Redux.js

React Redux Tutorial for Beginners

Leandro.frigerio@devugees.org

States… and then?

I knew what’s the state. But actions, action creators, and reducers?

“I read that the new Context API will replace Redux. Should I bother learning Redux?” and/or “Redux is dead, your tutorial is outdated“. Well let’s make things clear: Redux is not dead. It won’t go anywhere anytime soon. Keep learning Redux!

http://blog.isquaredsoftware.com/2018/03/redux-not-dead-yet/

→ “Yes, the new context API is going to be great for passing down data to deeply nested components - that's exactly what it was designed for. If you're only using Redux to avoid passing down props, context could replace Redux - but then you probably didn't need Redux in the first place. Context also doesn't give you anything like the Redux DevTools, the ability to trace your state updates, **middleware** to add centralized application logic, and other powerful capabilities that Redux enables.”

React Redux tutorial: what you will learn

In the following guide you will learn:

- what is Redux (more)
- how to use Redux with React (less) https://github.com/reactjs/redux/tree/master/examples/todos

To understand what is Redux you must first understand what is the state:

- A stateful React component is basically a Javascript ES6 class.
- Every stateful React component carries its own state.
- In a React component the state holds up data.
- The component might render such data to the user. And there's setState for updating the local state of a component.

React way

What you learned so far with **React**

- render some data from the local state
- update the state with React setState

Now, it is fine to keep the state within a React component as long as the application remains small.

But things could become tricky in more complex scenarios. You will end up with a bloated component filled with methods for managing and updating the state.

The frontend shouldn’t know about the business logic (in a proper MVC).

React and Redux

So what are the alternatives for managing the state of a React component?

Redux is <u>one of them</u>

Redux solves a problem that might not be clear in the beginning: it helps giving each React component the exact piece of state it needs.

Redux holds up the state within a single location (the "Store")

Also with Redux the logic for fetching and managing the state lives outside React. The benefits of this approach might be not so evident. Things will look clear as soon as you'll get your feet wet with Redux.

We'll see why you should learn Redux and when to use Redux within your applications.

About Redux

“Are you trying to learn Redux but you’re going nowhere?”
Redux literally scares most beginners. But that shouldn’t be your case.

Redux is not that hard. **The key is: don’t rush learning Redux just because everybody is using it**. You should start learning Redux if you’re motivated and passionate about it. Take your time.

Good reasons to learn Redux:

- It is interested in learning how Redux works
- You'll improve your React skills
- the combination React/Redux is ubiquitous
- Redux is framework agnostic: learn it once, use it everywhere (Vue JS, Angular)

When to use Redux

It is feasible to build complex React application **without even touching Redux**. That comes at a cost. Redux has a cost as well: it adds another layer of abstraction and complexity. But it's better thinking about Redux as an investment, not as a cost.

Another common question for Redux beginners is: “how do you know when you’re ready to use Redux?” If you think about it there is no rule of thumb for determining when you do need Redux for managing the state.

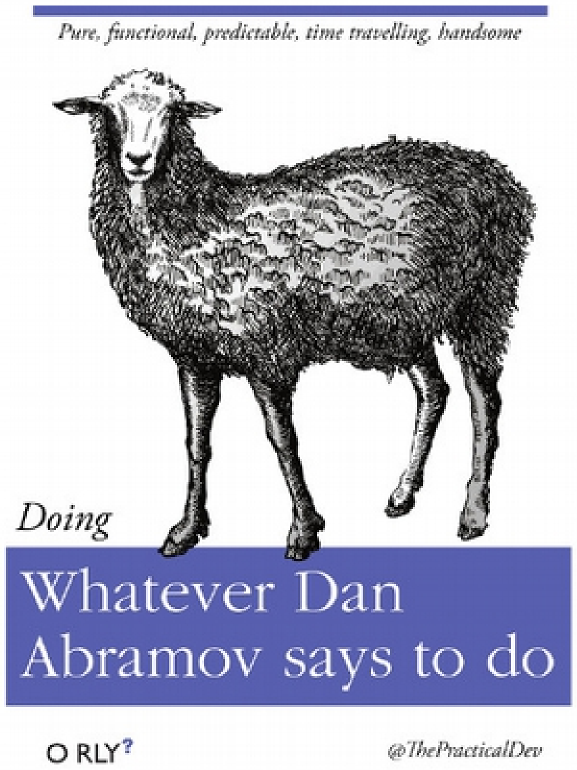
You should consider using Redux when:

- multiple React components needs to access the same state but do not have any parent/child relationship
- you start to feel awkward passing down the state to multiple components with props

Do like Dan says

If that makes still no sense for you do not worry, well it's not that bad...

Dan Abramov says “Flux libraries are like glasses: you’ll know when you need them.” And in fact it worked like that for me.


Pure, functional, predictable, time travelling, handsome
Doing
Whatever Dan Abramov says to do
O RLY?
@ThePracticalDev

Before you start

Before going further take your time to understand what problem does Redux solve and whether you’re motivated or not to learn it.

Be aware that Redux is not useful for smaller apps. It really shines in bigger ones. Anyway, learning Redux even if you’re not involved in something big wouldn’t harm anyone.

Be familiar with https://redux.js.org/introduction/three-principles

Redux can be described in three fundamental principles:

- Single source of truth
- State is read-only
- Changes are made with pure functions

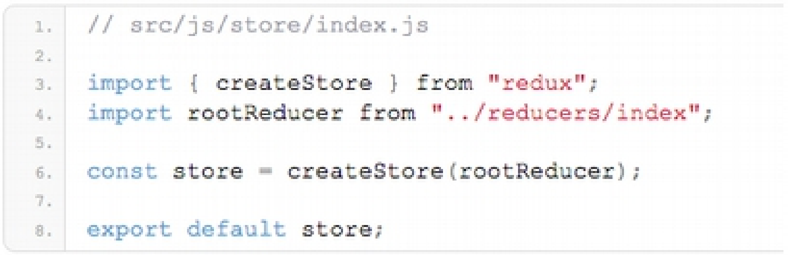
The Redux store, the actions and reducers.

Actions. Reducers. You kind of knew about them. But one thing wasn’t clear to is how were all the moving parts glued together?

The Store

The **Store** orchestrates all the moving parts in Redux. Repeat with me: the store. The store in Redux is like the human brain: it’s kind of magic.

The Redux store is fundamental: the state of the whole application lives inside the store. So to start playing with Redux we should create a store for wrapping up the state.



```
// src/js/store/index.js

import { createStore } from "redux";
import rootReducer from "../reducers/index";

const store = createStore(rootReducer);

export default store;
```

createStore is the function for creating the Redux store.
createStore takes a reducer as the first argument, rootReducer in our case.

States & Reudcers

You may also pass an initial state to createStore. But most of the times you don't have to. Passing an initial state is useful for server side rendering. Anyway, the state comes from reducers.

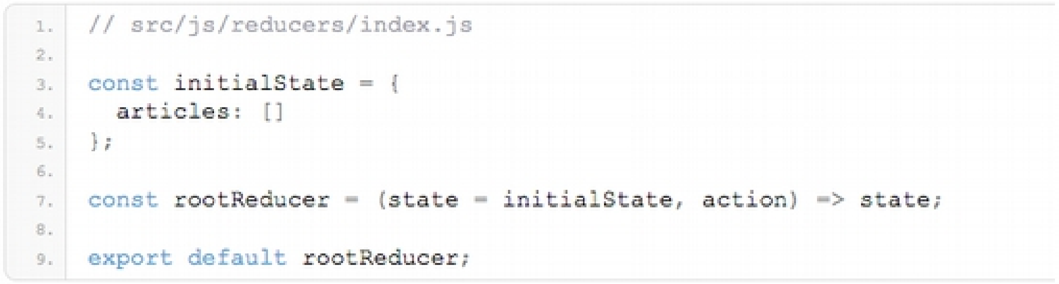
In Redux reducers produce the state. The state is not something you create by hand. A reducer is just a Javascript function. A reducer takes two parameters: the current state and an action (more about actions soon).

The third principle of Redux says that the state is immutable and cannot change in place. This is why the reducer must be pure. A pure function is one that returns the exact same output for the given input. In plain React the local state changes in place with setState. In Redux you cannot do that.

In plain React the local state changes in place with setState.
In Redux you cannot do that.

Reducer example

In our example we'll be creating a simple reducer taking the initial state as the first parameter. As a second parameter we'll provide action. As of now the reducer will do nothing than returning the initial state.



```
// src/js/reducers/index.js

const initialState = {
  articles: []
};

const rootReducer = (state = initialState, action) => state;

export default rootReducer;
```

I promised to keep this guide as simple as possibile. That's why our first reducer is a silly one: it returns the initial state without doing anything else.

Notice how the initial state is passed as a default parameter.

Redux actions

Redux reducers are without doubt the most important concept in Redux. Reducers produce the state of the application.

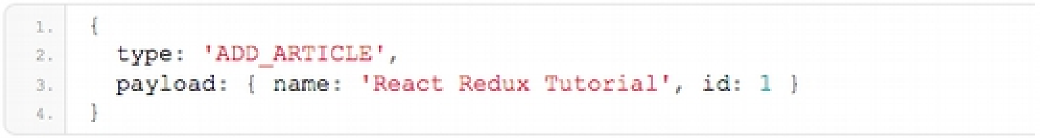
But how does a reducer know when to produce the next state?

The second principle of Redux says the only way to change the state is by sending a signal to the store. This signal is an action. “Dispatching an action” is the process of sending out a signal.

Now, how do you change an immutable state? **You won’t**.
The resulting state is a copy of the current state plus the new data.

Redux actions are objects!

The reassuring thing is that Redux actions are nothing more than Javascript objects. This is what an action looks like:



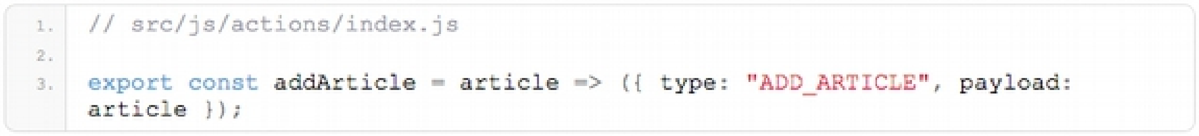
```
{
  type: 'ADD_ARTICLE',
  payload: { name: 'React Redux Tutorial', id: 1 }
}
```

Every action needs a type **property** for describing how the state should change. You can specify a **payload** as well. In the above example the payload is a new article. A **reducer** will add the article to the current state later.

It is a best pratice to wrap every action within a function.
Such function is an **action creator**.

Action in action

Let’s put everything together by creating a simple Redux action.



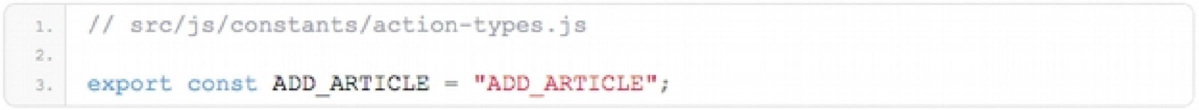
```
// src/js/actions/index.js

export const addArticle = article => ({ type: "ADD_ARTICLE", payload:
article });
```

So, the type property is nothing more than a string. The reducer will use that string to determine how to calculate the next state. Since strings are prone to typos and duplicates it’s better to have action types declared as constants.

This approach helps avoiding errors that will be difficult to debug.

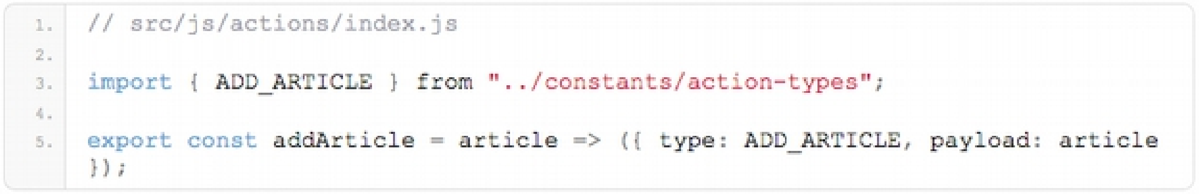
How to Action-types

How to: create a new file named action-types.js



```js
// src/js/constants/action-types.js

export const ADD_ARTICLE = "ADD_ARTICLE";
```

Import it and use it in your Action



```js
// src/js/actions/index.js

import { ADD_ARTICLE } from "../constants/action-types";

export const addArticle = article => ({ type: ADD_ARTICLE, payload: article
});
```

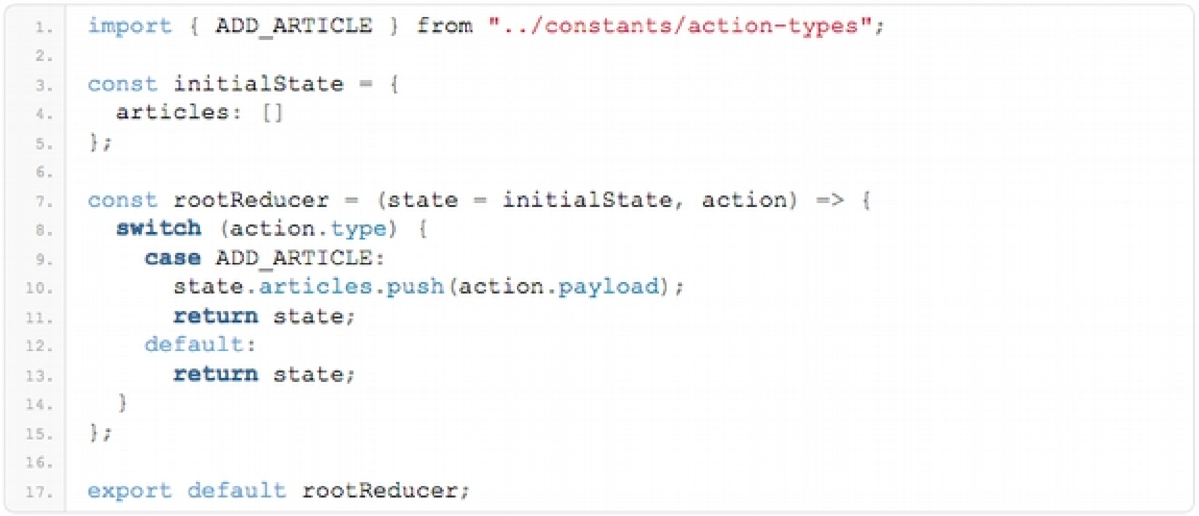
Re-wrap all together

- the Redux store is like a brain: it’s in charge for orchestrating all the moving parts in Redux
- the state of the application lives as a single, immutable object within the store
- as soon as the store receives an action it triggers a reducer
- the reducer returns the next state

What’s a Redux reducer made of? A reducer is a Javascript function taking two parameters: the state and the action. A reducer function has a switch statement (although unwieldy, a naive reducer could also use if/else). The reducer calculates the next state depending on the action type. Moreover, it should return at least the initial state when no action type matches.

Reducers example

When the action type matches a case clause the reducer calculates the next state and returns a new object. Here’s an example



```
import { ADD_ARTICLE } from "../constants/action-types";

const initialState = {
  articles: []
};

const rootReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_ARTICLE:
      state.articles.push(action.payload);
      return state;
    default:
      return state;
  }
};

export default rootReducer;
```

Although it’s valid code the above reducer breaks the main Redux principle: immutability.

Pure & Impure Functions

Impure functions

An impure function is a function that mutates variables/state/data outside of it’s lexical scope, thus deeming it “impure” for this reason. There are many ways to write JavaScript, and thinking in terms of impure/pure functions we can write code that is much easier to reason with.

Pure functions

A pure function is much easier to comprehend, especially as our codebase may scale, as well as role-based functions that do one job and do it well. Pure functions don’t modify external variables/state/data outside of the scope, and returns the same output given the same input. Therefore it is deemed “pure”.

→ https://egghead.io/lessons/react-redux-pure-and-impure-functions

Some functions are more predictable than others.

Pure function

A function is considered **pure** if:

- a) It always returns the same value when given the same arguments
- b) It does not modify anything (arguments, state, database, I/O, ..)

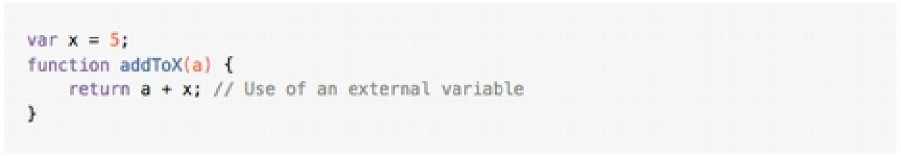
```
function add(x, y) {
   return x + y;
}
```

Impure functions

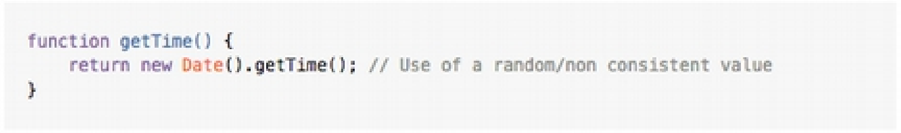
A function is considered impure if it is not pure (!), typically because:

a) it makes use of an external or random value. (It stops being entirely contained and predictable)

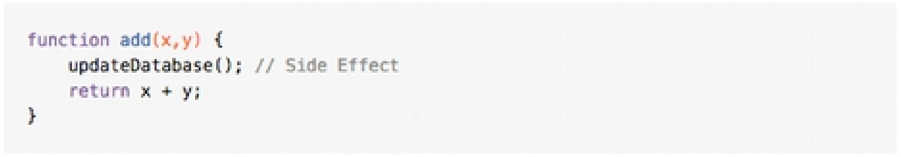
b) It performs an external operation, in other words it causes side effects



```
var x = 5;
function addToX(a) {
    return a + x; // Use of an external variable
}
```



```
function getTime() {
    return new Date().getTime(); // Use of a random/non consistent value
}
```



```
function add(x,y) {
    updateDatabase(); // Side Effect
    return x + y;
}
```

Big deal of Pure functions

Predictable

Yep, I know, it’s part of the definition, but using a function and knowing exactly what it does feels so good.

Portable

Pure functions can be reused easily as they do not hold any form of state.

Cacheable

Since the result will always be the same, it becomes easy to cache results.

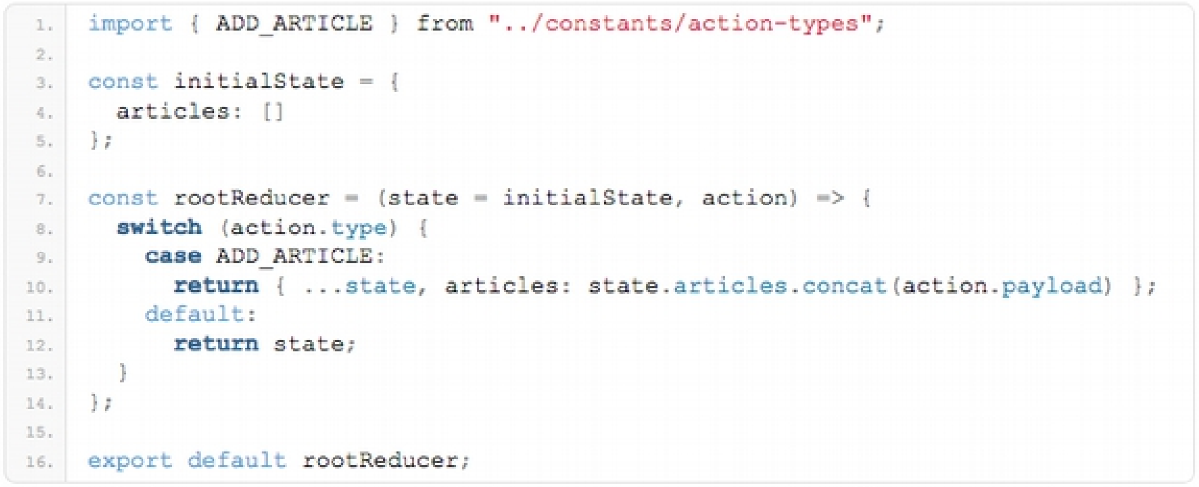
Testable

Last but not least, pure functions are easy and a pleasure to test with automated tests.

Purify

Array.prototype.push is an impure function: it alters the original array.

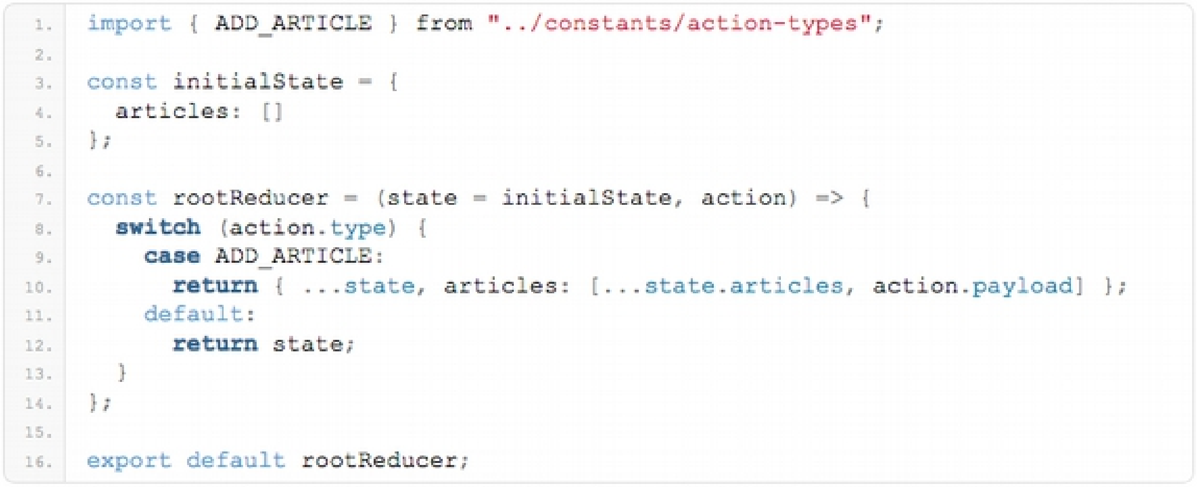
Making our reducer compliant is easy. Using Array.prototype.concat in place of Array.prototype.push is enough to keep the initial array immutable:



```
import { ADD_ARTICLE } from "../constants/action-types";

const initialState = {
  articles: []
};

const rootReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_ARTICLE:
      return { ...state, articles: state.articles.concat(action.payload) };
    default:
      return state;
  }
};

export default rootReducer;
```

Best practices

We’re not done yet! With the spread operator we can make our reducer even better:



```
import { ADD_ARTICLE } from "../constants/action-types";

const initialState = {
  articles: []
};

const rootReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_ARTICLE:
      return { ...state, articles: [...state.articles, action.payload] };
    default:
      return state;
  }
};

export default rootReducer;
```

In the example above the initial state is left utterly untouched. The initial articles array doesn’t change in place. The initial state object doesn’t change as well. The resulting state is a copy of the initial state.

Avoiding mutations in Redux

There are two key points for avoiding mutations in Redux:

- Using concat(), slice(), and …spread for arrays
- Using Object.assign() and …spread for objects

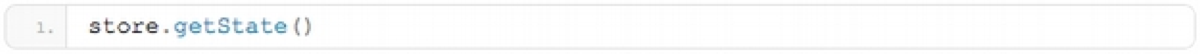
Redux protip: the reducer will grow as your app will become bigger.

You can split a big reducer into separate functions and combine them with **combineReducers**

Redux store methods

Redux itself is a small library (2KB). The Redux store exposes a simple API for managing the state. The most important methods are:

- **getState** for accessing the current state of the application
- **dispatch** for dispatching an action
- **subscribe** for listening on state changes



```
store.getState()
```

GetState, Subscribe, Dispatch

The subscribe method accepts a callback that will fire whenever an action is dispatched. Dispatching an action means notifying the store that we want to change the state.

```
store.subscribe(() => console.log('Look ma, Redux!!'))
```

To change the state in Redux we need to dispatch an action. To dispatch an action you have to call the dispatch method.

```
store.dispatch( addArticle({ name: 'React Redux Tutorial for Beginners', id:
1 }) )
```

How to Redux...

- You know what reducers and actions are
- You know how to access the current state with getState.
- You know how to dispatch an action with dispatch
- You know how to listen for state changes with subscribe

Then you know Redux… how to couple React and Redux together?

“should I call getState within a React component? How do I dispatch an action from a React component? ”… and so on… questions, you know...

Resources

- https://redux.js.org
- https://redux.js.org/introduction/three-principles
- https://egghead.io/courses/getting-started-with-redux
- https://www.valentinog.com/blog/react-redux-tutorial-beginners/
- https://github.com/markerikson/react-redux-links
- http://jonnyreeves.co.uk/2016/redux-middleware/
- https://github.com/LeoDvg/React-Redux-TodoMvc