How to handle
asynchronous actions in ⟳ Redux

Jacek Tomaszewski
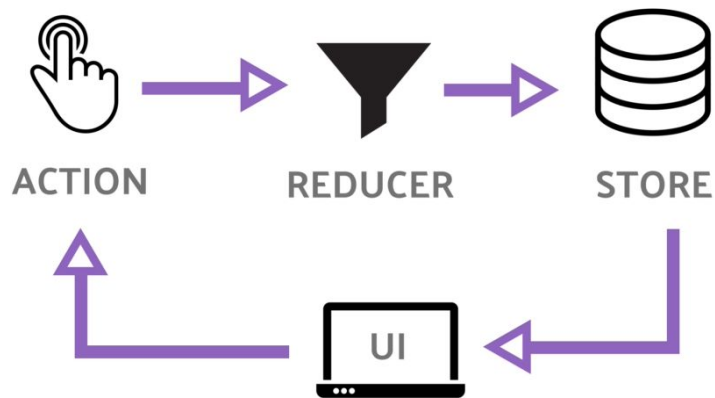
Fullstack Web Developer at Recruitee.com

Freelancer and Consultant at jtom.me

Redux is a beautiful concept

- One-way data flow

- Everything is pure and simple:

  - Actions and State are just immutable data objects

  - Reducer is a pure function, that has no side effects and produces a new state immediately

Redux is a beautiful concept

- It lets us grasp all UI events and other side effects into a controllable, testable data form

  - All of them produce and dispatch a Redux action payload,
    so they are no longer side-effects per se

- An app with multiple actions being done at the same time is no longer a problem

  - Redux applies all the actions consecutively thanks to the reducer function

... but the world is not synchronous

Not all of the actions can be handled immediately.

- Some of them will take time.
  They are asynchronous.

- Some of them need to be aborted when another action appears in the meantime.
  They are dependant on each other.

Redux leaves us to solve it on our own

> "It is up to you to try a few options, choose a convention you like, and follow it, whether with, or without the middleware."

Redux official documentation

Current approaches to the problem

An action creator that returns a function which will dispatch multiple actions over time

- redux-thunk
- redux-promise

A "4th" element in the Redux store, that listens to the actions, acts on them, and dispatches another actions in the background as a result

- using JS generators syntax and custom APIs
  - redux-saga
  - redux-ship

- using Observables
  - redux-effects
  - redux-cycles

Solution no 1: Allow action creators to dispatch multiple actions over time

1. An instiantated action A0 emits a side effect (f.e. a fetch action that starts an API request)

2. After some time another side effect arrives (f.e. the API request had been finished)

3. The function that created the action A0, picks up the side effect, and creates an action A1 (f.e. a fetch success action)
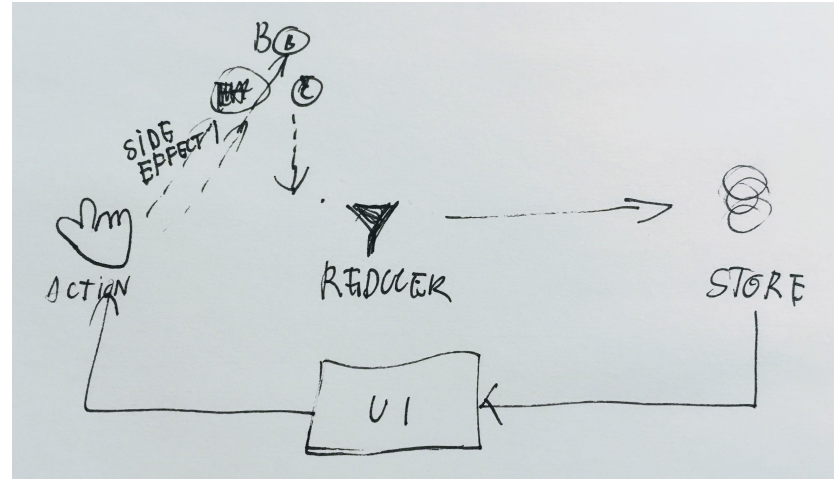
Allow action creators to dispatch multiple actions over time:
redux-thunk

```javascript
function fetchUser() {
 return dispatch => {
    dispatch({ type: FETCH_USER_START });
    UserService.getCurrentUser().then(
      user => {
        dispatch({ type:
FETCH_USER_SUCCESS, user });
      },
      error => {
        dispatch({ type:
FETCH_USER_FAILURE, error });
      }
    )
 }
}
```

Pros & cons of redux-thunk-like solution

+ Doesn't differ much from normal synchronous action creators

+ Easy and quick to write, barely any API that you need to learn

+ Is good enough in most simple cases

- The action creators, that were supposed to be simple, quickly get over bloated with business logic code

- It's still impossible (or at least veery hacky) to do stuff like debouncing the action or cancelling an ongoing one

Solution no 2a: Use JS generators to listen to actions, act on them and create another ones

1. State machine listens to an action A0 and switches to a state A1, and then to A2;

2. State machine waits for an action A3 to come;

3. State machine dispatches a new action into the Redux reducer function.
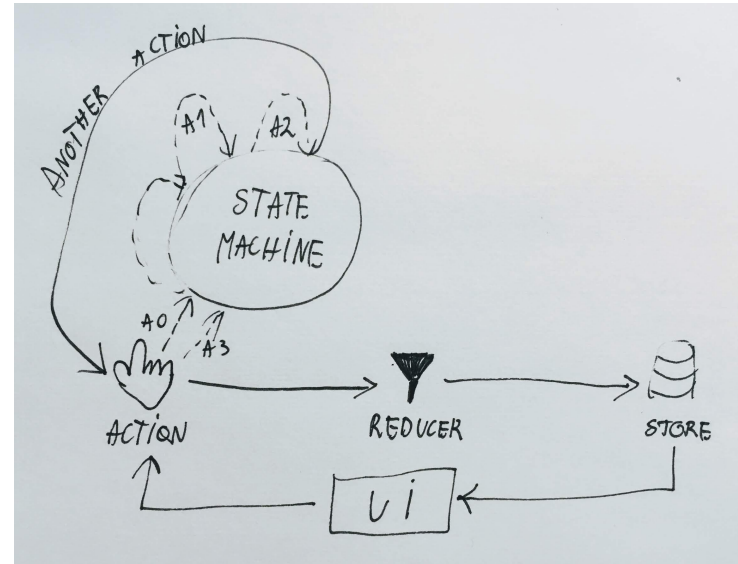
Use JS generators to listen to actions, act on them and create another ones:
redux-saga

```javascript
import { call, put, takeLatest } from
"redux-saga/effects";

function* fetchUser(action) {
 yield put({ type: FETCH_USER_START });
 try {
   const user = yield
call(UserService.getCurrentUser);
   yield put({ type: FETCH_USER_SUCCESS,
user: user });
 } catch (error) {
   yield put({ type: FETCH_USER_FAILURE,
message: error });
 }
}

function* fetchUserSaga() {
 yield takeLatest(FETCH_USER,
fetchUser);
}
```

Pros & cons of redux-saga-like solution

+ Acting on actions, initiating side effects and build actions from them is extracted to Sagas
+ You can wait for another action, cancel ongoing action, debounce, throttle, etc.
+ Is easily testable (you can just directly test a given saga, by putting in actions and expecting another ones)
+ Quite easy to learn (~1-2 days)

- You need to learn JS generators syntax
- You need to learn the custom library's API, which isn't used anywhere else (~around 10-40 functions that allow you to instruct the State machine about how and for what actions should it wait)
- Very difficult to make it work with TypeScript
- // Personally, I never liked the syntax too much (too imperative)

Solution no 2b: Use Observables to store, control and produce actions

1. Observable picks up an action A0

2. Observable waits a few seconds

3. Observable switches to do some other stuff (like an API request) and waits for the result

4. Observable waits for another action A1 to come

5. Observable finally results with another action, which is dispatched to the reducer
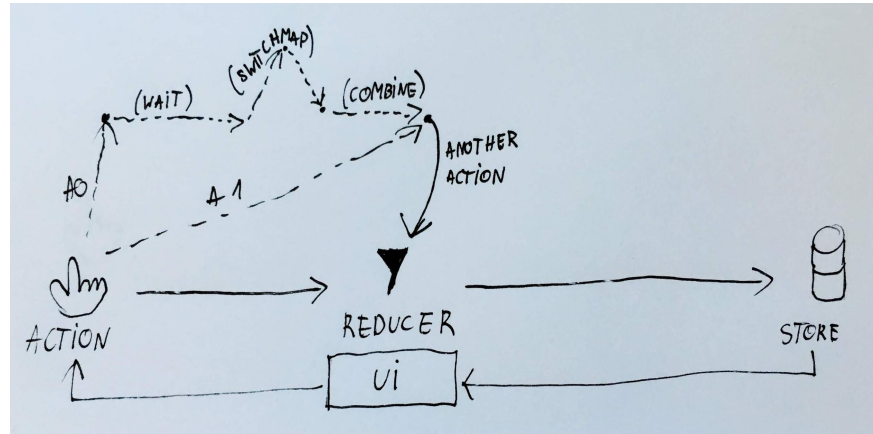
Use Observables to store, control and produce actions:
redux-effects

```javascript
const fetchUserEpic = action$ => {
 return action$
   .ofType(FETCH_USER)
   .switchMap(action => {
     return Observable.concat(
       Observable.of({ type:
FETCH_USER_START }),
       Observable.fromPromise(

UserService.getCurrentUser().then(
         user => ({ type:
FETCH_USER_SUCCESS, user }),
         error => ({ type:
FETCH_USER_FAILURE, error })
       )
     )
   );
 });
}
```

Use Observables to store, control and produce actions:
redux-effects

```javascript
const refetchResultsEpic = action$ => {
  return Observable.merge(
    action$.ofType(CHANGE_QUERY)
      .debounceTime(500),
    action$.ofType(CHANGE_FILTERS)
      .debounceTime(300),
    action$.ofType(CHANGE_PAGE),
    action$.ofType(CHANGE_SORT_BY),
  )
    .mapTo({ type: FETCH_RESULTS });
}
```

Use Observables to store, control and produce actions:
redux-effects

```javascript
const validateOnSubmitEpic = (action$,
store) =>
 action$.ofType(SUBMIT).switchMap(() =>
{
  const validate$ = Observable.of({
type: VALIDATE });
  const onValidate$ = Observable.merge(
   action$.ofType({ type:
VALIDATE_SUCCESS }),
   action$.ofType({ type:
VALIDATE_FAILURE })
  )
   .first()
   .filter(action => action.type ===
VALIDATE_SUCCESS)
   .mapTo({ type: SUBMIT_START });
  return Observable.merge(validate$,
onValidate$);
 });
```

Pros & cons of redux-observable-like solution

+ Acting on actions, initiating side effects and build actions from them is extracted to Epics
+ You can wait for another action, cancel ongoing action, debounce, throttle, etc.
+ Is easily testable
  + you can directly test an epic, by putting in actions and expecting another ones)
+ Supports TypeScript well
+ After you learn Observables, you can use them in other use cases as well (and 18 other programming languages)

- You need to learn Observables
  - Imagine learning and using Promise for the first time - this is how Observables will be for you in the beginning
  - Because of that, it might take you a few days until you get fluent in them
    P.S. Rx Marbles help a lot
- You might end up having too much business logic in your epics
  - I suggest extracting it into separate services
- You might end up with controlling the state in effects instead of reducers
  - Avoid it, epics should be simple and declarative

In short

1. Redux is an awesome concept, but:
    a. it gives no solution to act on asynchronous actions
2. Redux-thunk is enough in a short term, but:
    a. you'll need to switch to something else eventually
3. A choice between redux-saga and redux-effects (and their other equivalents) is mostly based on your personal taste and previous experience, but:
    a. I suggest going with the Observables way, as:
        i. it is a perfect way to grasp a series of events in a time;
        ii. they are getting more and popular nowadays
        iii. (promoted in Angular since years, and now going into the official ES spec as well)

Related links

- "The world is asynchronous", Werner Vogels, All Things Distributed (15th July 2004)
- "Redux-Saga V.S. Redux-Observable"
- Demo app:
  - redux-thunk: https://stackblitz.com/edit/redux-observable-slide-thunk-demo
  - redux-effects: https://stackblitz.com/edit/redux-observable-slide-demo

Thanks!
Any questions?

Jacek Tomaszewski

Fullstack Web Developer at Recruitee.com (We're hiring!)

Freelancer and Consultant at jtom.me