REDUCE

YOUR NEW BEST FRIEND

JU GONÇALVES

- jugoncalv.es
- @cyberglot
- github/jugoncalves

AGENDA

- Functional Programming
- Abstracting List transformations
- Reducing everything
- Reduce in your everyday code

ES6/2015 CRASH COURSE

ARROW FUNCTIONS

```
var hi = () => console.log('hi')
```

means*

```
var hi = function () {
    return console.log('hi')
}
```

Array.from

```
var arr = Array.from(arrayLikeObject)
```

means*

```
var arr = Array.prototype.slice.call(arrayLikeObject)
```

MODULE PT. 1

```
export default function () {}
export var hi = function () {}
```

means*

```
module.exports = function () {}
module.exports = { hi: function () {} }
```

MODULE PT. 2

```
import { createRedux } from 'redux'
import * as stores from '../stores/index'
```

means

```
var createRedux = require('redux').createRedux
var stores      = require('../stores/index')
```

FIN

WHAT'S

(PROBABLY) THE MOST IMPORTANT COMPUTER PROGRAMMING CONCEPT?

```
                     ;-----------------------------------------------------------
                     ; zstr_count:
                     ; Counts a zero-terminated ASCII string to determine its size
                     ; in:   eax = start address of the zero terminated string
                     ; out:  ecx = count = the length of the string

                     zstr_count:                    ; Entry point
00000030 B9FFFFFFFF      mov  ecx, -1               ; Init the loop counter, pre-decrement
                                                    ;  to compensate for the increment
                     .loop:
00000035 41              inc  ecx                   ; Add 1 to the loop counter
00000036 803C0800        cmp  byte [eax + ecx], 0   ; Compare the value at the string's
                                                    ;  [starting memory address Plus the
                                                    ;  loop offset], to zero
0000003A 75F9            jne  .loop                 ; If the memory value is not zero,
                                                    ;  then jump to the label called '.loop',
                                                    ;  otherwise continue to the next line
                     .done:
                                                    ; We don't do a final increment,
                                                    ;  because even though the count is base 1,
                                                    ;  we do not include the zero terminator in the
                                                    ;  string's length
0000003C C3              ret                        ; Return to the calling program
```

```
function str_count (string) {
    return string.length
}
```

ABSTRACTION !

FUNCTIONAL PROGRAMMING

GUIDES US THROUGH THIS PATH

The functional programmer sounds rather like a mediæval monk, denying himself the pleasures of life in the hope that it will make him virtuous.

- John Hughes

YOU WRITE FUNCTIONS IN TERMS OF EACH OTHER

- abstract common patterns; and
- compose them

```
var getLoggedAPI = R.compose(
                        attendee,
                        event,
                        genApi,
                        R.path(['body', 'token']),
                        checkForErrors )

getLoggedAPI(genCredentials())
// => { token: ..., event: ..., attendee: ...}
```

LOOKS CONFUSING ?

LET'S START EASY

SUM

```
var sum     = 0
var numbers = [1,2,3,4,5,6,7,8,9,10]
var length  = numbers.length

for(var i = 0; i < length; i++){
    sum += numbers[i]
}

// => sum: 55
```

PRODUCT

```
var product = 1
var numbers = [1,2,3,4,5,6,7,8,9,10]
var length  = numbers.length

for(var i = 0; i < length; i++){
    product *= numbers[i]
}

// => product: 3628800
```

HM

IT LOOKS LIKE I HAD TO REWRITE A LOT OF STUFF

BUT WHAT HAS CHANGED?

```
var sum     = 0
var product = 1

sum     += numbers[i]
product *= numbers[i]
```

CONGRATS

YOU'VE LEARNT `reduce`

`reduce`

IS A HIGH-ORDER FUNCTION THAT RECEIVES:

- a function
- a initial value
- a list*

```
(init, val) => init
```

SUM AND MULTIPLY

```
var sum = (x, y) => x + y
var product = (x, y) => x * y

var numbers = [1,2,3,4,5,6,7,8,9,10]

reduce(sum, 0, numbers)
// => 55

reduce(product, 1, numbers)
// => 3628800
```

WAIT

HOW DOES IT WORK?

[1, 2, 3, 4, 5]

LET'S SAY THE SYMBOL : MEANS `concat`

1 : 2 : 3 : 4 : 5 : []

`reduce`*

REPLACES EACH

`:` WITH THE FUNCTION PROVIDED AND THE

`[]` WITH THE INITIAL VALUE

```
1 : 2 : 3 : 4 : 5 : []
1 + 2 + 3 + 4 + 5 + 0
1 * 2 * 3 * 4 * 5 * 1
```

FORMALLY,

IT IS ACTUALLY CALLED `fold` OR `foldRight`. IN JAVASCRIPT, THIS BEHAVIOUR IS SIMULATED IN `Array.prototype.reduce`.

`Array.prototype.reduceRight` is the same as `[].reverse().reduce`.

RIGHT

```
(1 * (2 * (3 * (4 * (5 * 1)))))
```

LEFT

```
(((((1 * 1) * 2) * 3) * 4) * 5)
```

"OK

BUT I DON'T NEED TO SUM AND MULTIPLY THINGS VERY OFTEN, SO"

#NOT IMPRESSED

- map
- filter
- and

```
var cons = (item, list) => list.concat(item)

var map = (fn, list)
    => reduce(
        (init, element) => cons(fn(element), init)
        [],
        list)
```

```
var cons = (item, list) => list.concat(item)

var filter = (predicate, list)
    => reduce(
        (init, element) => predicate(element) ? cons(element, init) : init,
        [],
        list)
```

```
var and = ()
    => reduce(
        (init, element) => init && element,
        true,
        Array.from(arguments))
```

```
var or = ()
    => reduce(
        (init, element) => init || element,
        false,
        Array.from(arguments))
```

NOT ONLY LISTS

```
var insert = (key, value, object) => {
    var newObj = Object.create(object)
    newObj[key] = value
    return newObj
}

var omap = (f, obj)
    => reduce(
        (init, key) => insert(key, f(obj[key]), init),
        {},
        Object.keys(obj))
```

```
var compose = () => {
    var fns = Array.from(arguments)
    return (value) =>
        reduceRight(
            (init, element) => element(init),
            value,
            fns)
    }
```

AND WHAT ABOUT PROMISES?

Save depending data sequentially.

```
var save = (noDeps, deps)
    => reduce(
        (init, element) => init.then(() => element.save()),
        q.all(noDeps(dep => dep.save()),
        deps)
```

Compose functions to transform data within a Promise.

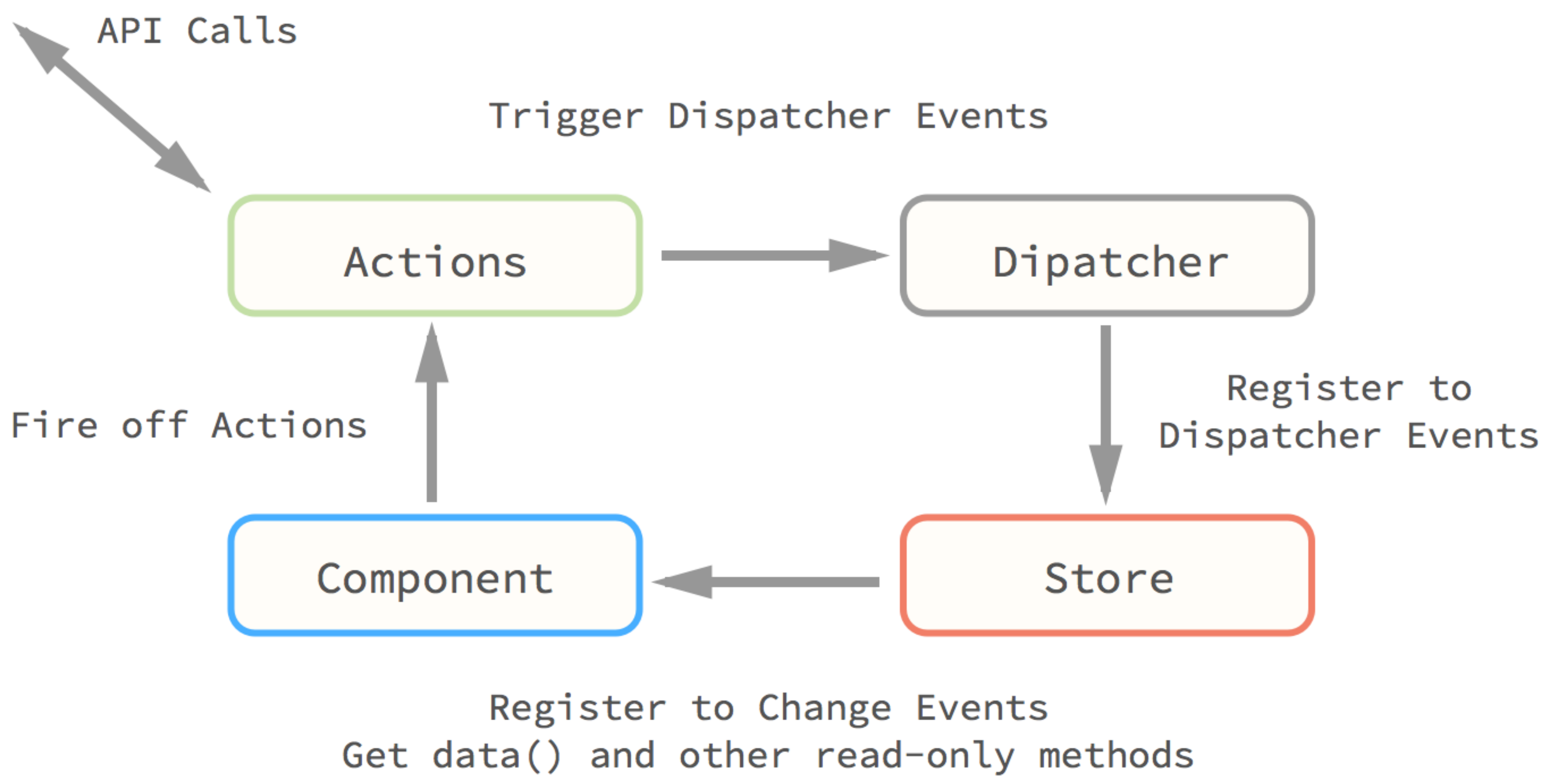
```
var composeP = () => {
    var fns = Array.from(arguments)
    return (promise)
        => reduceRight(
           (init, element) => init.then(element),
           promise,
           queries)
    }
```

ramda has a `composeP` function

HOW ABOUT

USING REDUCING FUNCTIONS IN YOUR EVERYDAY CODE?

FLUX ARCHITECTURE


API Calls
Trigger Dispatcher Events
Actions
Dipatcher
Register to
Dispatcher Events
Fire off Actions
Component
Store
Register to Change Events
Get data() and other read-only methods

from 'Transitioning to Flux Architecture'

STORES
TRANSFORM
DATA

```
var fluce = require('fluce/create-fluce')()

fluce.addStore('counter', {
  initial() {
    return 0
  },
  reducers: {
    counterAdd(init, x) {
      return init + x
    },
    counterSubtract(init, x) {
      return init - x
    }
  }
})

fluce.actions.dispatch('counterAdd', 10)
// => fluce.stores.counter : 10
```

from rpominov/fluce

```
(state, action) => state
 (init, val)    => init
```

```
export default function counter(state = 0, action) {
  switch (action.type) {
  case INCREMENT_COUNTER: return state + 1
  case DECREMENT_COUNTER: return state - 1
  default: return state
  }
}

// app.js

import { createRedux } from 'redux'
import * as stores from '../stores/index'

var redux = createRedux(stores)
```

from gaearon/redux

THANKS !

- @cyberglot
- github/jugoncalves