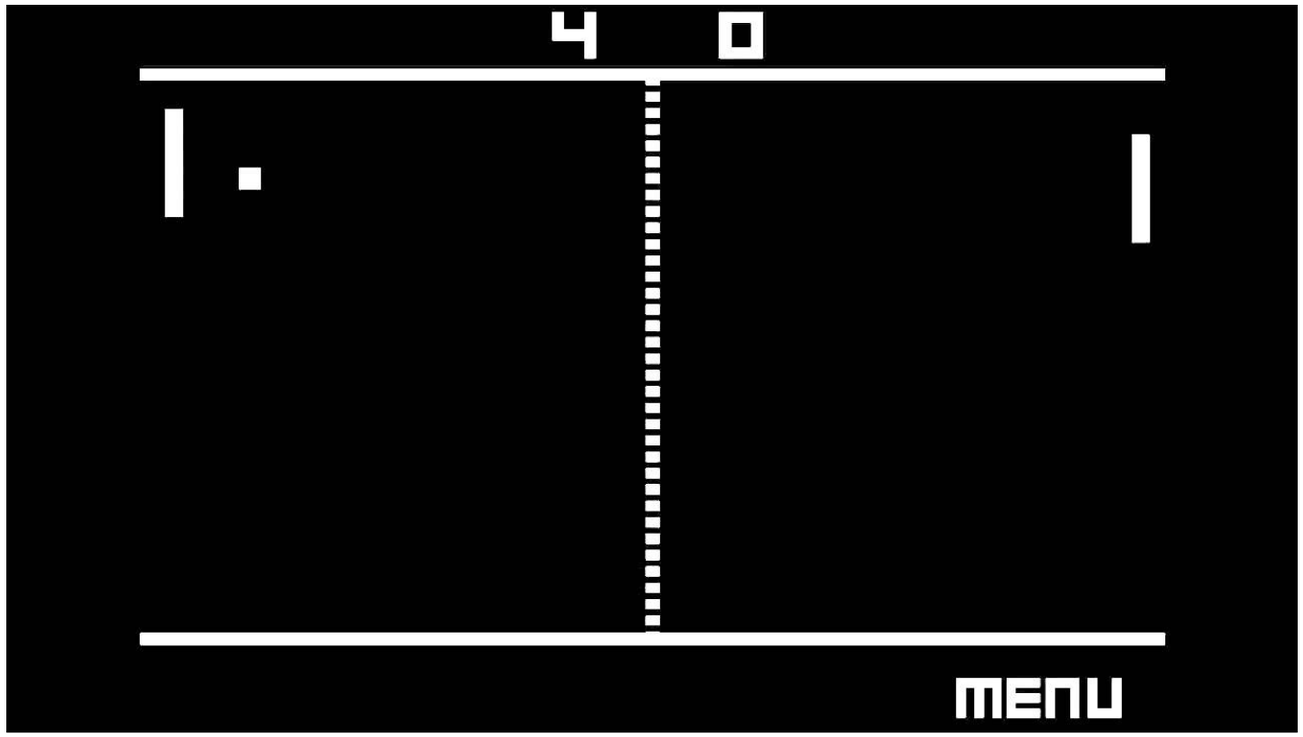
REDUX

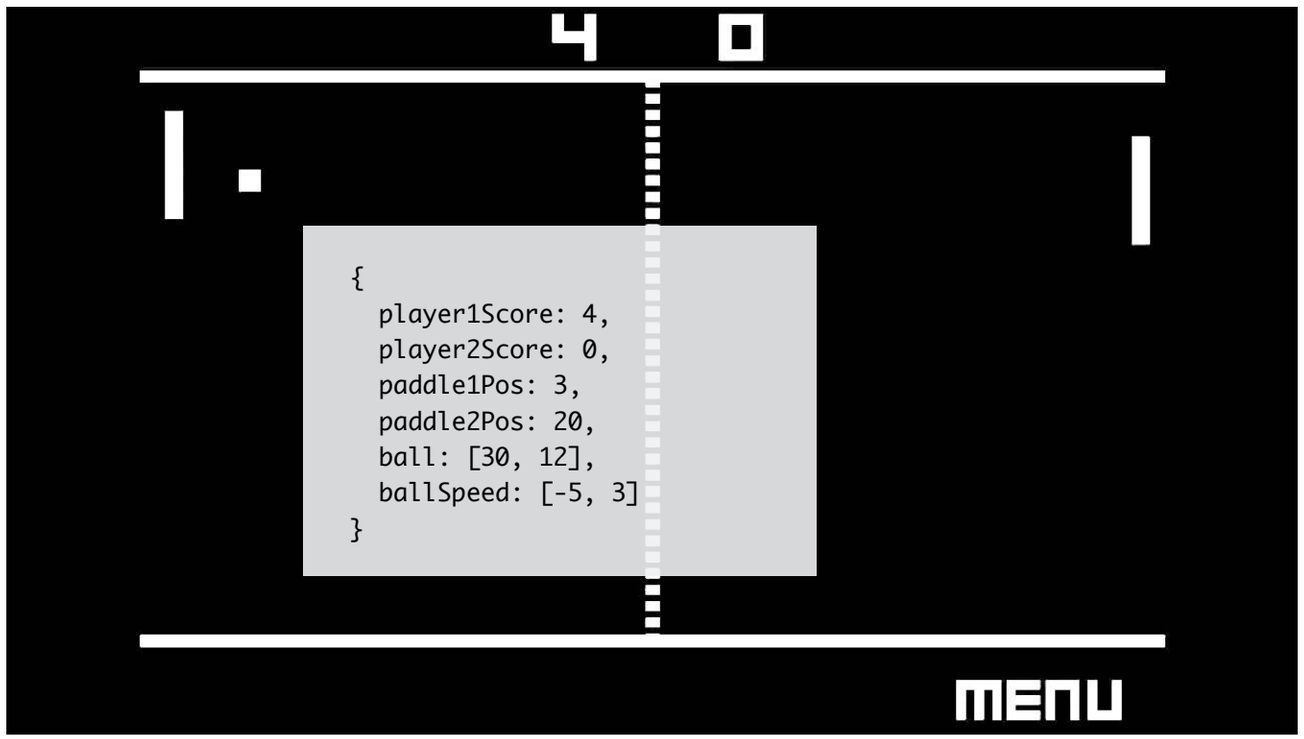
A JavaScript state management library

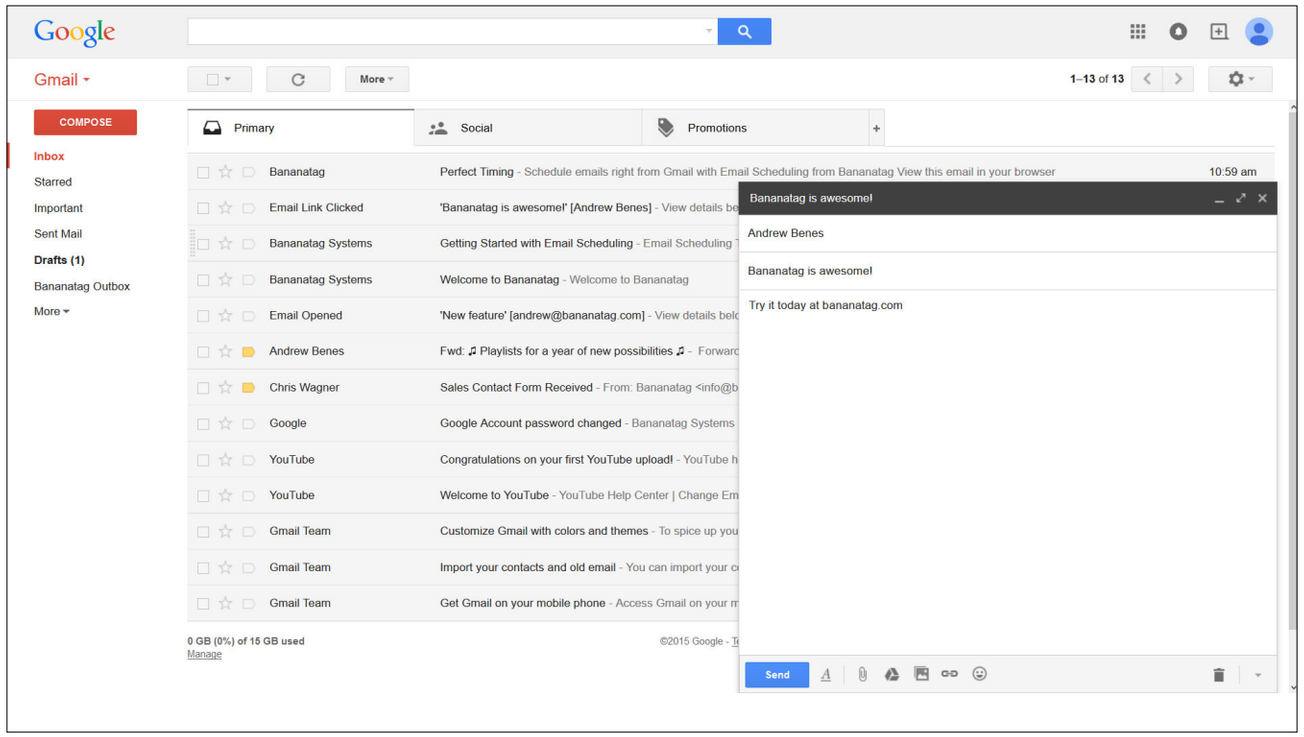
STATE



Everybody knows Pong, right. Now, can you describe all the state in this application?

In most cases, we can simply use a plain JavaScript object to hold the state:


4
0
{
player1Score: 4,
player2Score: 0,
paddle1Pos: 3,
paddle2Pos: 20,
ball: [30, 12],
ballSpeed: [-5, 3]
}
MENU



But what if the application gets especially complex, with lots of state changing over time?

Can you quickly tell me what is the state in this application? No, right?

WHAT REDUX IS

- **State Management Library:**
  - A small tool for containing, accessing and affecting a set of information — often called “state”

- If you **do not** have problems with state management, you might find the benefits of Redux harder to understand.

You’re gonna have to believe us on this one, since you haven’t had experience enough to encounter state management problems. Just know that many people have, especially as applications get bigger and more complex, and that’s why Redux is so popular in the industry.

PRINCIPLES

- Single source of truth
- Data is read-only
- Changes can be requested through actions and are made with pure functions

In practice, what Redux provides us with is a STORE

THE STORE

- The single holder of information
- **Read-Only:** Provides methods to **access state** & **listen for state changes**
- Store can receive dispatched signals (actions) meant to affect state.

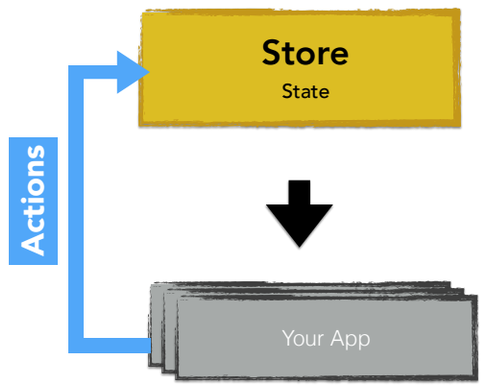
The Redux store provides methods for getting the state and subscribing to state changes – but there is no setter method to directly manipulate state inside the Store. As far as the rest of the application is concerned, the store is read-only.

Obviously there are many circumstances where you might want to change the state in the store: user interactions, AJAX requests, timers, web socket events etc.

The store provides a mechanism by which other parts of the application can signal that the state needs to change: dispatching actions.

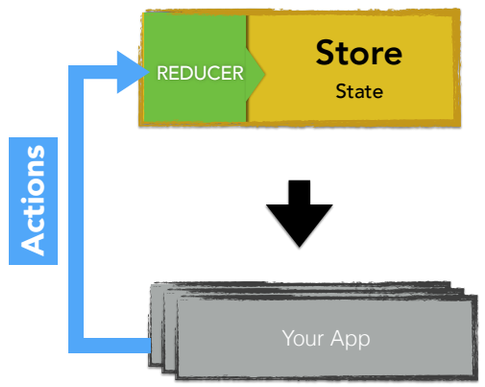
ACTIONS

- Store can receive dispatched signals (actions) meant to affect state.
- Loosely defined as **“things that happen in your app that affect state”**
- Dispatching an action triggers the **reducer** to produce a new state



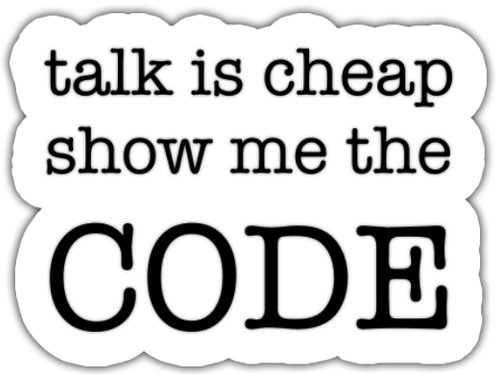
INSIDE THE STORE: **REDUCERS**

- Dispatching an action triggers the **reducer** to produce a new state
- Decides: based on this signal (action), what the new state should be.



Decides: based on this signal (action), the new state should be this

Creates new states per action, rather than modifying previous state


talk is cheap
show me the
CODE

```
import { createStore } from 'redux';

const reducer = (state, action) => {
  /* check the action, maybe produce a new state */
  return state;
};

const store = createStore(reducer);
```


Here I’m just creating the store - notice that in order to create the store I must provide the reducer.

The reducer is responsible to producing the state. If I simply want to start with some default value, I can use JavaScript default function parameters.

In this case, the store simply starts with the value {balance: 0}

```
import { createStore } from 'redux';

const reducer = (state = {balance: 0}, action) => {
  /* check the action, maybe produce a new state */
  return state;
};

const store = createStore(reducer);
store.getState() // {balance: 0}

store.dispatch({type: 'DEPOSIT', amount: 100})
```

Action

Now, remember that the store is read-only. The only way to cause the store state to change is by dispatching an action.

In plain english, an action is simply a JavaScript object with a `type`.

```
import { createStore } from 'redux';

const reducer = (state = {balance: 0}, action) => {
  if(action.type === 'DEPOSIT')
    return {balance: state.balance + action.amount};
  else
    return state

};

const store = createStore(reducer);
store.getState() // {balance: 0}

store.dispatch({type: 'DEPOSIT', amount: 100})

store.getState() // {balance: 100}
```


Now, remember that the store is read-only. The only way to cause the store state to change is by dispatching an action.

In plain english, an action is simply a JavaScript object with a `type`.


CAUTION
ANALOGIES AHEAD
PROTECTIVE HEADGEAR MUST BE WORN IN THIS AREA

THE STORE

- Single holder of state
- You can check the state, be notified of changes, but you cannot directly modify the state inside the store.

ACTIONS ≅ BANK TRANSACTIONS

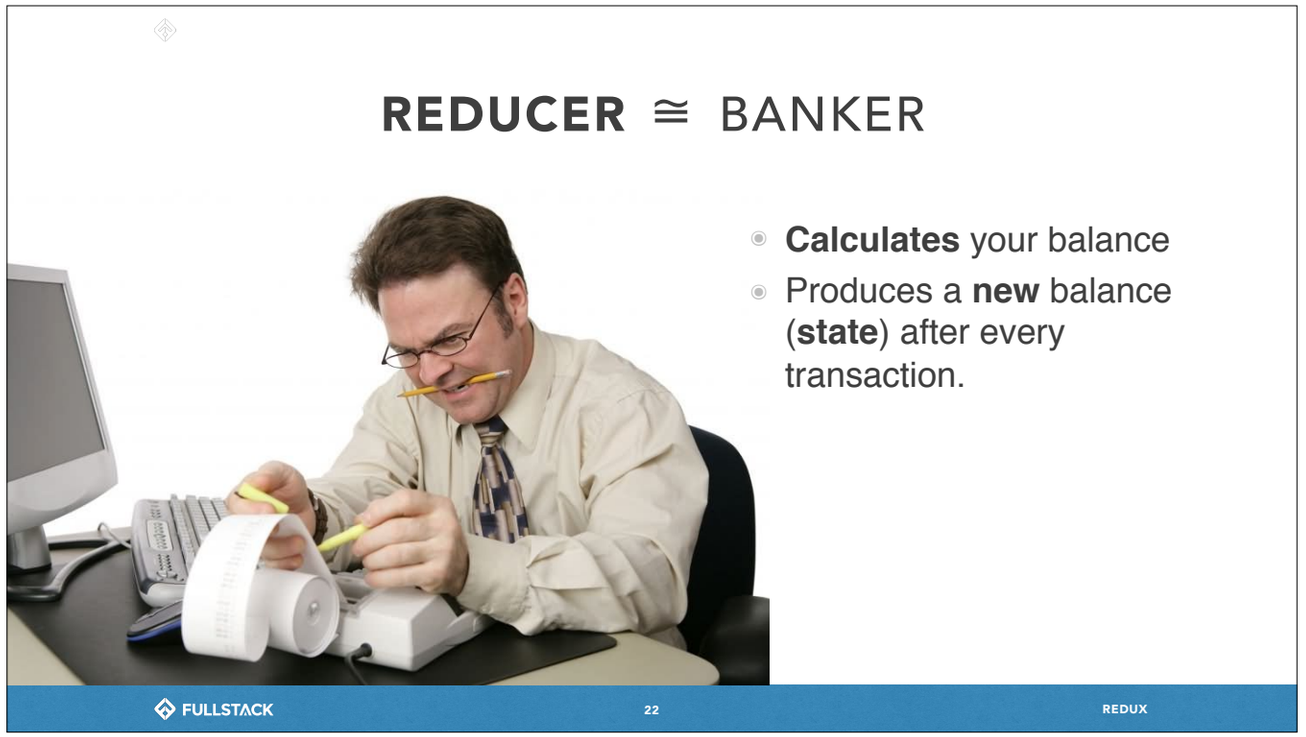
Transaction	Amount	Balance
Create Account	—	$0.00
Deposit	$200.00	$200.00
Withdraw	($50.00)	$150.00
Deposit	$100.00	$250.00

A bank account is made of transactions. With every transaction we update another value called balance.

These transactions are how we're interacting with our bank. They modify the state of our account.

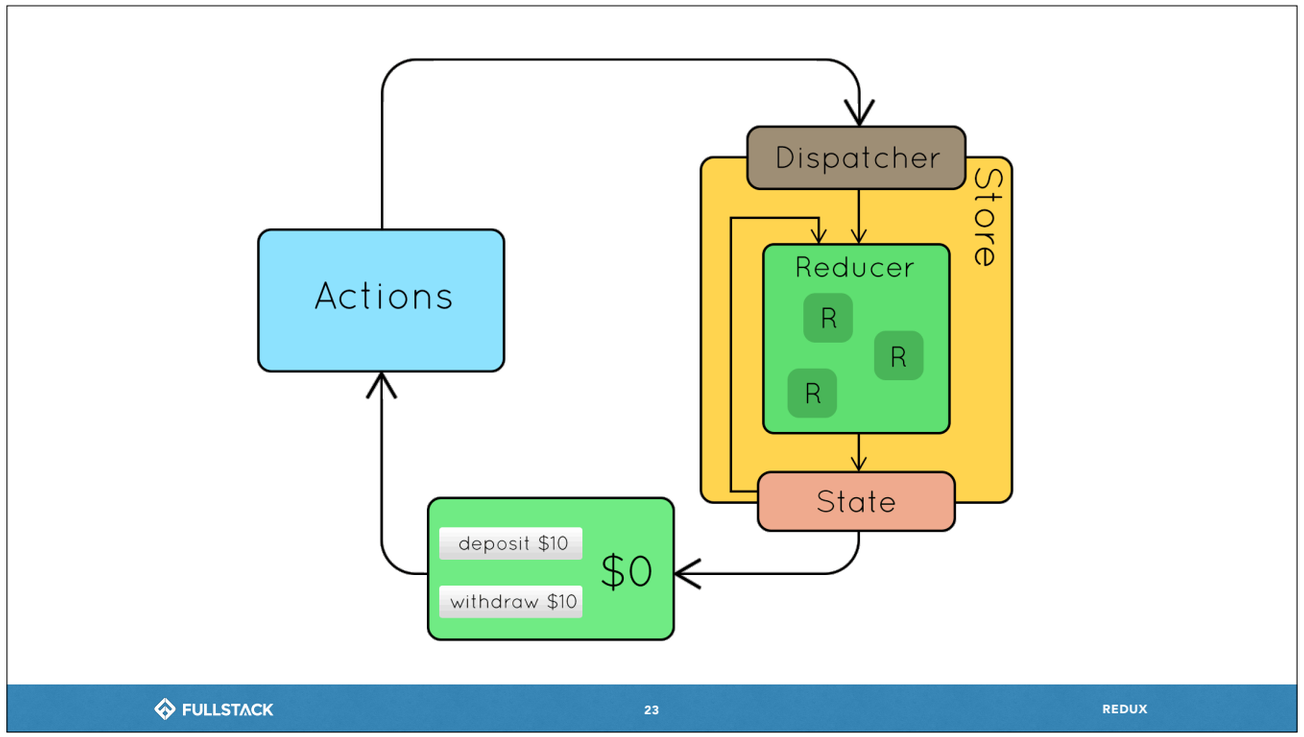
Note: If we perform the same transactions, same order, these results will be the same.

In Redux terms, the transactions on the left are our actions, and the balance on the right is the state that we would track in the store



If you ever find yourself wondering about your current bank balance, just trace back the list of transactions. The same is true for Redux - the biggest value it provides for developers is the ability to trace back all dispatched actions to understand how the state got into its current value.

Actually, understanding which pieces of the app caused changes in the state as an app grows is a frequent pain among developers. Redux helps cope with this.


Actions
Dispatcher
Store
Reducer
R
R
R
State
deposit $10
withdraw $10
$0

Before we finish, a small curiosity: Why is the function that updates the state called “reducer” instead of, say, “updater” or something?

(Actually, the name “updater function” was actually discussed when Redux was being developed)

ARRAY.PROTOTYPE.REDUCE

ARRAY SUM

```
const sum = [1, 2, 3].reduce((total, next) => {
  return total + next;
});

console.log(sum) // 6
```

“REDUCE” MULTIPLE VALUES INTO A **SINGLE VALUE**

REDUCER IN REDUX: TURN A SERIES OF ACTIONS INTO A SINGLE OUTCOME

Lab