Where do I put my business logic in a React-Redux application?

Jeff Barczewski
Aug 3, 2016 · 14 min read

This is a question that we have all struggled with in building applications. It is an important question because it influences the architecture of our code and how well our app will absorb new features and complexity as it naturally grows during its lifetime.

TL;DR (Summary)

There are many approaches that will work but it is important to understand the tradeoffs to pick the best solution. I discuss some of the key highlights for each of these options.

Finally I propose a new library, **redux-logic**, which combines power and flexibility with simplicity. It is a hybrid of common declarative functionality (like cancellation) that wraps your imperative laser focused business code.

Approaches for implementing business logic that I cover:

- action creators
- reducers
- thunks
- sagas
- epics
- effects
- custom middleware
- redux-logic — a new approach

The journey to enlightenment

Here is my journey to trying to find the best approach to structuring business logic in my React + Redux app.

What do I mean by business logic?

According to wikipedia, “In computer software, business logic or domain logic is the part of the program that encodes the real-world business rules that determine how data can be created, displayed, stored, and changed. It is contrasted with the remainder of the software that might be concerned with lower-level details of managing a database or displaying the user interface, system infrastructure, or generally connecting various parts of the program.”

This is the heart of your application. Given the current state of your application it governs what a user is allowed to do, what is allowed to happen next, and what happens when they attempt to make changes.

In some apps this might mean validation, verification, authorization. It might mean denying certain invalid attempts to change state or prevent illegal operations. Your

business logic governs the user's UI and helping them to understand actions that are appropriate at any given time.

It governs what happens when data is updated. Much of this might occur behind the scenes on a server, but in today's world more and more is being done client-side and even offline.

It also might govern when and where data is fetched from or sent to a server. This work needs to be managed asynchronously and dealt with as the data or response returns. Depending on the state of the application it will determine whether to use it (it may have been cancelled or outdated due to a more recent request).

So it's really the substance that makes your application unique. There are many e-commerce sites out there, but the specifics of what happens when you use the site and move through the purchase cycle is the business logic that makes each special.

Most business logic could probably be categorized into these three main categories:

- validation, verification, authorization
- transformation, augmentation, applying defaults
- processing — asynchronous orchestration

Note that you can also model your action flow such that you don't need to intercept actions for validation, verification, and auth. You would instead listen for end user actions, perform the validation, auth, etc then dispatch different reducer actions to actually make the state changes or if a there was an error, you dispatch error actions instead. This is a workable solution but it increases the number of actions and handlers you are dealing with. I think it is nice to have the ability to intercept actions directly if you desire to do so.

Dispatching from action creators

The first obvious choice when considering where to put business logic is simply to try to perform all of the work in an action creator. You can simply perform your work and then dispatch when you have results.

```
const fetchUser = (dispatch, id) => {
  dispatch({ type: USER_FETCH, payload: id });
  axios.get(`https://server/user/${id}`)
```

```
      .then(resp => resp.data)
      .then(user => dispatch({ type: USER_FETCH_SUCCESS,
                               payload: user }))
      .catch(err => {
        console.error(err); // log since might be a render err
        dispatch({ type: USER_FETCH_FAILED,
                               payload: err,
                               error: true });
      });
  };
```

This seems pretty reasonable. It's simple and the code is focused on performing the user fetching and dispatching of the result or error.

Advantages:

- simple and focused code, not much additional ceremony

Limitations:

- No easy way to cancel pending requests or only take the latest if multiple requests are made
- No access to the global state to make decisions with. You only have access to whatever props or component state that you pass in. This might mean requiring extra data to be provided to a component that normally wouldn't be needed for rendering but is solely needed for the business logic. Any time the business logic changes it might require new additional data to be passed down possibly affecting many components in the process.
- No interception. I also have no easy way to create business logic that applies across many actions, so if I wanted to augment all actions to have a timestamp or unique ID, I'd have to include a call to that code in all action creators.
- Testing components and fat action creators may require running the code (possibly mocked).

So this was a good start. Many people can get by pretty well with this choice.

Reducers

Another approach that only works for synchronous business logic is to simply perform it in your reducers.

So in addition to updating the state, the reducer might first decide what it is going to do by looking at the current state.

Reducers operate synchronously so they can't directly initiate asynchronous behavior. One variation on this is using redux-loop to initiate additional async work. That is discussed later.

Advantages:

- It's easy to perform some logic inside of the reducer
- Business logic can choose whether to apply the state change

Disadvantages:

- Conflates the business logic with the state updates
- Only synchronous tasks
- No ability to replace or dispatch a different error action
- Access is limited to partial state if reducers are being combined
- No global interception of actions

Thunks — functions and promise

The next suggestion is to use one of the variety of thunk middleware available. You to pass a function or promise from your action creator that is intercepted by a special thunk middleware and executed.

```
const fetchUser = id => (dispatch, getState) => {
  dispatch({ type: USER_FETCH, payload: id });
  axios.get(`https://server/user/${id}`)
    .then(resp => resp.data)
    .then(user => dispatch({ type: USER_FETCH_SUCCESS,
                             payload: user }))
    .catch(err => {
      console.error(err); // log since might be a render err
      dispatch({ type: USER_FETCH_FAILED,
                              payload: err,
                              error: true });
    });
};
```

This is an improvement since now we have access to the global state. The thunk middleware will give us the getState method when it runs the function.

However it still has the same limitations of being non-cancelable and no interception.

Advantages:

- Simple and focused
- Access global state easily
- Slightly simpler redux mapDispatchToProps for binding action creators

Limitations:

- No easy way to cancel pending requests or only take the latest if multiple requests are made
- No interception. I also have no easy way to create business logic that applies across many actions, so if I wanted to augment all actions to have a timestamp or unique ID, I'd have to include a call to that code in all action creators.
- Testing components and thunked action creators may require running the code (possibly mocked). When you have a thunk (function or promise) you don't know what it does unless you execute it.

Promise thunks are similar in that instead of returning a function you return a promise.

Sagas — redux-saga

Sagas introduce a way to use ES6 generators to run additional business logic and async code. Your action creators simply trigger the saga by dispatching a simple action that the saga is listening for.

The triggering action is recognized by the sagaMiddleware, it lets the message continue to the reducers, then the particular saga which is listening for this action wakes up and begins to run the logic.

```
// this action creator will be bound to dispatch
const fetchUser = id => ({ type: USER_FETCH, payload: id });

// saga code elsewhere
function* watchUserFetch() {
```

```
 yield* takeLatest("USER_FETCH", fetchUser);
}

function* fetchUser(action) {
 try {
   const user =
     yield axios.get(`http://server/user/${action.payload}`);
   yield put({ type: USER_FETCH_SUCCESS, payload: user.data });
 }
 catch(err) {
   yield put({ type: USER_FETCH_FAILED, payload: err, error: true
});
 }
}

// rootSaga.js
export default function* rootSaga() {
 yield fork(watchUserFetch);
 yield fork(…);
 // etc
}
```

So Sagas bring some improvement to our situation in that now we can deal with limiting response to the most recent request.

We could also handle cancellation using code similar to the following.

```
// this action creator will be bound to dispatch
const fetchUser = id => ({ type: USER_FETCH, payload: id });

// saga code elsewhere
function* watchUserFetch() {
  while ( yield take(USER_FETCH) ) {
    const userFetchTask = yield(fork(fetchUser));

    // wait for cancel action
    yield take (USER_FETCH_CANCEL);

   // we have the cancel action, cancel task
   yield cancel(userFetchTask);
 }
}

function* fetchUser(action) {
 try {
   const user = yield fetch(`http://server/user/${action.payload}`);
   yield put({ type: USER_FETCH_SUCCESS, payload: user });
 }
 catch(err) {
   yield put({ type: USER_FETCH_FAILED,
               payload: err,
               error: true });
```

```
  }
  finally {
    if (yield cancelled()) { // if was cancelled
      yield put({ type: USER_FETCH_CANCELLED });
    }
  }
}

// rootSaga.js
export default function* rootSaga() {
 yield fork(watchUserFetch);
 yield fork(…);
 // etc
}
```

So while this brings us additional power we didn't have before, the code has become more complex. It requires a good grasp of generators to use effectively.

Advantages:

- ES6 generators allow for limiting and cancellation
- Action creators now only dispatch simple objects so testing components is easy. Generator code can be tested independently.
- Asynchronous orchestration looks like synchronous code via yield

Limitations:

- ES6 generators are not commonly well understood by many developers
- Fair amount of code to setup watch loops and implement cancellation
- No interception — sagas always run after actions have been given to the reducers

Epics redux-observable

Epics are the redux-observable approach to Sagas. They use RxJS Observables to allow for filtering, limiting, cancellation, and asynchronous orchestration.

Each epic is given an incoming action observable and it returns an observable of actions to dispatch.

```
// this action creator will be bound to dispatch
const fetchUser = id => ({ type: USER_FETCH, payload: id });
```

```
// userEpics.js
const fetchUserEpic = action$ =>
  action$.ofType(FETCH_USER)
    .mergeMap(action =>
      Rx.Observable.create(obs => {
        axios.get(`https://server/user/${action.payload}`)
          .then(resp => resp.data)
          .then(user => {
            obs.next({ type: FETCH_USER_SUCCESS, payload: user });
            obs.complete();
          })
          .catch(err => {
            obs.next({ type: FETCH_USER_FAILED,
                       payload: err,
                       error: true });
            obs.complete();
          });
      })
      .takeUntil(action$.ofType(USER_FETCH_CANCELLED)));

// rootEpic.js — export rootEpic used with createEpicMiddleware()
export default combineEpics(
 fetchUserEpic,
 fooEpics
);
```

Alternatively if you instead use the RxJS ajax utilities you can eliminate the create observable code. (import { ajax } from ‘rxjs/observable/dom/ajax’)

```
// this action creator will be bound to dispatch
const fetchUser = id => (
  { type: USER_FETCH, payload: id }
);

// userEpics.js
const fetchUserEpic = action$ =>
  action$.ofType(FETCH_USER)
    .mergeMap(action =>
      ajax.getJSON(`https://a/${action.payload}`)
        .map(user => ({ type: FETCH_USER_SUCCESS,
                        payload: user }))
        .catch(err => ({ type: FETCH_USER_FAILED,
                         payload: err,
                         error: true }))
        .takeUntil(
          action$
            .ofType(USER_FETCH_CANCELLED)));

// rootEpic.js - export rootEpic used with the
createEpicMiddleware()
export default combineEpics(
  fetchUserEpic,
```

```
  fooEpics
);
```

So Epics bring about some nice functionality allowing for easily cancellation, throttling, debouncing, taking the latest.

However it's not all roses though since now instead of ES6 generators you need to understand RxJS observables well. I like the power of observables but they can be a bit of a challenge to master, there are a variety of edge cases you can run into.

Advantages:

- Observables bring the power of cancellation and limiting

Limitations:

- RxJS Observables can be difficult to learn and grasp.
- A good portion of the code is now there simply for setting up the observable behavior
- No interception — actions are sent through reducers first before being handed to epics
- Testing can require a bit of setup

Effects — redux-loop

redux-loop is a way to deal with side effects using an elm inspired approach. The idea is that your reducer can return a structure of effects to run, then the redux-loop middleware runs it and dispatches the results.

```
const fetchUser = id =>
  axios.get(`https://server/user/${id}`)
    .then(resp => resp.data)
    .then(user => ({ type: USER_FETCH_SUCCESS, payload: user }))
    .catch(err => ({ type: USER_FETCH_FAILED,
                     payload: err,
                     error: true }));

function reducer(state, action) {
  switch(action.type) {
    case USER_FETCH :
      return loop(
        state.set('status', 'fetching'),
```

```
        Effects.promise(fetchUser, action.payload)
      );
  }
}
```

Advantages:

- Effect code is pretty clean, not much extra ceremony
- Everything is fairly testable in isolation since the effects are returned as structures

Limitations:

- Reducers no longer return just state but possibly effects
- No cancellation or limiting functionality
- No global interception of actions

Custom Middleware

Custom middleware can pretty much do anything and that's what we'd expect that since that is what all of these other solutions are built on.

If you implement custom middleware you can intercept all actions, transforming as necessary, and perform any number of async operations and dispatching.

```
const fetchUserMiddleware = store => next => action => {
  if (action.type === FETCH_USER) {
    axios.get(`https://server/user/${action.payload}`)
      .then(resp => resp.data)
      .then(user => store.dispatch({ type: USER_FETCH_SUCCESS,
                                     payload: user }))
      .catch(err => {
        console.error(err); // log since might be a render err
        store.dispatch({ type: USER_FETCH_FAILED,
                                     payload: err,
                                     error: true });

      });
     return next(action); // pass the original
  } else { // pass other actions
    return next(action);
  }
};
```

So you have full power to do anything here, but unfortunately you have to remember to deal with all actions and it is up to you to implement your own cancellation, throttling, debouncing, dynamic loading of code, etc.

Advantages:

- Full power

Limitations:

- All custom code — you implement all functionality
- No safety net, if your code errors, it could stop all future actions
- Remember to pass on actions that you aren't concerned with
- Need to mock things to test your code

redux-logic — A new approach

After learning all of these different ways, I still wasn't happy with finding one approach that I could use for all of my business logic.

Epics were pretty nice for async code, but observables can be tricky to understand. Sagas look interesting but generator code is also challenging.

So here is what I really wanted:

- Declarative functionality for cancellation and limiting. The power of observables without the need to write observable code. Hide that complexity from the user and just make it configurable.
- Ability to intercept some or all actions to provide validation, verification, authorization, and transformation
- Ability to do async processing and dispatching
- Ability to create long running cancelable subscriptions for streaming updates
- Little to no ceremony — code I write should be focused on performing business logic
- Easy testing in isolation — be able to run my code without a bunch of hassle or setup

After pondering this for a bit, I realized that this wouldn't be that hard to build. I simply need to be able to configure a few behaviors and my middleware would invoke the necessary RxJS observable code to enable that behavior. Then I could simply provide a simple interface to the business logic code.

> *"I wrote the RxJS code so you wouldn't have to"*

If I organize this in the middleware properly I can perform interception of actions before they hit the reducers as well as perform async processing and dispatching after.

After a little bit of experimenting with the API possibilities, I settled on the following API. A small amount of behavioral functionality can be configured to enable cancellation and limiting, etc. Next I created simple execution phase hooks, that your business logic could tap into based on what you need to accomplish. That way it would be easy to perform validation, transformation, and/or processing. You can define one or more of those hooks in your logic based on what you are trying to do.

```
// bound to dispatch
const fetchUserAction = id => ({ type: USER_FETCH, payload: id });

// in userLogic.js
const fetchUserLogic = createLogic({
  // declarative behavior
  type: USER_FETCH, // filter for actions of this type
  cancelType: USER_FETCH_CANCEL, // cancel on this type
  latest: true, // only provide the latest if fired many times

// execution phase hooks: validate, transform, process
 // implement one or more of these
 process({ getState, action }, dispatch, done) {
   axios.get(`https://server/user/${action.payload}`)
     .then(resp => resp.data)
     .then(user => dispatch({ type: USER_FETCH_SUCCESS,
                              payload: user }))
     .catch(err => {
       console.error(err); // log since might be a render err
       dispatch({ type: USER_FETCH_FAILED,
                              payload: err,
                              error: true });
     })
     .then(() => done()); // call when done dispatching

  }
});
```

Another way to write this same code but using the new `processOptions` feature in redux-logic@0.8.2+ which cleans up the code further.

```
const fetchUserLogic = createLogic({
  // declarative behavior
  type: USER_FETCH,  // filter for actions of this type
  cancelType: USER_FETCH_CANCEL, // cancel if action is dispatched
  latest: true, // only provide the latest if fired many times

  processOptions: { // options for process hook, default {}
    dispatchReturn: true // dispatch from resolved/rejected promise
    successType: USER_FETCH_SUCCESS,  // use action type for success
    failType: USER_FETCH_FAILED       // use action type for errors
  },

  // No need to dispatch since we are using the returned promise
  // and automatically applying the actions to the raw values which
  // get mapped to the action payload
  process({ getState, action }) {
    return axios.get(`https://server/user/${action.payload}`)
      .then(resp => resp.data);
  }
});
```

That's it. A small amount of declaration to get the advanced cancellation and take latest behavior, with concise business logic as code.

I'm leveraging the power of RxJS observables without making developers write code with them. I can use whatever type of code I am familiar with in writing my business logic. I am of course free to use observables if I want, but I can focus on my main purpose. You may even dispatch an observable to create a long running subscription. It is entirely up to you, if you want to use callbacks, promises, async await, etc., use what you are comfortable with for your code.

If I want to write validation, verification, authorization type code then I can use the validate execution phase hook and I have full access to the global state to make decisions for validation and verification.

```
const userVerifyLogic = createLogic({
  type: [USER_ADD, USER_UPDATE] // filter to only these types
  validate({ getState, action }, allow, reject) {
    const user = action.payload;
    const state = getState();
    // can perform checks on anything in global state
    if (!user.name) {
```

```
        reject({ type: USER_ERROR,
                 payload: new Error('missing name'),
                 error: true });
      } else { // valid user
        allow(action);
      }
    }
  });
```

Another example is if we want to add timestamp and unique ID to every action, this time I will use the transform execution phase hook.

```
const addTimestampAndUniqueIDLogic = createLogic({
  type: '*', // apply to all actions
  transform({ getState, action }, next) {
    const existingMeta = action.meta || {};
    const meta = {
      …existingMeta,
      timestamp: Date.now(),
      uniqueId: shortId.generate()
    };
    // now simply call next with the augmented action
    next({
      …action,
      meta
    });
  }
});
```

The full redux-logic API

```
const fooLogic = createLogic({
  // filtering/canceling
  type, // required str, regex, array of str/regex, use '*' for all
  cancelType, // string, regex, array of strings or regexes

  // limiting - optionally define one of these
  latest, // only take latest, default false
  debounce, // debounce for N ms, default 0
  throttle, // throttle for N ms, default 0

  // Put your business logic into one or more of these
  // execution phase hooks.
  //
  // Note: If you provided any optional dependencies in your
  // createLogicMiddleware call, then these will be provided to
  // your code in the first argument along with getState and action
  validate({ getState, action }, allow, reject) {
    // run your verification logic and then call allow or reject
    // with the action to pass along. You may pass original action
```

```
    // with the action to pass along. You may pass original action
    // or a modified/different action. Use undefined to prevent any
    // action from being propagated like allow() or reject()
    allow(action); // OR reject(action)
  }),

  // alias for the validate hook
  transform({ getState, action }, next) {
    // perform any transformation and provide the new action to next
    next(action);
  }),

  // options influencing the process hook, defaults to {}
  processOptions: {
    // dispatch resolved/rejected promise/observable from return
    dispatchReturn: false, // default false
    // string or action creator fn wrapping dispatched value
    successType: undefined, // default undefined
    // string or action creator wrapping rejected or thrown errors
    failType: undefined // default undefined
  },



  // perform async processing and dispatching
  process({ getState, action, cancelled$ }, dispatch, done) {
    // Perform your processing then call dispatch with an action
    // then call done() when finished dispatching
    // See other ways to use process in advanced API docs
    dispatch(myNewAction);
    done(); // call when done dispatching
  })
});

const logicMiddleware = createLogicMiddleware(
  logic, // array of logic items
  deps   // optional injected deps/config, supplied to logic
);

// dynamically add logic later at runtime, keeping logic state
logicMiddleware.addLogic(newLogic);

// replacing logic, logic state is reset but in-flight logic
// should still complete properly
logicMiddleware.replaceLogic(replacementLogic);
```

So that's redux-logic in a nutshell.

I'm excited about how this unifies and simplifies business logic code letting us focus on the important details.

I'd love to hear what you think, my twitter handle is @jeffbski or you can contact me through my training site: https://codewinds.com/

You can find the code, docs, and examples at redux-logic repo

Redux React Business Logic Logic Async