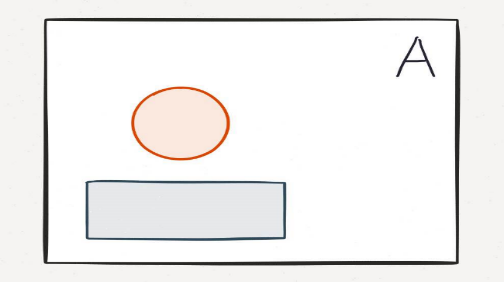
<https://www.youtube.com/watch?v=s9AC8KTcce8>

What is Redux?

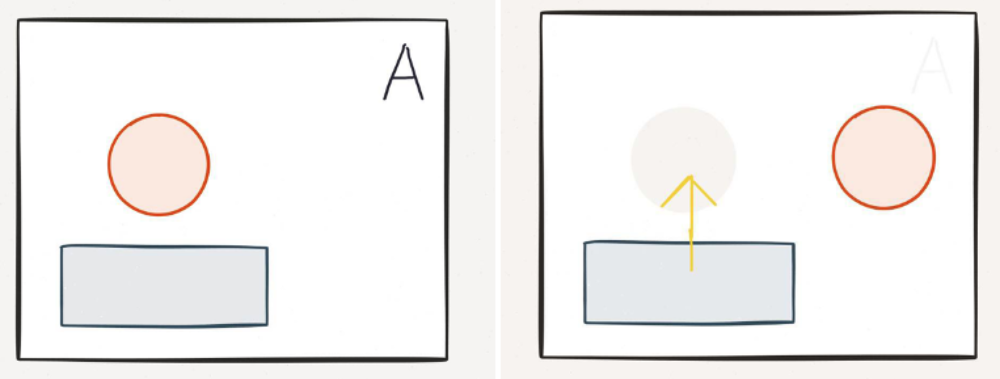
I'm going to start with an analogy. Imagine you have a room with a sofa and a table, like this:



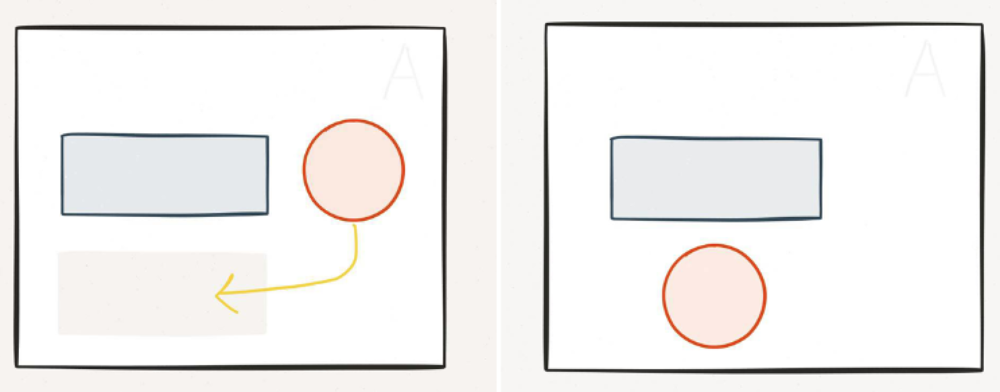
You want to swap the positions of the sofa and table, and you need it done FAST, so you hire two butlers, Alfred and Rudy, to complete the task at once.

Alfred is in charge of the sofa, and Rudy is in charge of the table. You give them instructions to coordinate, so they aren't crashing furniture into each other.

First, Rudy moves the table out of the way, then Alfred moves the sofa in position.



Next, Rudy moves the table where the sofa used to be. The room now reflects its new desired state.

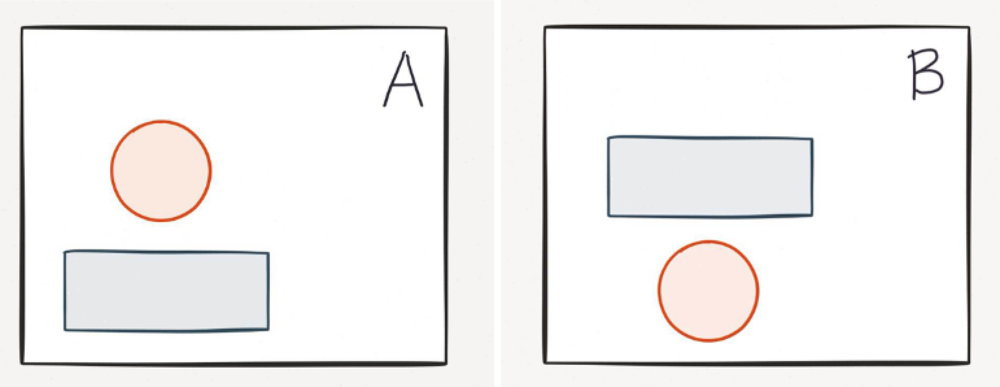


Super! I've just described a common implementation pattern found in MVC architecture. That wasn't so bad, but let's take it up a notch.

Now imagine you have a GENIE. This Genie provides a completely new instance of the room whenever you want to make a change to anything inside of it. And by "new instance" I mean, unlike Alfred and Rudy, she does not go inside Room A and move stuff around until Room A looks like Room B. She actually **creates a new, unique room** that is exactly Room B. If that wasn't enough, **she also reads your mind**. I know, right? It's an excellent setup you've got here.

Instead of giving instructions to multiple butlers about what goes where, all you do is think, Hey Genie, I want this room to be this other room, which right now exists in our shared mind palace.

And POOF. Your Genie returns that new room in real life, so you now have TWO rooms, like this:



The Genie? That's what Redux achieves with React. That's right. THE GENIE IS REAL.

Why is this desirable? Let's say the new furniture configuration is problematic for some reason. Like, maybe the sofa blocks a doorway, which creates suboptimal feng shui, and, also, a fire hazard. Instead of having Alfred and Rudy go back and re-position the furniture to where it was, you can just toss aside Room B, and flip back to Room A.

Or, you can think of a completely different configuration, Room C, and your Genie will return it. If you ever want to revisit Rooms A or B, you can flip back to them. You might think that producing new rooms in this manner is wasteful and slow. It's not. Why? Because she's a motherfucking genie. This is her thing.

Seriously though, right now, it's not important to know why it's fast, just know that it is very, very fast -- in fact, it's faster than the "If this, do that, then do this other thing..." approach embodied in the butlers example.

The capability to create new, immutable states is somewhat grandiosely (or maybe just dorkily) called "time travel" because you can traverse between your app's states. This is what React/Redux gives you. React provides the ability to spin up new rooms, and Redux provides the Genie mind-reader logic behind their production. This logic is very important, and is different from other implementations of a concept called flux, which we'll discuss briefly, next.

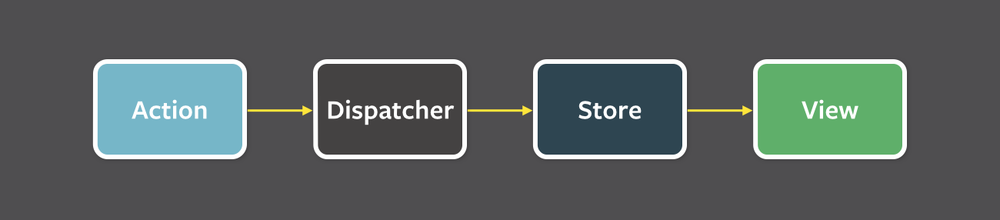
What is Redux, Actually?

Redux, according to [the docs](https://github.com/rackt/redux), is "**a predictable state container for JavaScript apps**." It's a very lightweight (2kB) implementation of flux. If you haven't heard of flux before, that's fine. Nick explained it to me in about 10 seconds.

Basically, flux is like a sedan. It's a concept with many agreed-upon characteristics associated with it, but it's not, itself, a tangible thing. Just as there's no single definitive "Sedan" that you can go out and buy, there's no definitive "Flux" that you can download and install. Instead, you choose one specific implementation of it. I don't know much about cars, but whatever the most minimal, stripped down, yet highly performant sedan is, that's Redux.

The basic idea behind flux is, you have a dispatcher that sends actions to the data store, and updates the view. It's an overall data flow architecture, where -- unlike the two-way data binding of MVCs -- updates only go one way. If you've ever been in getter/setter hell trying to track what view is touching what model, you can understand how a one-way data flow might be beneficial.

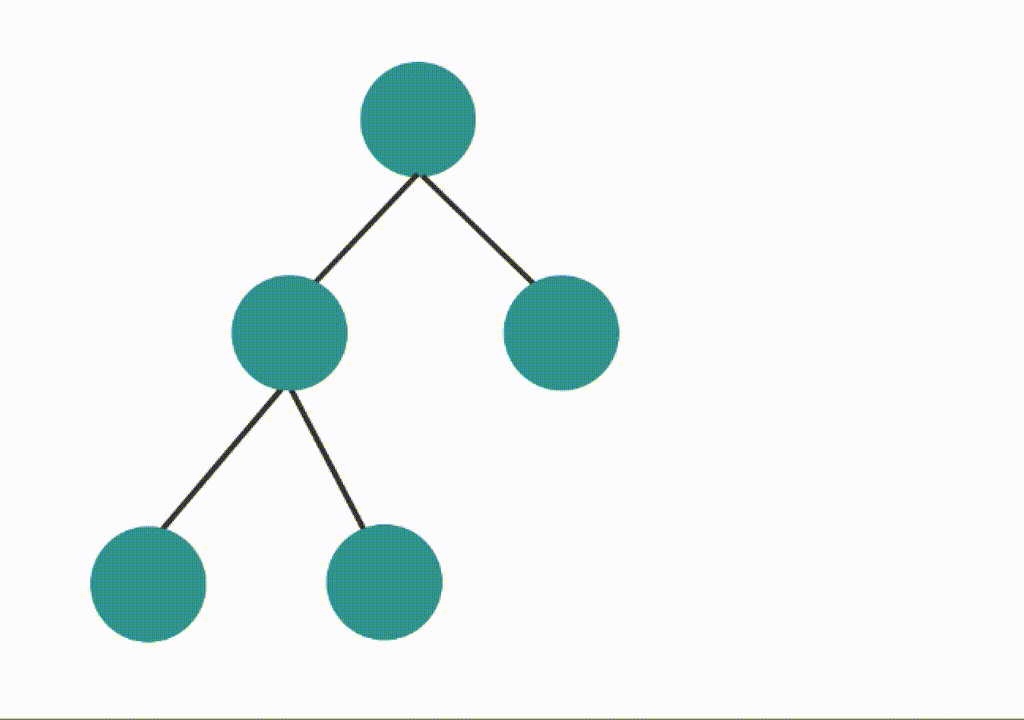
Note: This isn't a knock on frameworks like Angular (which I still think is pretty magical) or Backbone (which I... respect as a thing), or any other MVC. In general, I try not to get attached to any one way as the best way for everyone. What constitutes the "best" approach really depends on your application, your use case, and your organization's needs and constraints.



One huge implication of one-way (or unidirectional) data flow is that your data store is not touched by your view. At all.

Redux is one particular implementation of a **subset** of flux concepts. The main differences between Redux and full flux implementations are:

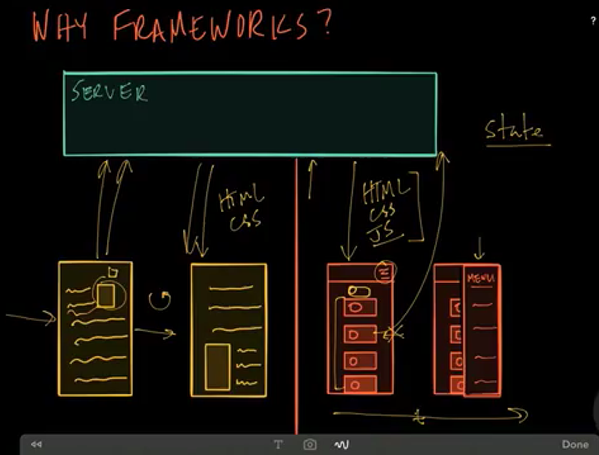
* There are no discrete dispatchers in Redux; your store listens directly for actions, and uses a function called a reducer (more on this later) to return a new app state each time an action is dispatched.
* Redux holds your entire application's state in one place.
* Your app's state is immutable.

Consider your app state as a tree, with a root and child nodes. If you want to make a change to a child node, you can't change it directly because state is immutable, so you make a copy of the target node and apply your changes. You also make a copy of the target node's parent, because it wouldn't make sense to have one parent produce two versions of the exact same node. Maybe you append a child node to the new node, too. That's fine. At the end of all this, you just need to attach the existing, unchanged nodes to your new ones. Like this: 

The best part is, the original state (the blue one) still exists. The unchanged blue nodes are not re-rendered in the new state, just the orange nodes. This makes debugging really easy because you can isolate the action that caused the state change, and go back and forth at any time. Redux dev tools also happen to be unusually good (see the next section for an example), and, combined with React dev tools, have basically spoiled me for life.

Another implication of this design pattern: You really cut down on the number of dependent asynchronous operations in your data flow. Going back to the room example, recall how you managed the butlers: "Wait for the table to be out of the way, then move sofa..."

Reach on React, Flux, Redux, ReactiveX.js, angularjs 2, Notejs



React's performance come from the idea of "diffing" DOM updates before actually updating the application if a list of three items becomes four, React will calculate the least amount of changes to make the new DOM of the app have a four item list. Drastically simplified, React runs a function to return a "virtual" DOM and efficiently compares it to the current virtual DOM.

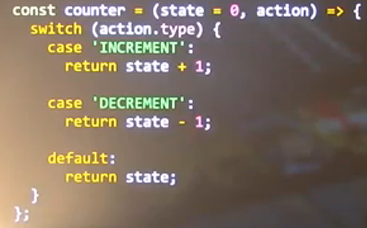
What is **Redux**? : Provide predictable state management using “actions” and “reducers”.

Or in other word it’s the implementation for flux to manage the state.

What is “**actions**”? : Describes something has (or should) happen, but they don’t specify how it should be done.

What is an “**reducer**”? : A pure function that takes the previous **state** and an **action** and returns the **new state**

**e.g of reducer**



Reducers handle state transitions, but they must be done synchronously, in other world Reducer can’t work async

What are async stuff do we commonly do?

* User interactions(mouse, keyboard, etx)
* Ajax
* Timers/Animations
* Web Sockets
* Work Workers, etc

Simple example

createStore(reducer, [preloadedState], [enhancer])

**Store Methods**

getState()

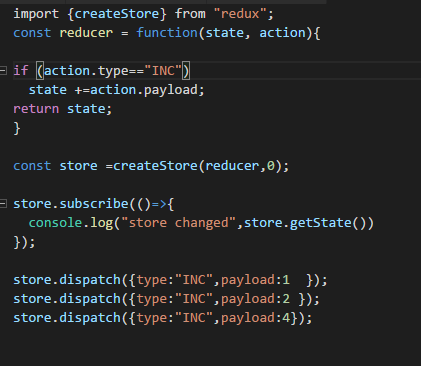
dispatch(action)

subscribe(listener)

replaceReducer(nextReducer)

combineReducers({ counter: counter, todos: todos }).

preloadedState: The initial state. You may optionally specify it to hydrate the state from the server in universal apps, or to restore a previously serialized user session. If you produced reducer with [combineReducers](http://redux.js.org/docs/api/combineReducers.html), this must be a plain object with the same shape as the keys passed to it. Otherwise, you are free to pass anything that your reducer can understand.

[enhancer] (Function): The store enhancer. You may optionally specify it to enhance the store with third-party capabilities such as middleware, time travel, persistence, etc. The only store enhancer that ships with Redux is applyMiddleware().

Some can be handled synchronously e.g

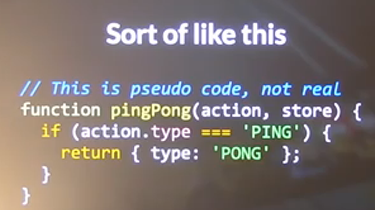


Sometimes you need more control on it e.g.

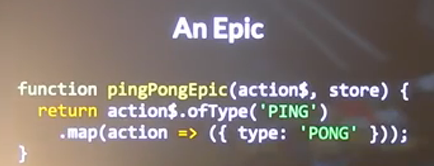
* Ajax cancellation/composing
* Debounce/throttle/buffer/etc
* Drag and Drop
* Web Sockets, Work Workers, etc

**What is an Epic**

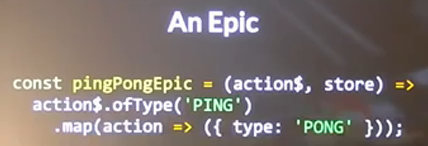
A function that takes a stream of all actions that application want to dispatched and returns a stream of new actions to dispatch. In short “actions in, actions out” e.g



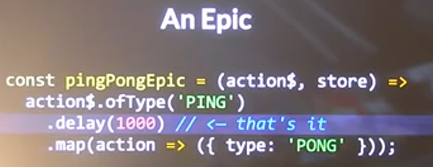
In An Epic it can be written like this



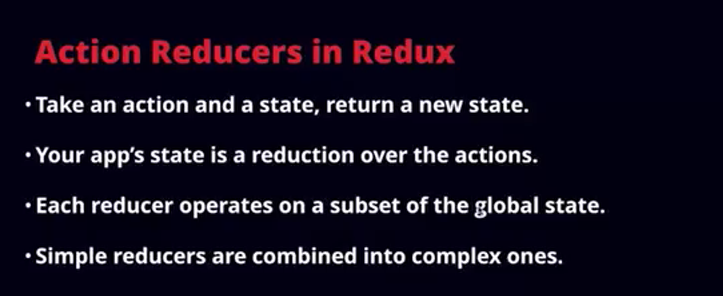
Or can be



Suppose if we want the delay in that we can do it by using like this





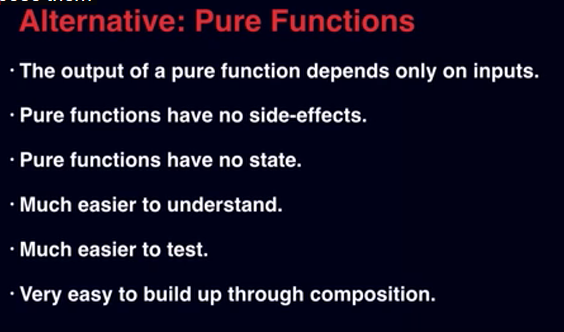


Reducers:





Redux : is pure function programming,





([Redux](http://redux.js.org/) is a predictable state container for JavaScript apps.)

***Redux provides an easy way to centralize the state of your application.***

Redux is like “*a backend’s database on the client-side where you store all the required information that are required in order to generate the view*”. It helps you to query SELECT, INSERT and UPDATE the data (a record) into the JSON’s database (called also single-state tree).

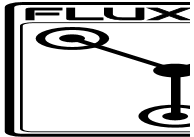
In other words

Redux is a flux implementation but the simplest one to follow and it brings predictability to your application state which helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

It's basically comparing a JS object with another JS object. Redux's reducers are where state is updated, and your data set would be operated on.

If application is complex it would be advisable to use redux, However if it’s just a basic app, redux/flow is not necessary really.

[***https://www.youtube.com/watch?v=s4xr2avwv3s&list=PLw5h0DiJ-9PB3iBj86OUsFIu\_yN7v8IOw***](https://www.youtube.com/watch?v=s4xr2avwv3s&list=PLw5h0DiJ-9PB3iBj86OUsFIu_yN7v8IOw)

 What is Flux?[**¶**](http://fluxxor.com/what-is-flux.html#what-is-flux-)

Flux is an architecture for creating data layers in JavaScript applications. It was designed at Facebook along with the [React](http://facebook.github.io/react/) view library. It places a focus on creating **explicit and understandable update paths** for your application's data, which makes **tracing changes during development simpler** and makes **bugs easier to track down and fix**.

[***http://blog.angular-university.io/angular-2-application-architecture-building-flux-like-apps-using-redux-and-immutable-js-js/***](http://blog.angular-university.io/angular-2-application-architecture-building-flux-like-apps-using-redux-and-immutable-js-js/)

***(React and Angularjs2)***[***http://blog.celerity.com/react/flux-from-an-angularjs-perspective***](http://blog.celerity.com/react/flux-from-an-angularjs-perspective)

[***https://github.com/przeor/react-router-flux-starter-kit***](https://github.com/przeor/react-router-flux-starter-kit)

**flux-angular** [***https://github.com/christianalfoni/flux-angular***](https://github.com/christianalfoni/flux-angular)

Flux is the programming pattern which takes care of M in MVC, where all models are event driven. It means that you don't ask for the data, the model is going to tell you when the data is ready. The great part about it that different application components can listen to this change and react accordingly.   
  
React.js is taking care of V in MVC, and does it great.   
  
But you still missing C part. And at C part is what makes your web feels like desktop app. For C I prefer react-router.  
  
With all this being said Flux + React + React-Router gives you an amazing framework for building fast, responsive, memory efficient isomorphic apps.

**What’s a thunk?!**

A [thunk](https://en.wikipedia.org/wiki/Thunk) is a function that wraps an expression to delay its evaluation.

// calculation of 1 + 2 is immediate

// x === 3

let x = 1 + 2;

// calculation of 1 + 2 is delayed

// foo can be called later to perform the calculation

// foo is a thunk!

let foo = () => 1 + 2;

**What’s a redux promise?**

Redux promise middleware enables robust handling of async code in [Redux](http://redux.js.org/). The middleware enables optimistic updates and dispatches pending, fulfilled and rejected actions. It can be combined with [redux-thunk](https://github.com/gaearon/redux-thunk) to chain async actions.

Are you using [redux-promise](https://github.com/acdlite/redux-promise), or [redux-promise-middleware](https://github.com/pburtchaell/redux-promise-middleware)?

If using redux-promise-middleware:

The action returned by your action creator should have a **payload** property, in that payload there should be a promise property

function apiCallToAddTodo(text) {

return aPromise;

}

function addTodo(text) {

return {

type: 'TODO\_ADDED',

payload: {

promise: apiCallToAddTodo(text)

}

}

}

This is a little more explicit and clear to me than what redux-promise is doing, in which it looks like the thing returned from your action creator itself is a promise, /or/ your payload itself is a promise. If using redux-promise, I think it would be more like:

function apiCallToAddTodo(text) {

return aPromise;

}

function addTodo(text) {

return {

type: 'TODO\_ADDED',

payload: apiCallToAddTodo(text)

}

}

**Axios** is a promise-based HTTP client that works both in the browser and in a node.**js** environment. It basically provides a single API for dealing with XMLHttpRequest s and node's http interface