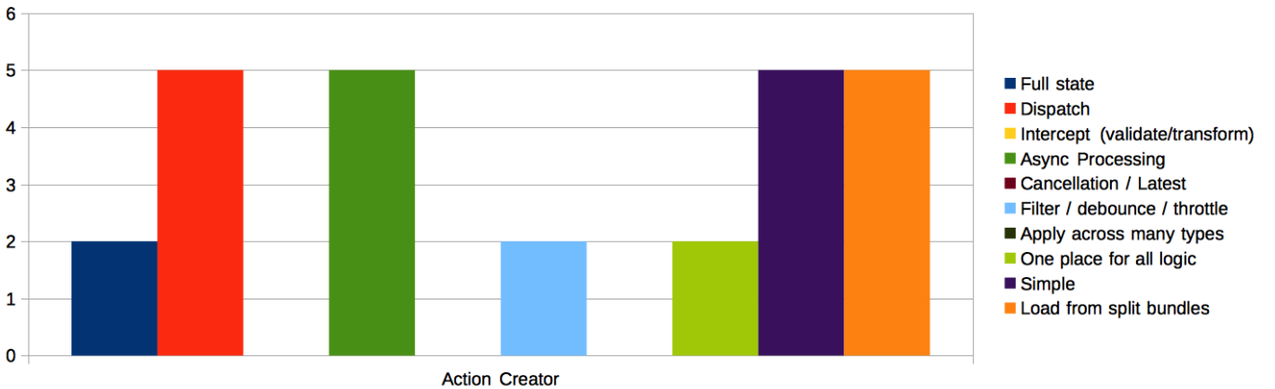
Action Creator



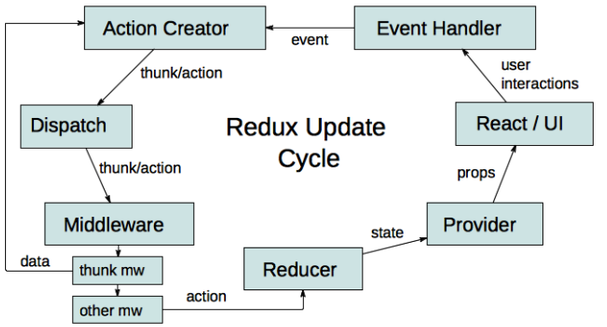

```
// thin action creator
function buyClicked(ev, a, b) {
  return { type: BUY };
}

// fat action creator
function buyClicked(dispatch, ev) {
  // sync or async code here
  dispatch({ type: BUY });
}
```

Action Creator 2



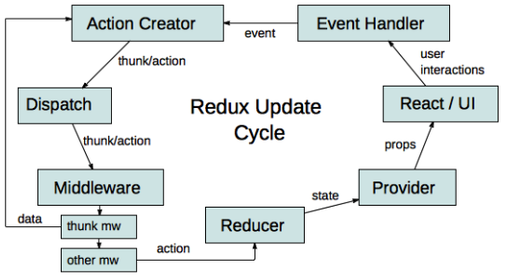


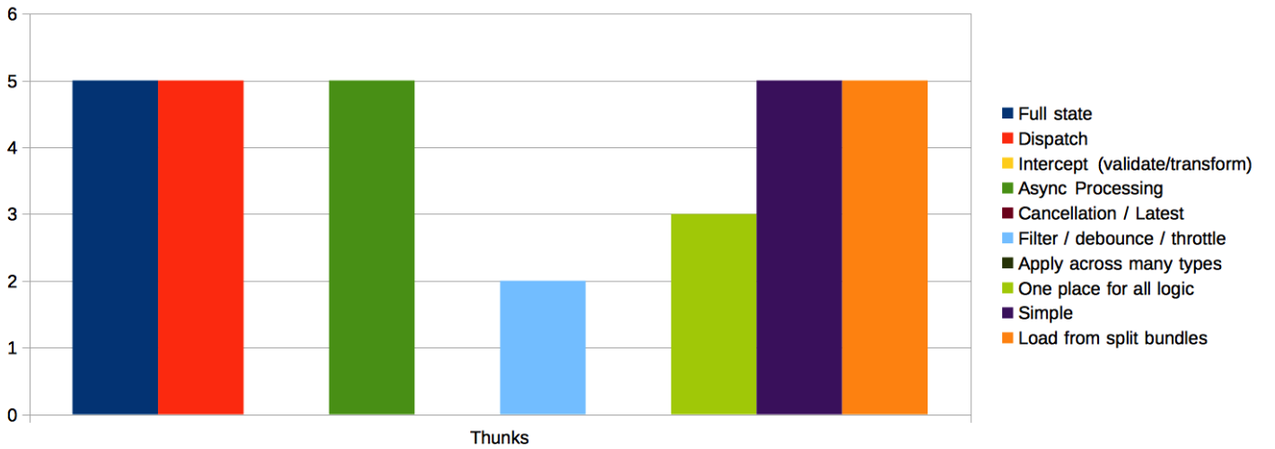
Thunks



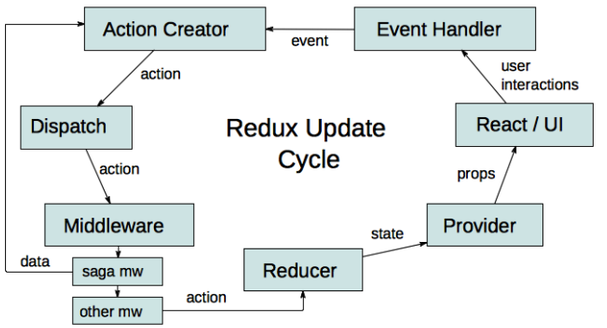

```
function buyClicked(ev, a, b) {
  return function(dispatch, getState) {
    // sync or async code here
    return dispatch({ type: BUY });
  };
}
```

Thunks 2


Action Creator
event
Event Handler
thunk/action
user interactions
Dispatch
Redux Update Cycle
React / UI
thunk/action
props
Middleware
state
Provider
data
thunk mw
Reducer
action
other mw

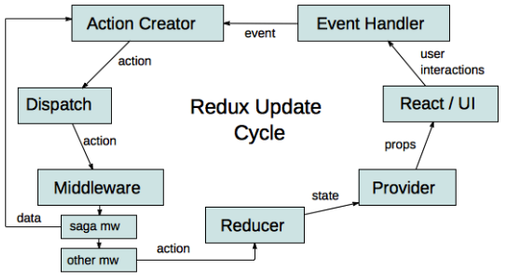

6
5
4
3
2
1
0
Thunks
Full state
Dispatch
Intercept (validate/transform)
Async Processing
Cancellation / Latest
Filter / debounce / throttle
Apply across many types
One place for all logic
Simple
Load from split bundles

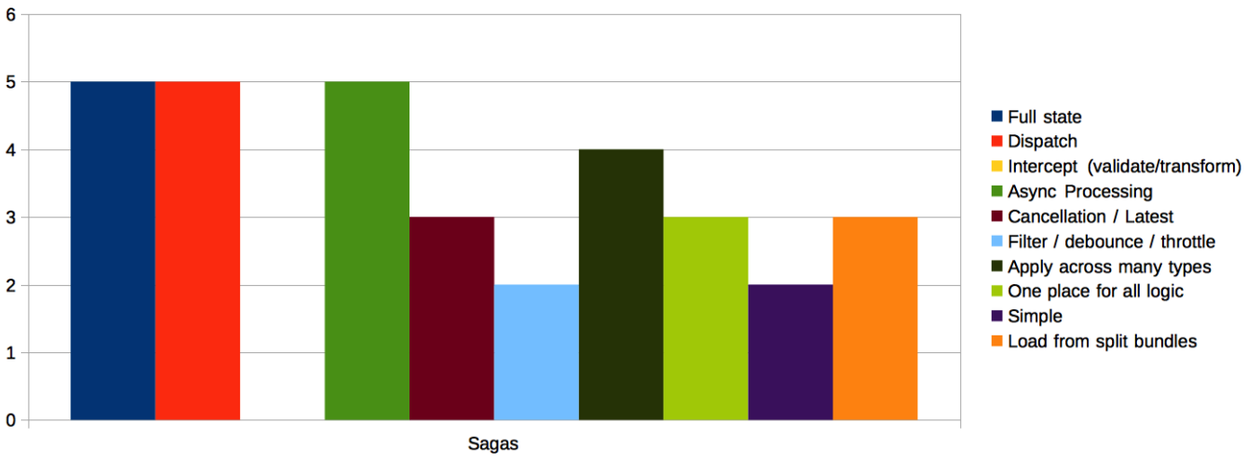
Sagas



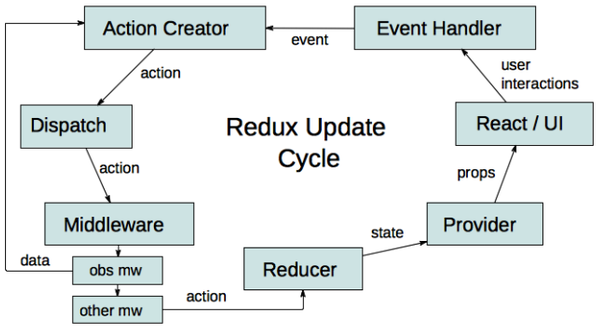

```
function* fetchUser(action) {
  try {
    const user = yield call(Api.fetchUser,
                  action.payload.userId);
    yield put({ type: "USER_FETCH_SUCCEEDED",
            user: user });
  } catch (e) {
    yield put({ type: "USER_FETCH_FAILED",
            message: e.message });
  }
}

function* mySaga() {
 yield* takeLatest("USER_FETCH_REQUESTED",
              fetchUser);
}
```

Sagas 2


Action Creator
event
Event Handler
action
user interactions
Dispatch
Redux Update Cycle
React / UI
action
props
Middleware
state
Provider
data
saga mw
Reducer
other mw
action

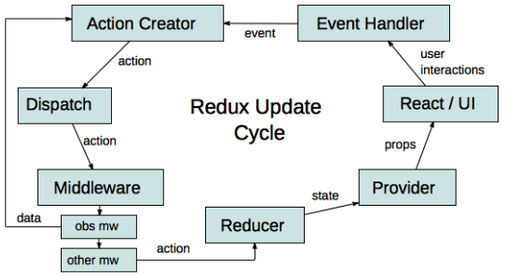

6
5
4
3
2
1
0
Sagas
Full state
Dispatch
Intercept (validate/transform)
Async Processing
Cancellation / Latest
Filter / debounce / throttle
Apply across many types
One place for all logic
Simple
Load from split bundles

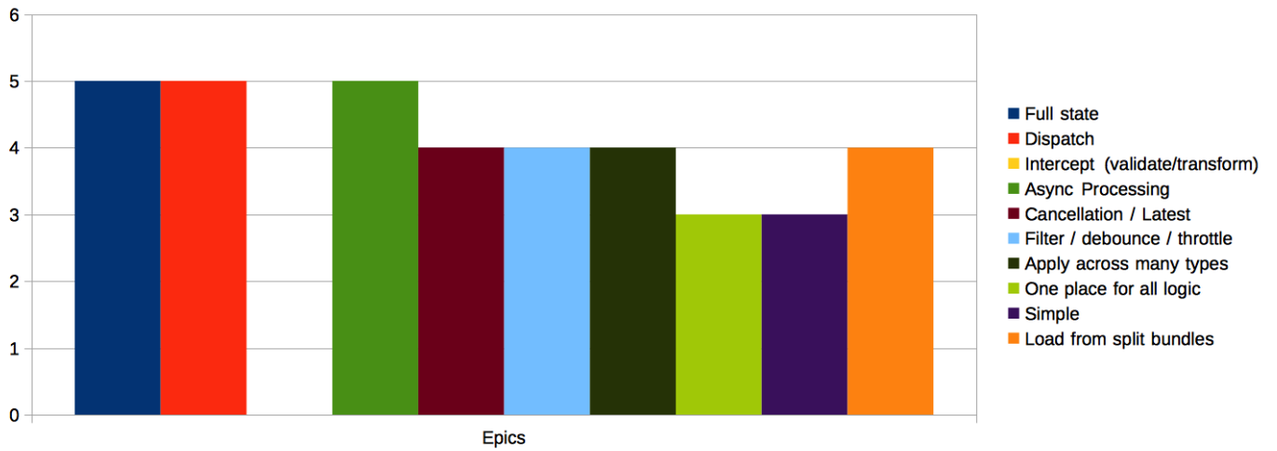
Epics – redux-observable



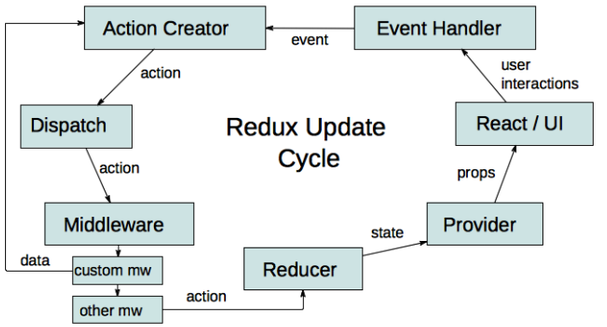

```
const fetchUserEpic = action$ => {
  const URL = `/api/users/${action.payload}`;
  return action$.ofType(FETCH_USER)
    .switchMap(action =>
      ajax.getJSON(URL)
        .map(fetchUserFulfilled)
        .takeUntil(
          action$.ofType(
            FETCH_USER_CANCELLED))
        .catch(err =>
          Observable.of({
            type: FETCH_ERROR,
            err }));
    );
};
```

Epics redux-observable 2



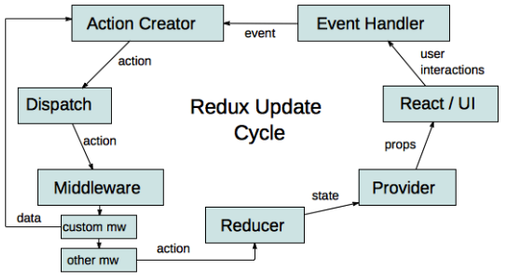


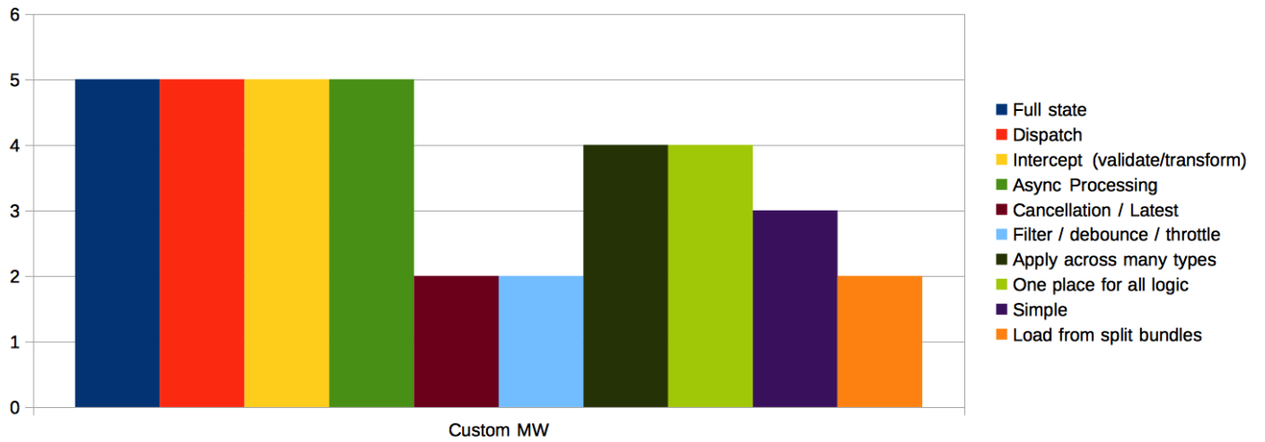
Custom Middleware



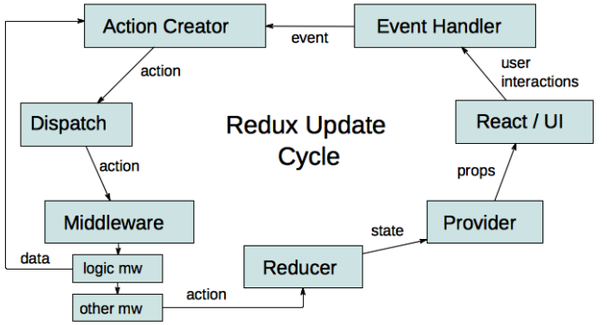

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const fetchUser = store => next => action => {
  if (action.type !== 'FETCH_USER') {
    return next(action);
  }
  // could modify, silence, log action
  let result = next(action);
  const state = store.getState();
  const { dispatch } = store;
  return fetch(url)
    .then(response => response.json())
    .then(user => dispatch({ type: USER_SUCCESS, user }))
    .catch(err => {
      console.error(err); // might be render err
      dispatch({ type: USER_ERROR, err });
    });
}
```

Custom Middleware 2



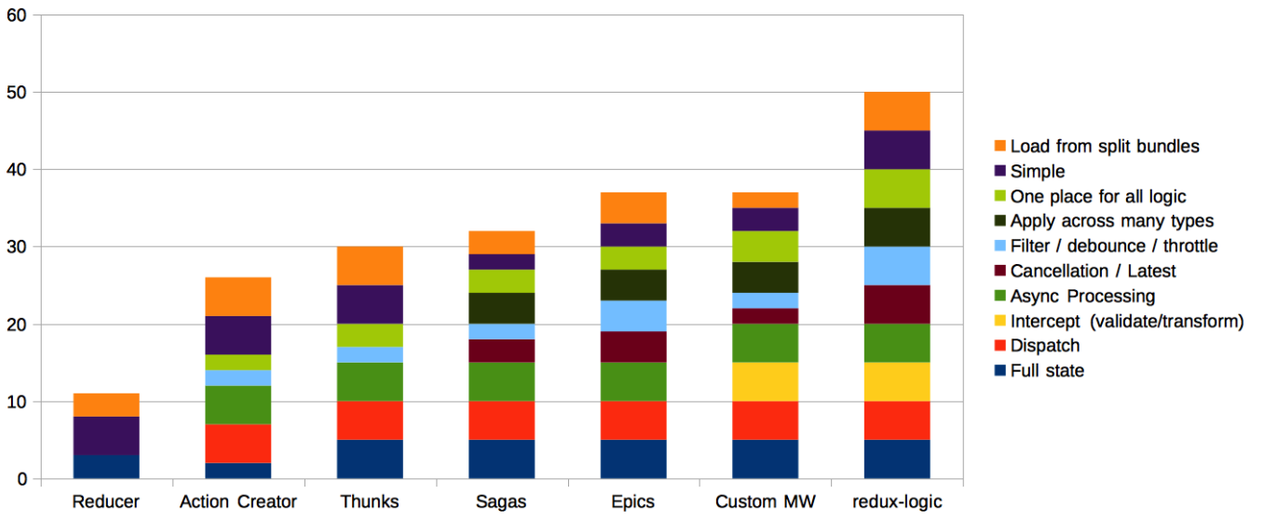


redux-logic




```
const logger = createLogic({
  type: '*',

  transform({ getState, action }, next) {
    console.log('dispatching', action)
    next(action);
    console.log('next state', getState())
  }
});

const fetchUser = createLogic({
  type: FETCH_POLLS,
  cancelType: FETCH_POLLS_CANCEL,
  latest: true,

  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); // could be render err
          dispatch({ type: FETCH_POLLS_FAILED,
                     payload: err,
                     error: true })
        .then(() => done()); // call when done dispatching
    });
  }
});
```

Unscientifc Results :-)



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);
  }
});
```

My current preferences

- 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

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 (my fav) 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
};
```

redux-logic-test - redux / redux-logic test helper

```
import { createMockStore } from 'redux-logic-test';

// specify as much as necessary for your particular test
const store = createMockStore({
  initialState: optionalObject,
  reducer: optionalFn, // default: identity reducer
  logic: optionalLogic, // default: []
  injectedDeps: optionalObject, // default {}
  middleware: optionalArr // other mw, exclude logicMiddleware
});

store.dispatch(...) // use as necessary for your test

// when all inflight logic has all completed calls fn + returns promise
  store.whenComplete(fn) - shorthand for store.logicMiddleware.whenComplete(fn)

store.actions - the actions dispatched, use store.resetActions() to clear
store.resetActions() - clear store.actions

// access the logicMiddleware created for logic/injectedDeps props
// use addLogic, mergeNewLogic, replaceLogic, whenComplete, monitor$
store.logicMiddleware
```

Summary

- Discussed types of business logic
- Ways of orchestrating async tasks and I/O
  - promises, async/await, observables, most, ...
- Places to implement business logic in react/redux and the tradeoffs

- Middleware - thunks, observables, sagas, custom, redux-logic
- Testing Tools - jest, mocha, expect, enzyme, nightmarejs, headless chrome, redux-logic-test

Thanks

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