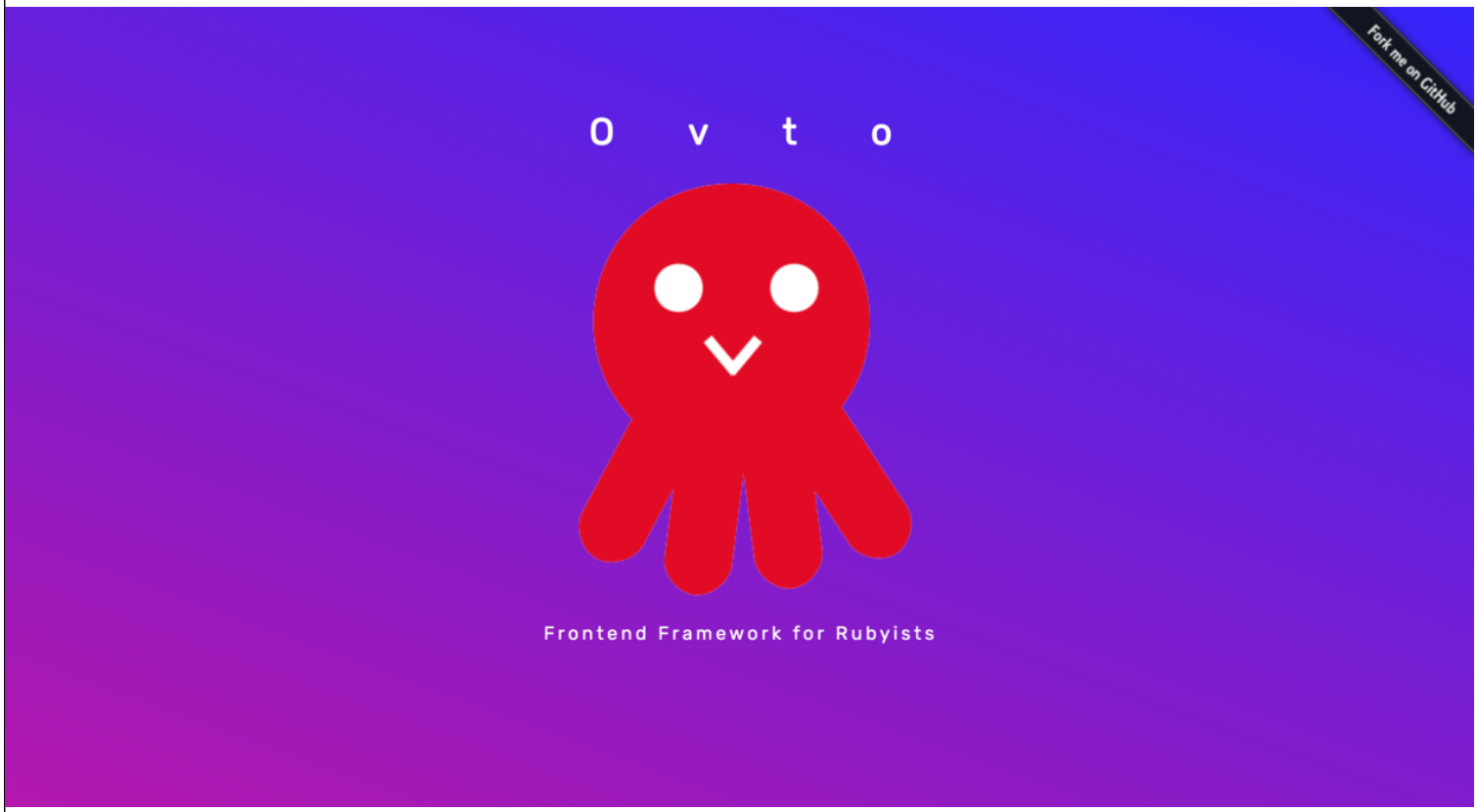
Ovto: Frontend web framework for Rubyists

- Yutaka HARA (@yhara)
- RubyKaigi2019Fukuoka
- 2019/04/19

http://yhara.github.io/ovto



~ こんにちは。今日は私が作ったOvtoというフレームワークの紹介をします。 ~ これはOvtoの公式ページです。

Ovto is (1/3)

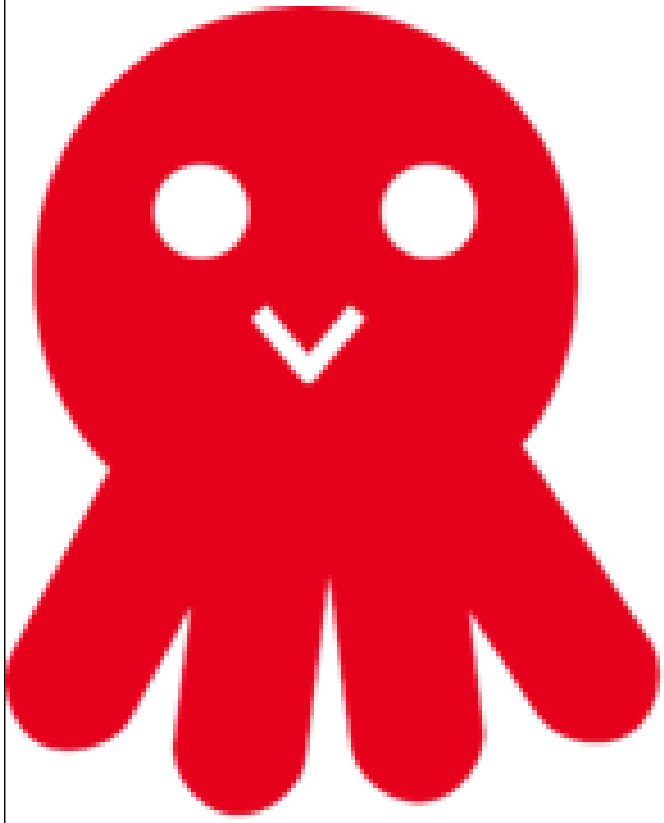
- Frontend web framework in Ruby
- Powered by Opal
- VirtualDOM + Single state (like react-redux)

~ もしOvtoを15秒で説明するとしたらこうなります。 ~ RubyでSPAを作れるフレームワークで、Opalを使っています。 ~ react-reduxみたいにVirtualDOMとSingle stateを備えています。

~ という説明で分かればいいんですが、そういう人は少ないですよね？大丈夫、残りの時間で説明します。

Ovto is (2/3)

- 'Octo' is already taken everywhere



Ovto is (3/3)

- DEMO: http://rk2019.yhara.jp

Agenda(1/3)

- This talk does not cover:
  - Difference with Angular, Vue, etc.
  - How to use Ovto
    - en: book/guides/tutorial.md
    - ja: "RubyistのためのフロントエンドフレームワークOvto" (Rubyist Magazine 0059)

~ このトークでは、AngularやVueとかとどう違うのかという話はしません。私はこれらを使ったことがないので、無理に話そうとしても良い話はできないだろうと考えたからです。 ~ またOvtoの詳しい使い方の紹介もしません。すでに英語と日本語でチュートリアルを書いたからです。

Agenda(2/3)

- This talk covers:
  - My thoughts on making Ovto
  - Knowledge you need to use Ovto

~ 今日は代わりに、チュートリアルの補足をします。 ~ 1つは、Ovtoを作るときに私が何を考えていたのかについて話したいと思います。 ~ また、先程のチュートリアルを読むための前提知識を提供することも意図しています。VirtualDOMやSingle Stateって何、という話ですね。

Agenda(3/3)

- Part 1: Background
- Part 2: Design
- Part 3: Implementation

~ 今日は三部構成でやろうと思います。１つ目は背景つまり「なぜOvtoを作ったのか」、２つ目は「Ovtoの設計思想」、３つ目は「Ovtoの実装」です。

Today's goal

- Find the 2nd user

~ 今日のゴールは、Ovtoの2人目のユーザを見つけることです。私自身は、最高のフレームワークができたと思っていて、趣味で何かを作るときはOvtoを使うつもりですが、Ovtoが私以外の約にも立てばいいなと思っています。

Myself

- @yhara (Yutaka HARA)

~ 改めましてyharaです。 ~ 私はNaCl、ネットワーク応用通信研究所という会社で働いています。 ~ NaClはRubyの作者であるまつもとさんがいる会社です。 ~ といっても今では珍しくないですが、1997年、Ruby1.1の頃からフェローをされている、老舗のRuby屋さんです。

Ad: RubyDebugCheatSheet

- NaCl @ 2F

~ 今回、「最もRubyらしいノベルティ」として、Rubyのデバッグ技法を紹介するチートシートを作りました。ぜひブースにいらしてください。

Past Kaigi Talks

- n-ways to distribute your Ruby apps (RubyKaigi2009)
- Road to Ruby Master (RubyKaigi2011)
- DIY Programming (SapporoRubyKaigi2012)
- Let's Make a Functional Language (RubyKaigi2015)
- Ruby, Opal and WebAssembly (RubyKaigi2017)
- Ovto: Frontend web framework for Rubyists (RubyKaigi2019)

~ これらは、私の過去の発表です。RubyKaigiで発表するのは今日が6回目くらいになります。だいたい2年に一回くらい話させていただいているんですが：

My interest

- DIY Programming
- Language processor

~ 私の興味としては「自給自足」、つまり自分で使うソフトウェアを自分で作るという行為と、と「言語処理系」があります。今回は前者の話ですね。

Part 1. Why I made Ovto

~ ということで、なぜOvtoを作ったのかという話から始めましょう。

DIY Programming

- Make software you need
- Ruby is very good for it
    - Supports many areas
    - Good for prototyping

~ 自給自足プログラミングというのは、自分が欲しいものを自分で作るということです。 ~ Rubyという言語はそれにとても向いています。 ~ 理由はいくつかありますが、一つはいろんなものを作れるということ。ワンライナからWebアプリまで。

~ もう一つはプロトタイピングに向くということですかね。今でこそ仕事で使う言語になりましたけど、もともとはスクリプト言語と言って、「ちゃんとしてないもの」を作るのに向いた言語だったんです。料理でいえば、冷蔵庫の残り物で夕飯をさっと作るようなね。

My DIYs(1/3)

yhara/fooods

Made with Sinatra

~ ということで私が自分のために作ったものをいくつか見せたいと思います。 ~ 一個目は、食材の賞味期限をメモるやつですね。Sinatraで作っています。

My DIYs(2/3)

~ 二個目は私のブログです。これもSinatraですね。

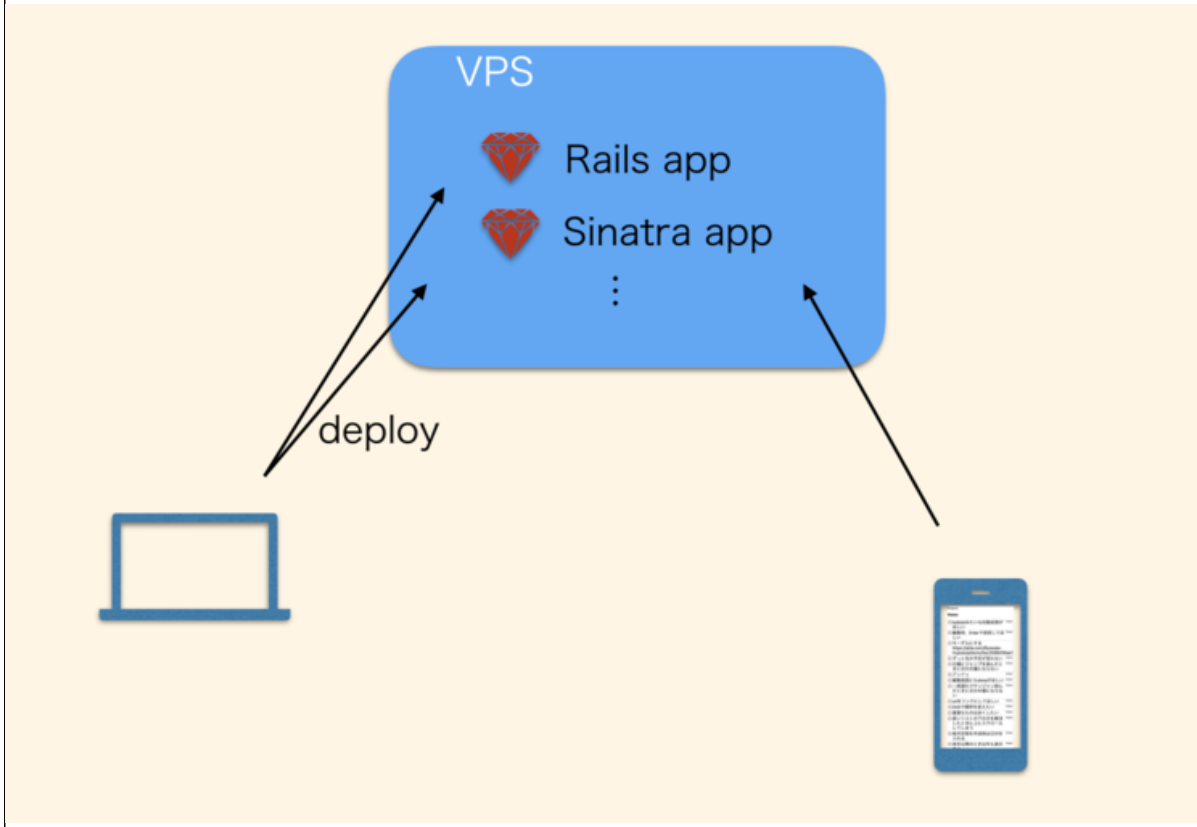
My DIYs(3/3)

yhara/monolist

Made with Rails

~ 3つ目は家じゅうの「物」を管理するためのシステムです。これはRailsです。

Web app is mobile app



~ で、これらはWebアプリなんですけど、モバイルアプリでもあるんですよ。 ~ どういうことかというと、さくらのVPSを年1万くらいで借りていて。そこにcapistranoでデプロイしている。そうすると自分のスマホから操作できるので、スーパーで買い物してるときに賞味期限を確認したりできる。

Mobile app is hard

- Swift or Kotlin?
- Obj-C or Java?
- Evolves to quickly (at least so far)

~ もちろんプログラマなので、モバイルアプリを自分で作ることもできます。けど大変なんですよね。iPhoneとAndroidで全然違うし。

~ 人々がスマホを使い始めた頃、「これからはモバイルアプリの時代だ」という雰囲気があったんですが、僕はあえてモバイルアプリはやらなかったんですね。プログラミングでできることってめちゃめちゃに多いので、全部をマスターすることはできないので、どの分野をやるのかは自分で選ぶ必要がある。

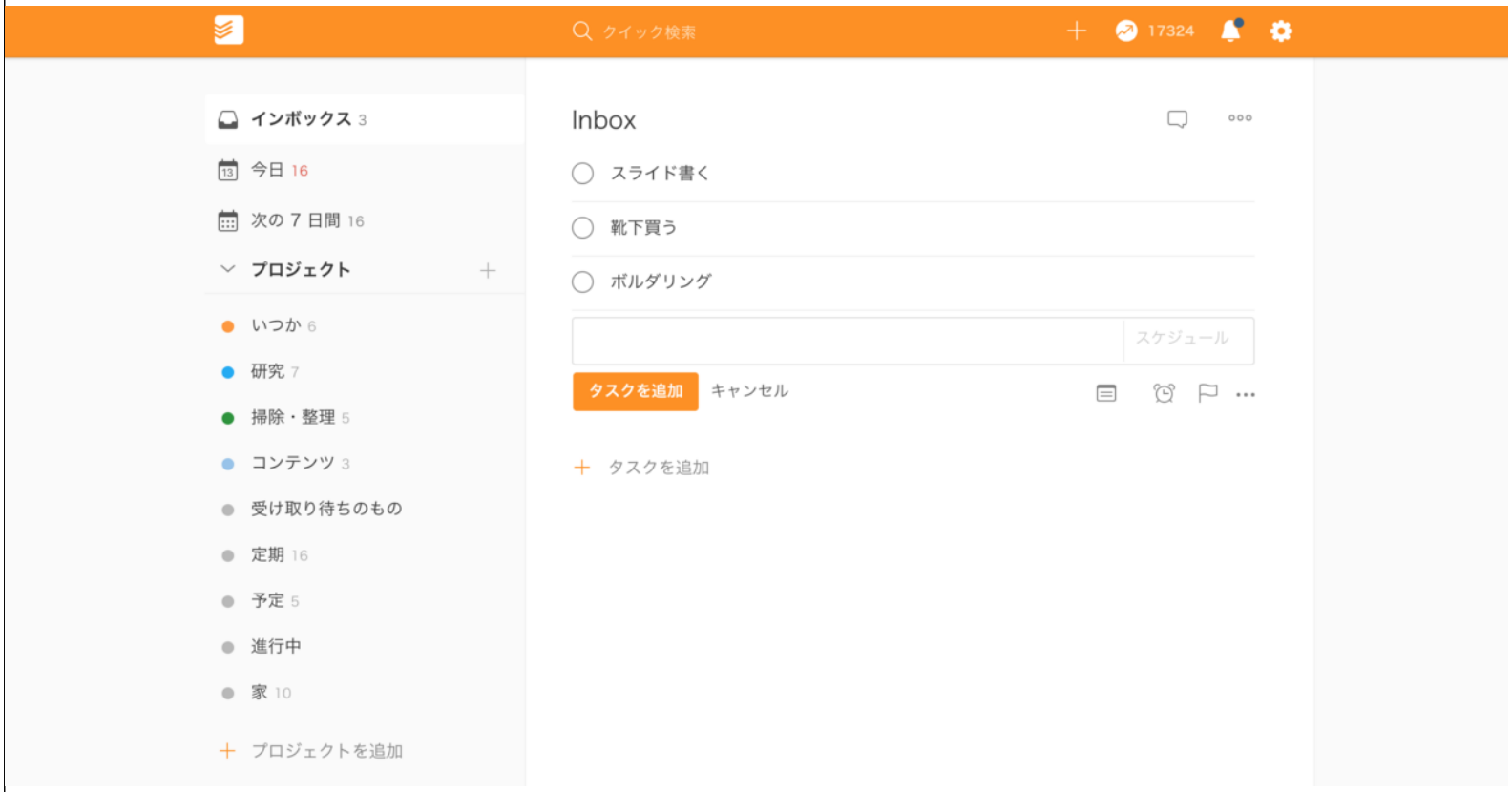
Web tech is everywhere

- Mobile web
- Mobile app (PhogeGap, etc.)
- Desktop app (Electron, etc.)

~ 僕は、その代わりにWeb技術の勉強に時間を使おうと思いました。Webを覚えれば、「なんちゃってモバイルアプリ」も作れる。自分用としては十分です。

But...

- How to make this kind of app?



~ そうはいってもRubyだけでは作りづらいものも出てきました。例えばTODOアプリのtodoistというのがあります。こういう凝ったものを生のJavaScriptとかjQueryでやろうとすると、コードがぐちゃぐちゃになってしまいがちです。

jQuery is not enough(1/2)

- You specify 'where' and 'what'

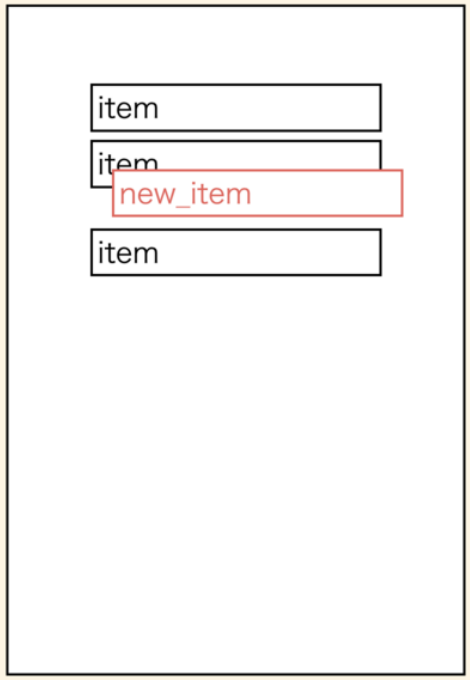
```
$(elem).append(item);
$(elem).prepend(item);
$(elem).remove();
$(item).insertBefore(elem);
...
```

~ なぜでしょうか？

~ 例えば、ボタンを押したら1行追加する、という例を考えてみましょう。 ~ jQueryだとこんな風に簡単に書けます。 ~ でもこの方法は、DOMの差分を記述しているといえます。ということは、「もとの状態」「変更する場所」「変更内容」の全部が正しくないといけない。

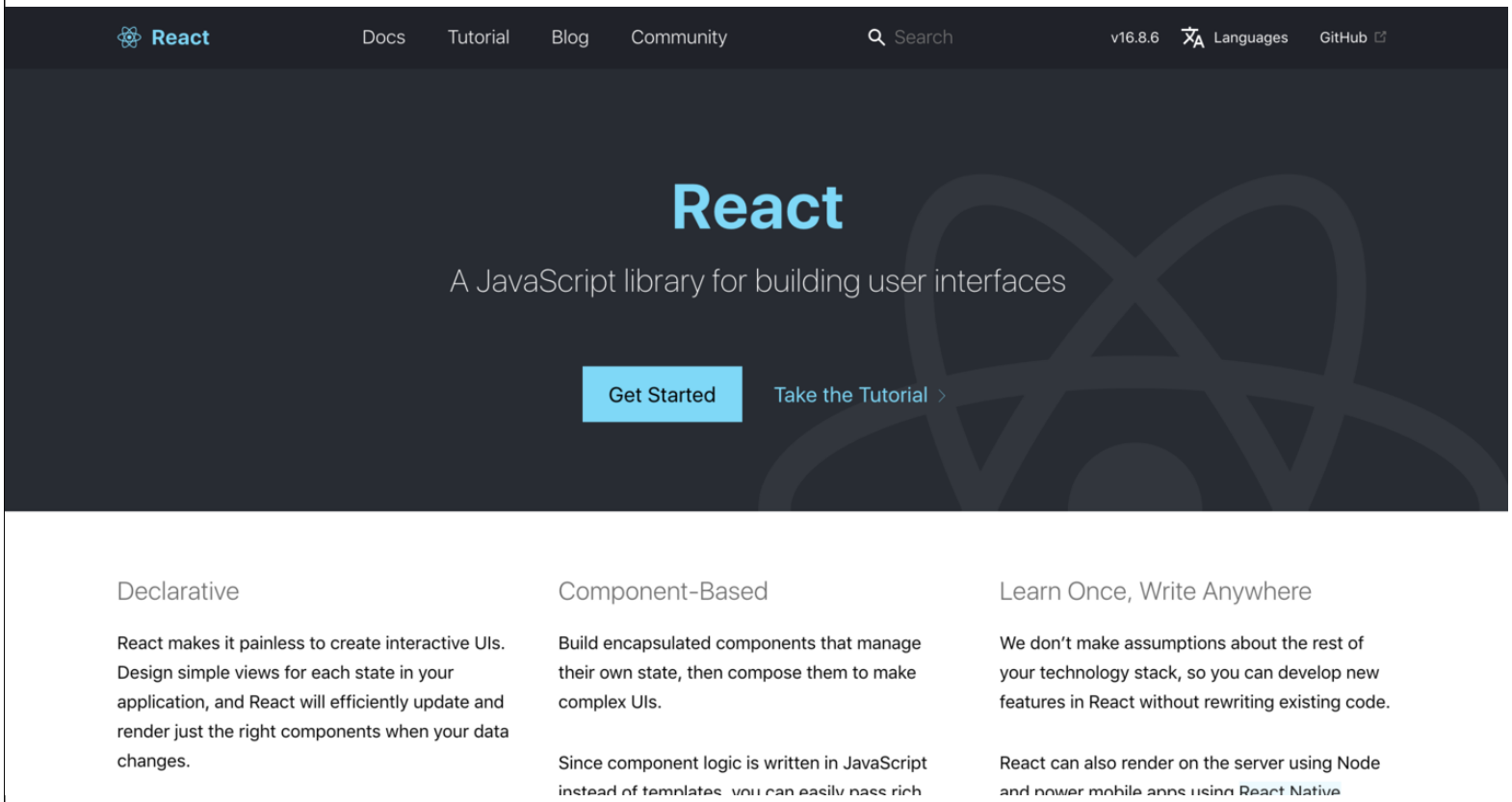
jQuery is not enough(2/2)

```
$(new_item).insertAfter(item[1]);
```



~ たとえば、リストに要素を追加するという例を考えましょう。jQueryだとこんな感じでしょうか？ ~ 「1番目のあとに要素を追加する」という書き方になりますね。要素を追加する場所をプログラマが指定しないといけない。これは、データを非同期でサーバから取ってきたりすると、難しくなります。

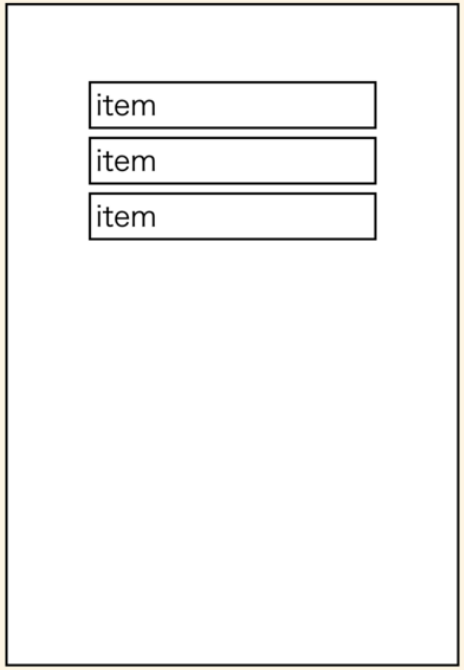
Here comes VirtualDOM(1/6)



~ 最近流行りのReactは、この問題を解決しています。

Here comes VirtualDOM(2/6)

```
<ul>
    {this.state.items.map((item) => {
          return <li key={item.id}>{item.text}</li>;
    })}
</ul>
```

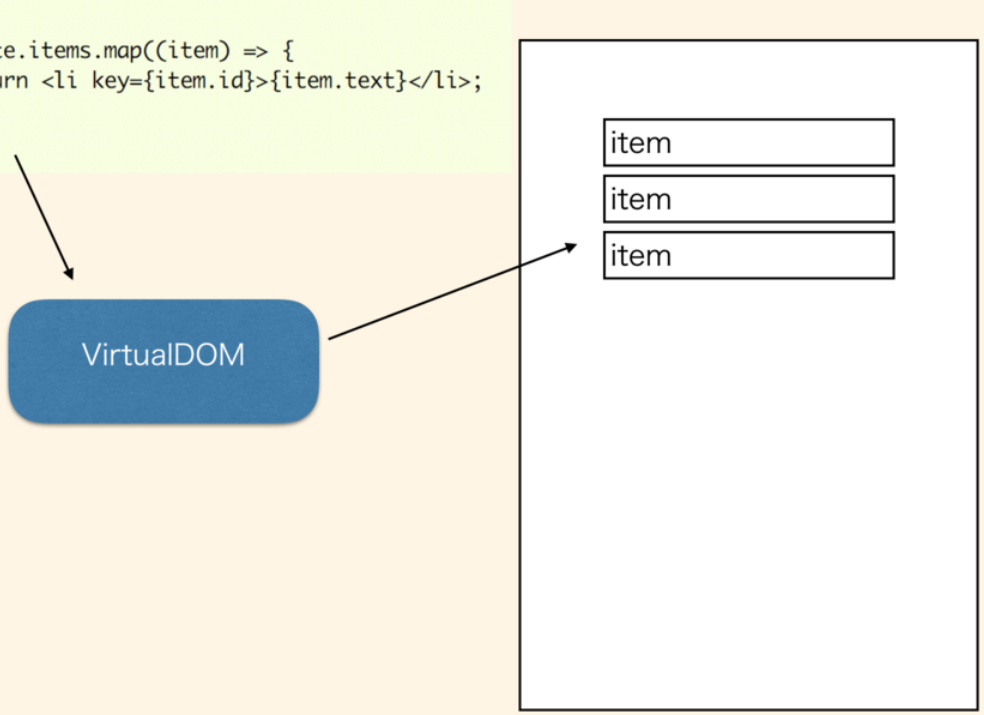


~ これはReactのコードです(読まなくていいです)。これを見ると、要素が増えたら・減ったらどうするというのは書いてなくて、「要素の数だけタグを作れ」と書いてある。これなら、どこに挿入するかとか考えなくていいので楽ですね。

~ でもこれだけだと、要素を一個増やしただけで全体が再描画されて遅そうですよね？

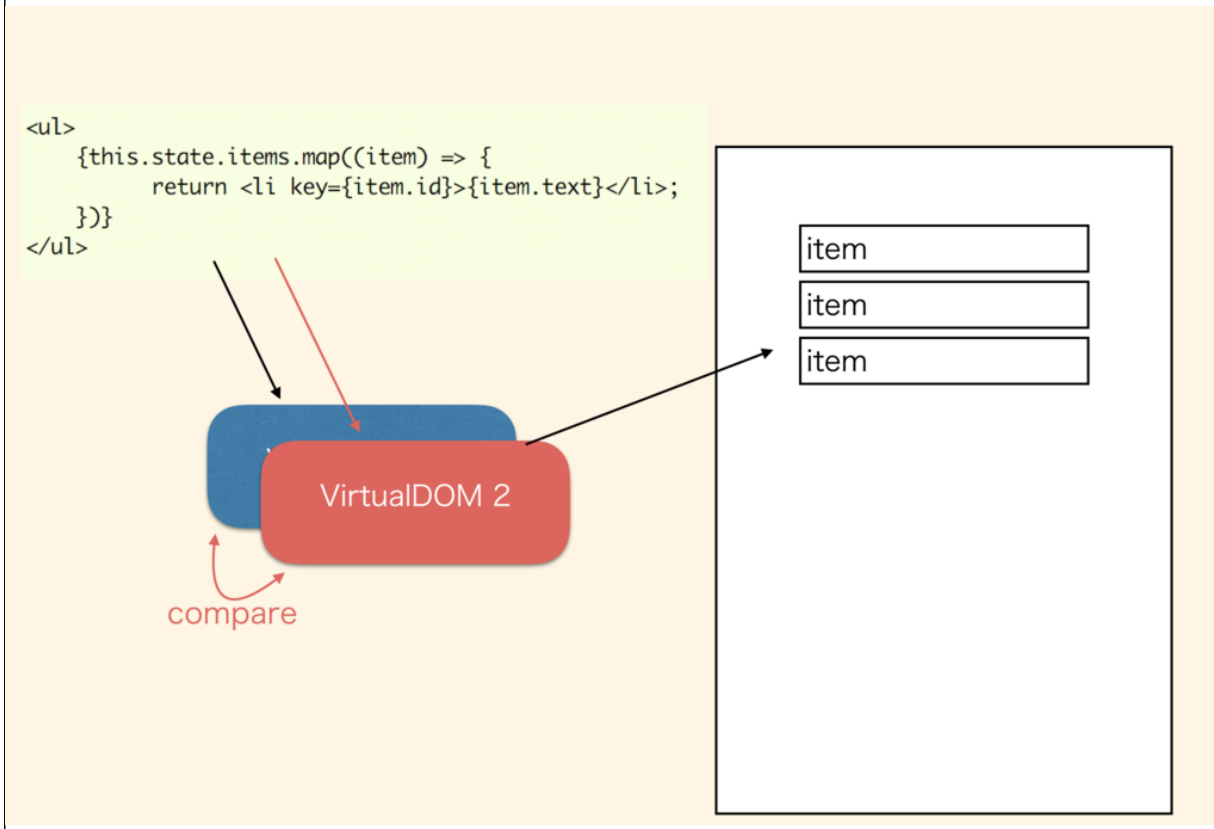
Here comes VirtualDOM(3/6)

```
<ul>
    {this.state.items.map((item) => {
        return <li key={item.id}>{item.text}</li>;
    })}
</ul>
```



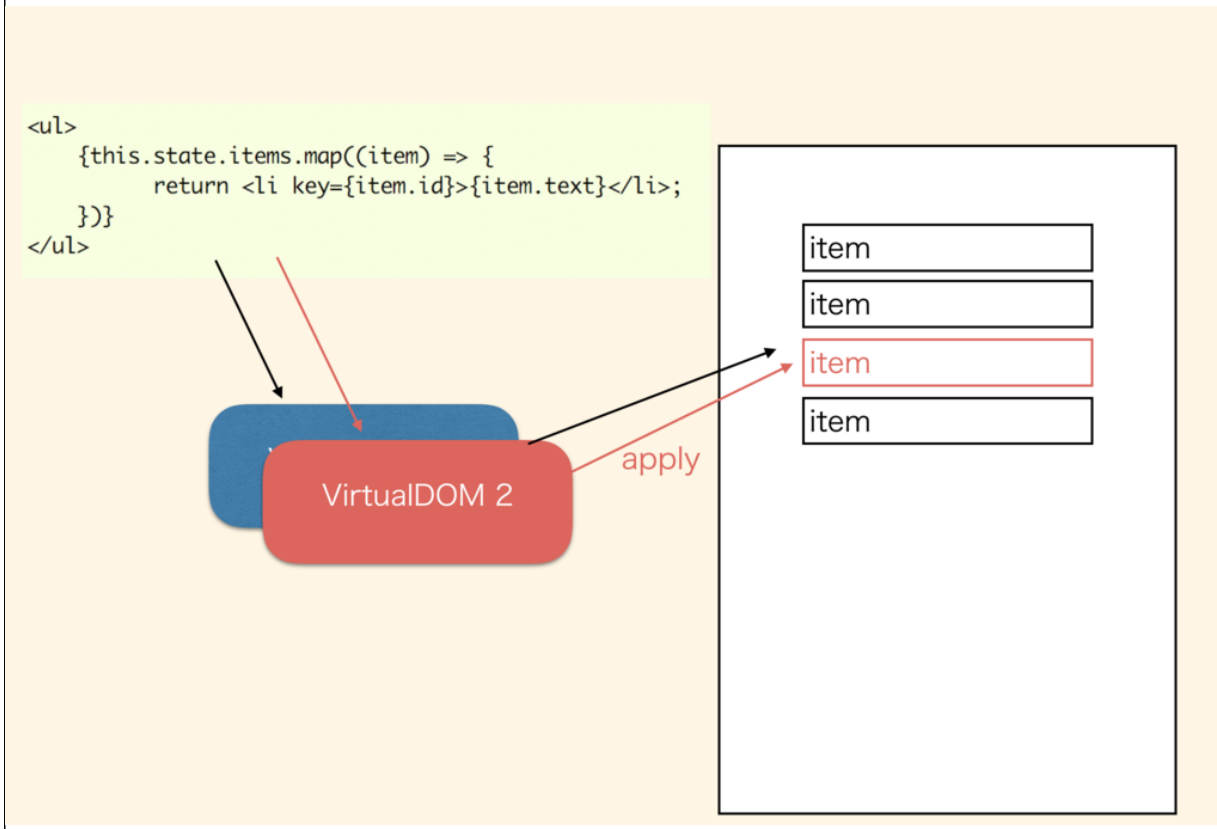
~ ところがこれが遅くならないような仕組みがReactにはあります。VirtualDOMです。 ~ このコードが直接タグを作るのではなく、いったんVirtualDOMというものを作ります。 ~ VirtualDOMは、「こういうタグを作りたい」というのをJSONで記述した、DOMの設計図のようなものです。

Here comes VirtualDOM(4/6)



~ 要素が増えたときは、まずコードを動かして新しいVirtualDOMを作ります。そして、もとのVirtualDOMとどこが変わったかを調べます。 ~ VirtualDOMは単なるメモリ上のオブジェクトなので、高速に比較できます。

Here comes VirtualDOM(5/6)



~ そして、変化があれば変わったところだけを自動で適用してくれます。

Here comes VirtualDOM(6/6)

- You write "What it should be"
- React calculates & applies the difference automatically

~ まとめると、VirtualDOMは「こうしたい」を記述すると、あとは自動でやってくれる仕組みです。楽だし、遅くない。

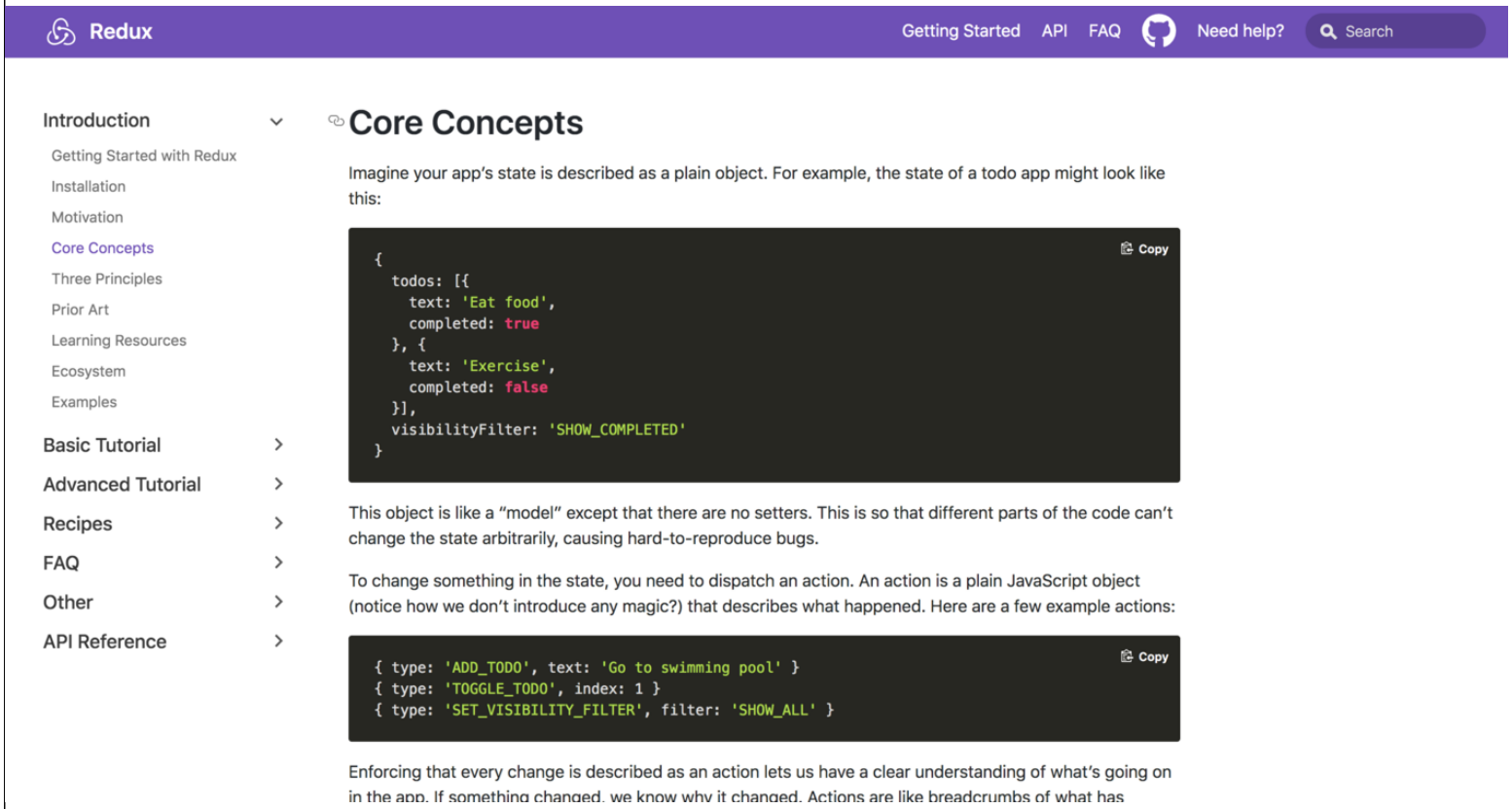
React and me

- 3 month exprerience of react-redux at work

~ 次に、Single Stateの説明をします。 ~ Reactはあまり使ったことなかったんですが、あるとき仕事で三ヶ月ほどreact-reduxを触る機会がありました。

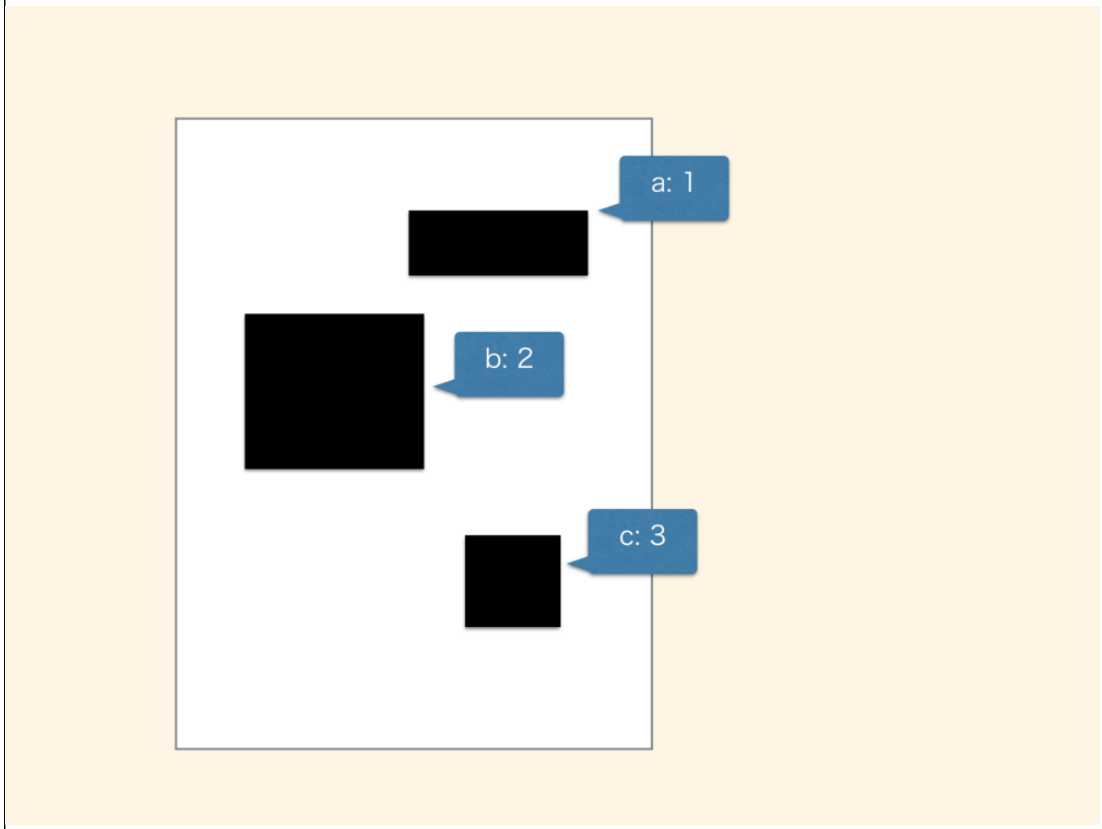
react-redux

- Make React to be "Single State"



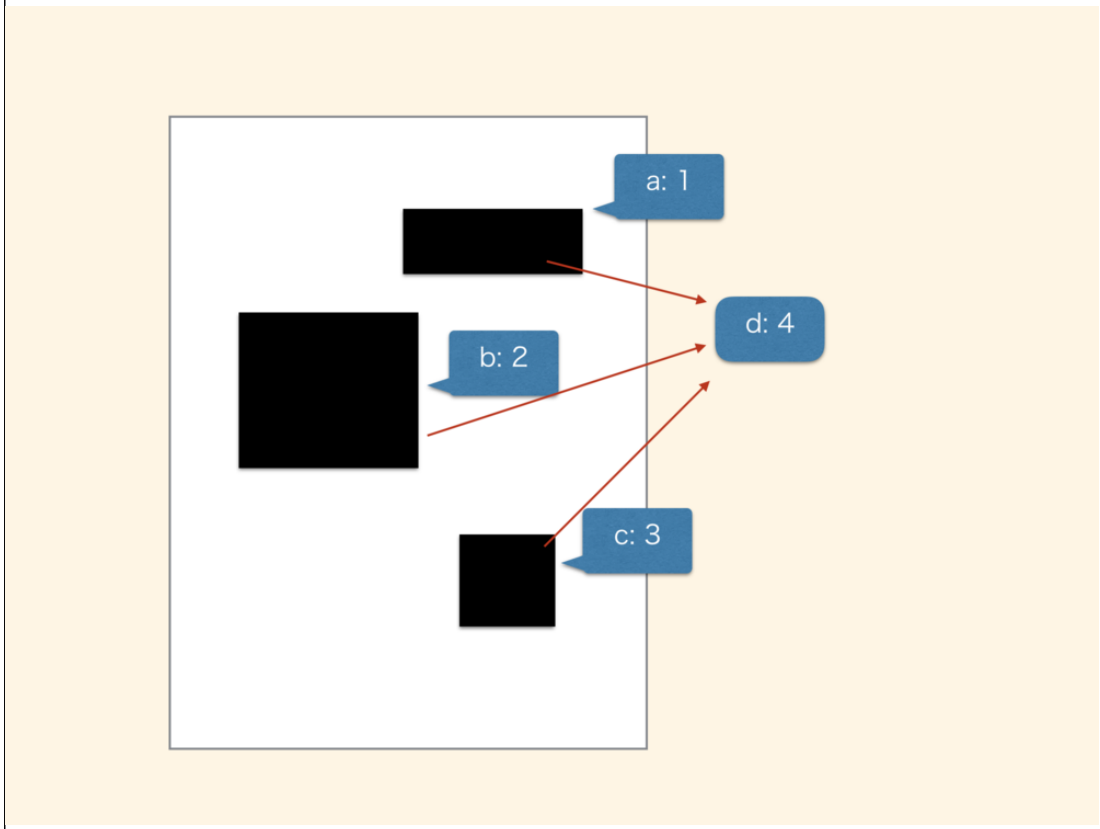
~ react-reduxはReactの上に乗せるフレームワークで、これを使うとReactがシングルステートになります。

Single state (1/4)



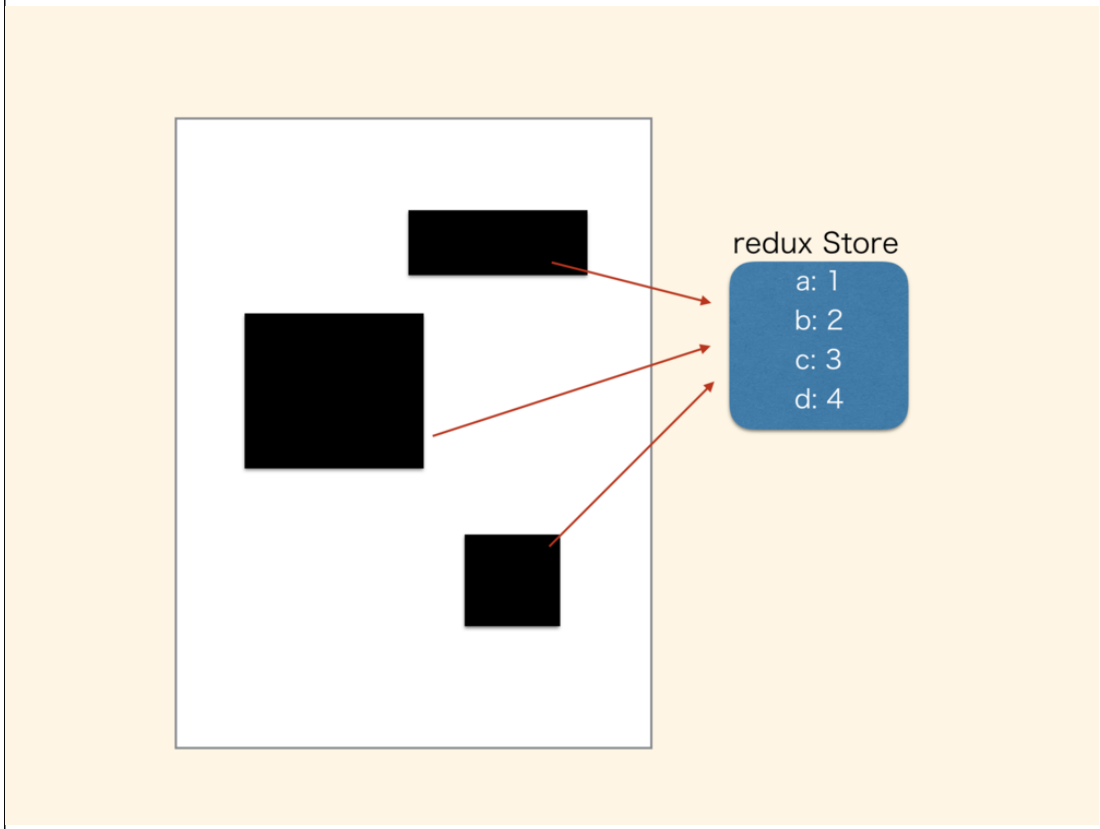
~ シングルステートとは何か。普通のReactアプリでは各コンポーネントがそれぞれ状態を持っています。

Single state (2/4)



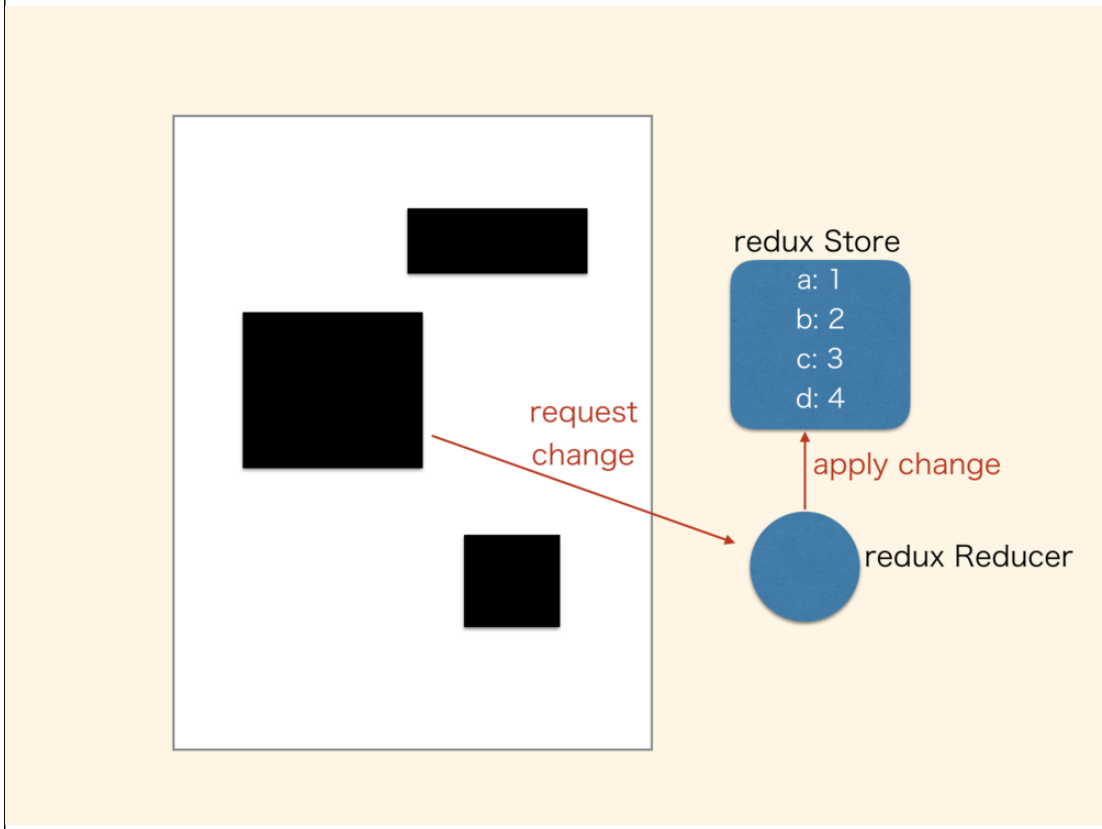
~ 簡単なアプリならいいんですが、コンポーネントをまたがった状態があるとか、それを非同期に更新するとか、そういうことをするとスパゲッティになる。

Single state (3/4)



~ reduxはこれを解決するためのもので、状態を一箇所にまとめてしまいます。アプリケーションの全部の状態を持ったStoreというものがあって、各コンポーネントはここから必要な値を取り出します。

Single state (4/4)



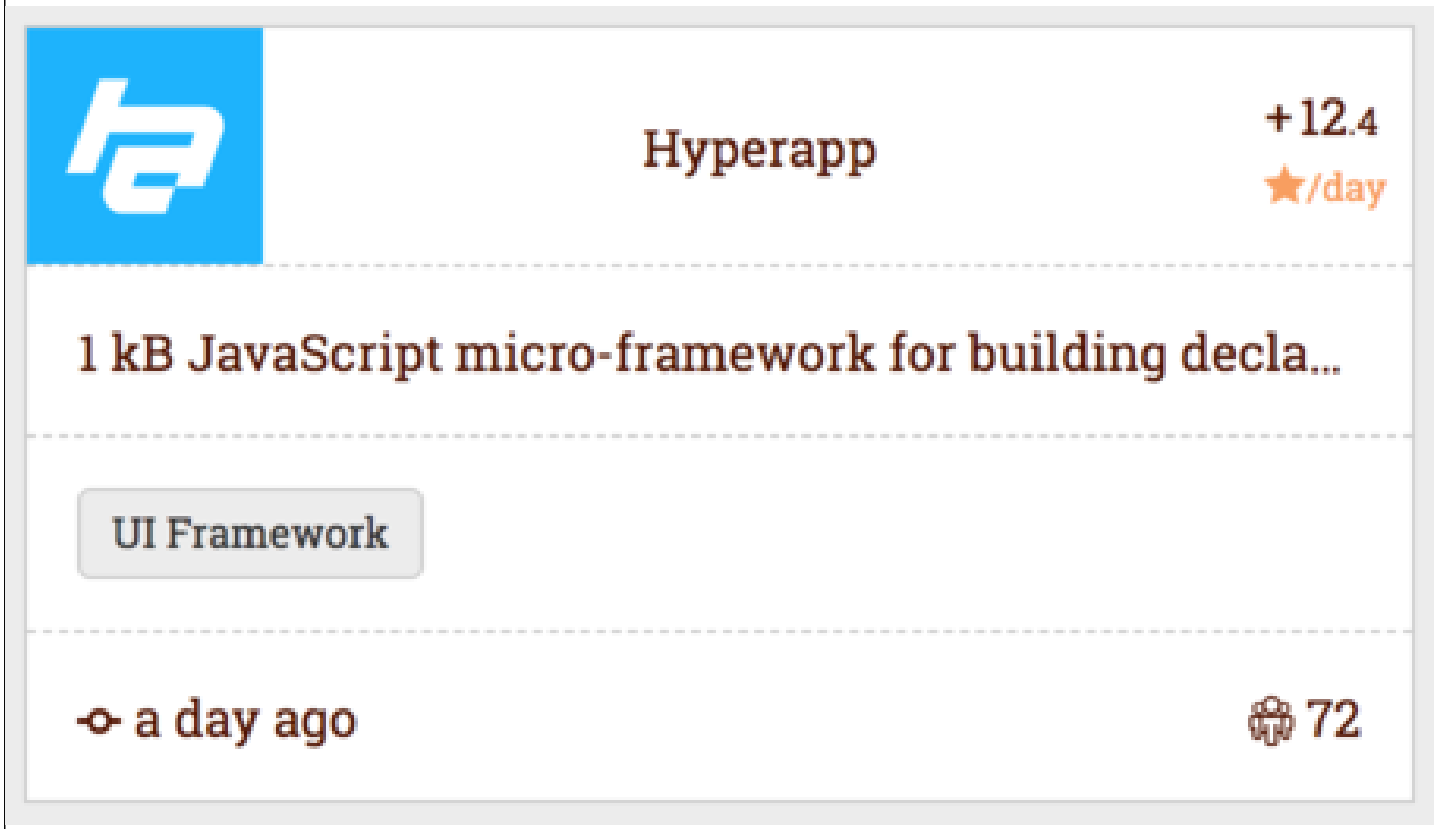
~ で、状態を更新するときもコンポーネントが勝手に書き換えてはいけない。reducerという、変更管理人みたいなオブジェクトがいて、何かを変更したいときはこいつに変更依頼を飛ばします。

~ しばらく使いましたが、なかなかいいものだと思いました。でもreduxって構成要素が多くてけっこうとっつきにくいんですよね。

And then:

~ そんな中で、hyperappというフレームワークがあるのを知りました。このQiitaの記事ですね。

hyperapp



~ hyperappはreact-reduxと似たようなVirtualDOMとSingle State機能を持つのに、たった数百行しかない。すごい。

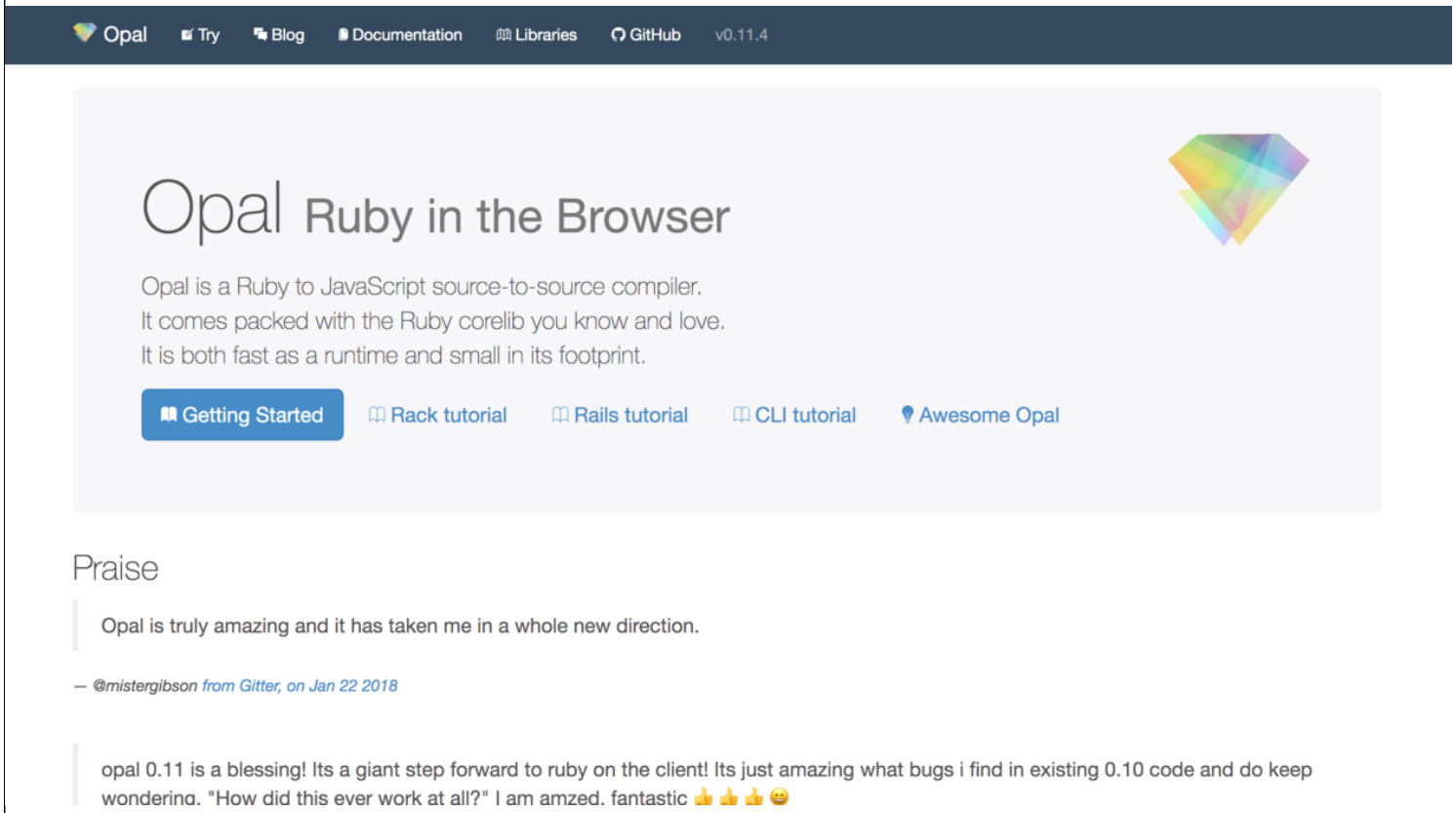
IDEA: Port hyperapp to Ruby

~ そこでふと思いました。これのRubyバージョンが作れるんじゃないか？と。

to Ruby?

- They say "You need to write JS to make an SPA"
- Actually it's not true

~ Rubyバージョンを作る、とはどういうことでしょうか。~ もしかしたら、ブラウザで動くものを作るにはJavaScriptを書かないといけない、と教わったかもしれません。~ でも実はそうじゃないんです。



~ Opalというものを使うと、Rubyのソースコードを等価なJavaScriptに変換することができます。

Opal (1/2)

```ruby
class A
  def hello
    "Hello"
  end
end
```

~ 例えばこのようなRubyコードがあるとします。

Opal (2/2)

```
  self.$require("corelib/random");
  return self.$require("corelib/unsupported");
})(Opal);

/* Generated by Opal 0.11.4 */
(function(Opal) {
  var self = Opal.top, $nesting = [], nil = Opal.nil, $breaker = Opal.breaker, $slice = Opal.slice, $klass = Opal.klass;

  Opal.add_stubs(['$puts', '$join']);

  (function($base, $super, $parent_nesting) {
    function $A(){};
    var self = $A = $klass($base, $super, 'A',

    var def = self.$$proto, $nesting = [self].

    return (Opal.defn(self, '$hello', TMP_A_he
      var self = this;

      return "Hello"
    }, TMP_A_hello_1.$$arity = 0), nil) && 'hello'
  })($nesting[0], null, $nesting);
  return self.$puts(["Hello, ", "world!"].$join());
})(Opal);

/* Generated by Opal 0.11.4 */
(function(Opal) {
  var self = Opal.top, $nesting = [], nil = Opal.nil, $breaker = Opal.breaker, $slice = Opal.slice;
```

```
class A
  def hello
    "Hello"
  end
end
```

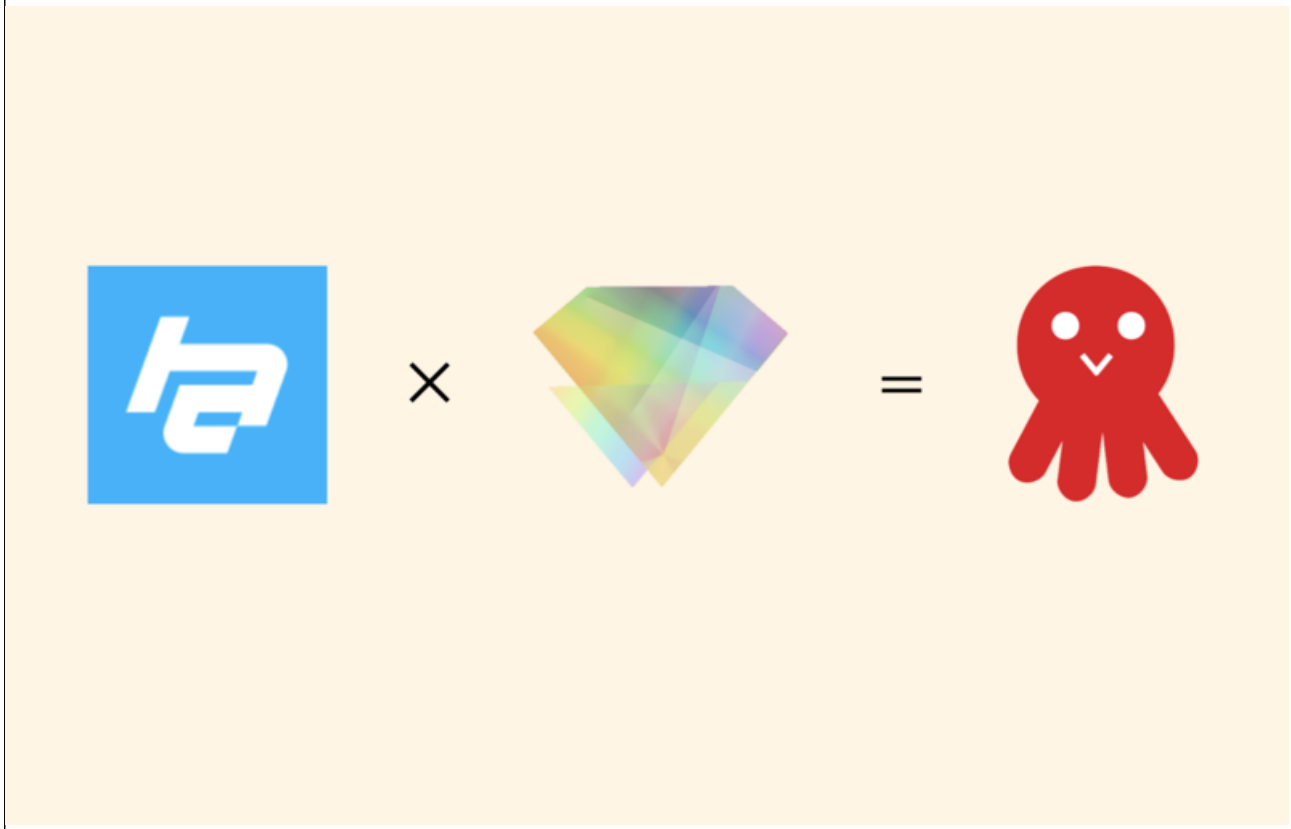
~ これをOpalにかけると、このような、同じ動作をするJavaScriptのコードを生成してくれます。これを使うと、Rubyで書いたものをブラウザで動かすことができる。

Why not JS?

- Ruby is my native tongue
- Minor but irritating difference
    - No Array#==, etc.

~ もちろん「そこまでしてRubyで書かないといけないのか？」という話はあるでしょう。 ~ いけなくはない。いけなくはないが、私にとってはRubyが一番速く書けるんです。 ~ JavaScriptもどんどん進化して便利な言語になっていますが、Rubyとは微妙に違いがありますよね。例えば配列がイコールで比較できないとか。

Ovto



~ というわけで、hyperappをRubyに移植したのがOvtoです。

(Notices)

- Based on hyperapp v1
- Not a 1-to-1 port
    - Redesigned to be "Rubyish" API

~ とはいえ、移植と言ってしまうと少し語弊があって。 ~ まず、hyperappはバージョン1と2があって結構違うんですが、Ovtoが参考にしたのはv1の方です。 ~ もう一つ、hyperappの完全コピーを目指したわけではなく、Rubyらしく書けるようにいろいろ変えています。

hyperapp example

```
import { h, app } from "hyperapp"

const state = {
  count: 0
}

const actions = {
  down: value => state => ({ count: state.count - value }),
  up: value => state => ({ count: state.count + value })
}

const view = (state, actions) => (
  <div>
    <h1>{state.count}</h1>
    <button onclick={() => actions.down(1)}>-</button>
    <button onclick={() => actions.up(1)}>+</button>
  </div>
)

app(state, actions, view, document.body)
```

~ これは、hyperappのサンプルコードです。 ~ これを単純にRubyに置き換えるとするとどうなるでしょうか？

Ruby port?

```
...

state = {count: 0}

actions = {
  down: ->(value){ ->(state){ {count: state.count - value} } },
  up:   ->(value){ ->(state){ {count: state.count + value} } },
}

view = ->(state, actions){
  h("div", {}, [
    h("h1", {}, state.count),
    h("button", { onclick: ->{ actions.down(1) }, "-"),
    h("button", { onclick: ->{ actions.up(1) }, "+"),
  ])
}

...
```

~ こう？

~ いやあ、なんか変ですね。Rubyらしくない。 ~ なんでかというと、hyperappのAPIはJavaScriptの関数をベースにしてるんですね。だから、単純に変換しただけだとProcオブジェクトがいっぱい出てきて、Rubyとしては不自然なプログラムになる。

The question is...

- What the sample program should look like?

~ つまり、「どう書きたいか？」をまず考えないといけない。

In Ovto

```
require 'ovto'

class MyApp < Ovto::App
  class State < Ovto::State
    item :count, default: 0
  end

  class Actions < Ovto::Actions
    def down(state:, amount:)
      return {count: state.count - amount}
    end

    def up(state:, amount:)
      return {count: state.count + amount}
    end
  end

  class MainComponent < Ovto::Component
    def render(state:)
      o "div" do
        o "h1", state.count
        o "button", {onclick: ->{ actions.down(amount: 1) }}, "-"
        o "button", {onclick: ->{ actions.up(amount: 1) }}, "+"
      end
    end
  end
end

MyApp.run(id: 'ovto')
```

~ ということで、さっきのやつはOvtoだとこうなります。

Demo

~ これを実際にデモしたいと思います。 ~ - Stateを書く ~ - MainComponentを書く ~ - 変更したい箇所を説明する ~ - Actionを書く ~ - Actionを呼ぶようにする ~ - 実行して、トリガーを押す ~ - 変化する

Vision

- github: yhara/vision

・yhara/vision
・Rails5 + Ovto

~ Ovtoを作りはじめたのが昨年の春で、秋ごろには実際にアプリを書ける状態になりました。そこで作ってみたのがこのTODOアプリです。Visionといいます。

Part 2. Design of Ovto

~ パート２ではOvtoの設計、どんなことを意識して作ったのかを話します。

Ovto's design principle

- Make it fun.
- Programming is fun
- Remove what's not fun → WIN

~ Ovtoの原則は、(私が)楽しくプログラミングできること。 ~ プログラミングはもともと楽しいものなので、楽しくないこと、つらいことを取り除いていけばいいはずと考えました。つらいことってどんなことでしょうか？

Simplicity

- "There's too many things to learn :-("
- Only have 3 classes
    - State, Actions, Component

~ 例えば、覚えることが多すぎるのはつらい。 ~ Ovtoはhyperappのシンプルさを引き継いでいるので、3つのクラスだけ覚えればアプリが作れます。

Debuggability

- "undefined method 'to_s' for nil:NilClass"
- Use Ovto::State instead of Hash
  - If state were a Hash:

```
state['pagee']  #=> nil
```

- Use keyword arguments

~ あるいは、デバッグが大変だとつらい。nilに対してメソッドを呼んでしまうこと、よくありますよね？

~ OvtoがStateクラスを持っているのは、実はこのためなんです。 ~ Hashって、値を取り出すときにtypoしてもエラーが出ず、nilになりますよね。そうするとnilを触ったときにエラーが出る。 ~ 間違った箇所とエラーが出る箇所が違うのは、つらいですよね。 ~ OvtoのStateはクラスなので、「pageeというメソッドはないよ」みたいにその場で教えてくれます。

~ またOvtoではキーワード引数を使っています。これも、間違えたときにわかりやすくするためです。フロントエンドは変更が激しいので、変更漏れとかよくありますよね。

Scalability

- "It's too simple for big project :-("
- State → Nesting
- Actions → Split into modules
- Component → o method supports sub component

~ あるいは、シンプルすぎて大きくなると破綻するようでは困ります。 ~ OvtoではState, Actions, Componentのそれぞれについて増えたときの対処を考えてあります。 ~ Stateについては、関連するものをまとめて新しいStateクラスにする(Taskに関するものをまとめたTaskState、とか)。 ~ Actinosはメソッドがたくさんあるだけなので、ふだんRubyクラスをリファクタリングするようにやればOKです。つまり、moduleに分けてincludeする。 ~ Componentは、sub componentをレンダリングする仕組みがあります。

Help you to design app

- "No idea how to make this feature"
- Design it in this order:
  - State
  - View
  - Action

~ あるいは、どこから作っていいかわからないとつらい。 ~ OvtoではState -> View -> Actionの順に考えると自動的にアプリができていきます。

Part 3. Implementation of Ovto

~ パート３ではOvtoの中身が気になる人のために、実装面についていくつか解説します。

Repository

- github: yhara/ovto
- Example codes:
  - examples/*
  - github: yhara/vision
  - github: yhara/OvtoShow

~ ソースコードはgithubにあります。 ~ examples以下にサンプルコードがあります。 ~ あとサンプルという意味では、visionというTODOアプリと、このプレゼンツールOvtoShowがあります。両方ともRails5です。

Reading Ovto's source

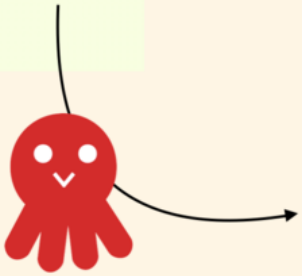
```
% wc -l lib/**/*.rb
      58 lib/ovto.rb
      10 lib/ovto/actions.rb
      85 lib/ovto/app.rb
     198 lib/ovto/component.rb
      53 lib/ovto/fetch.rb
     394 lib/ovto/runtime.rb
      72 lib/ovto/state.rb
       3 lib/ovto/version.rb
      35 lib/ovto/wired_actions.rb
     908 total
```

~ で、Ovtoの本体はlib以下にあります。まだ全部合わせて1000行くらいしかないですね。 ~ state, actions, componentが基本の3クラスです。 ~ で、それ以外で大事なのがruntimeとwired_actionsなので、それぞれ説明します。

runtime.rb(1/2)

Ovto app

```
require 'ovto'

class MyApp < Ovto::App
  class State < Ovto::State
    item :count, default: 0
  end

  class Actions < Ovto::Actions
    def down(state:, amount:)
      return {count: state.count - amount}
    end

    def up(state:, amount:)
      return {count: state.count + amount}
    end
  end

  class MainComponent < Ovto::Component
    def render(state:)
      o "div" do
        o "h1", state.count
        o "button", {onclick: ->{ actions.down(amount: 1) }}, "-"
        o "button", {onclick: ->{ actions.up(amount: 1) }}, "+"
      end
    end
  end
end

MyApp.run(id: 'ovto')
```

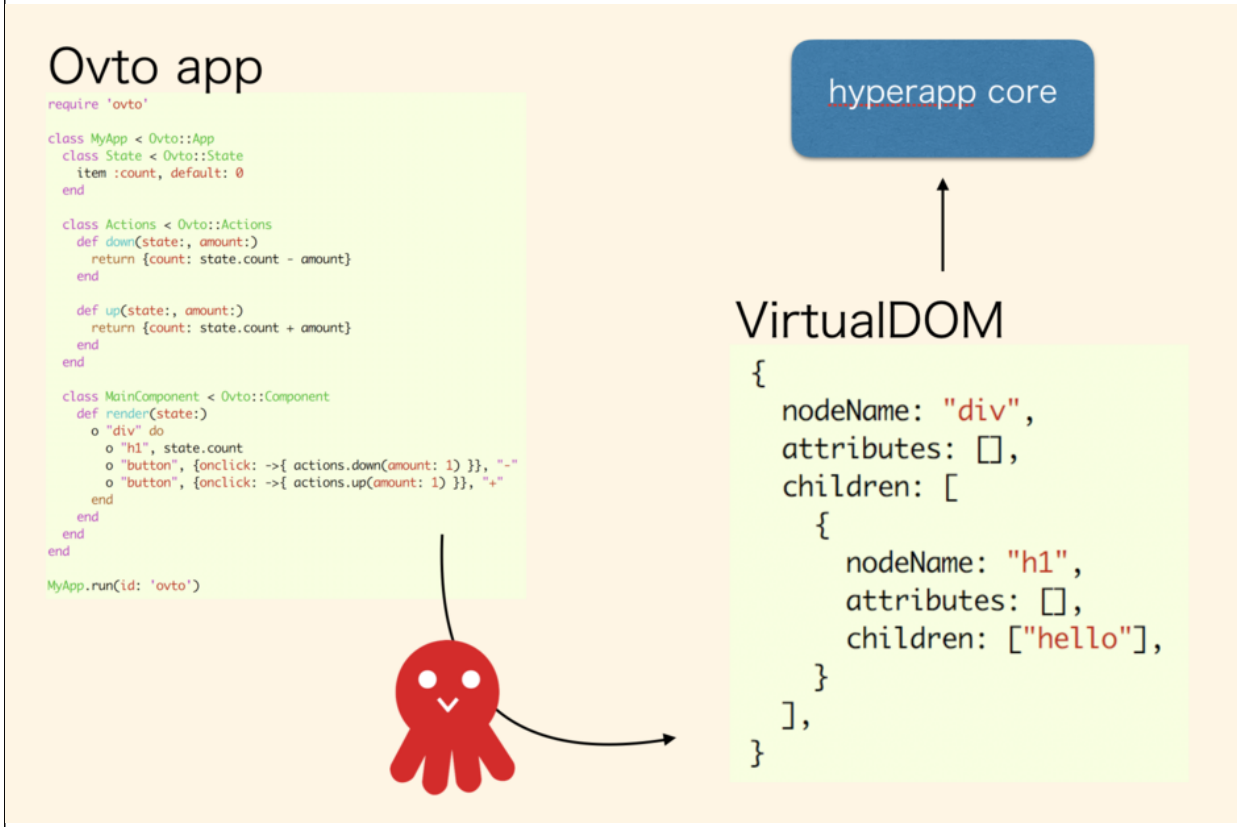


VirtualDOM

```
{
  nodeName: "div",
  attributes: [],
  children: [
    {
      nodeName: "h1",
      attributes: [],
      children: ["hello"],
    }
  ],
}
```

~ まずruntime.rbですが、このファイルにはhyperappのコア部分のコードが入っています。hyperappのコアをそのまま使わせてもらってるんですね。

runtime.rb(2/2)



~ VirtualDOM、つまりDOMの設計図を作るところまでがRubyで、その設計図をhyperappのコアに渡して、描画してもらっているという形です。

wired_actions.rb(1/4)

```
class MyApp < Ovto::App
  class State < Ovto::State
    item :count, default: 0
  end

  module Actions < Ovto::Actions
    def down(state:, amount:)
      return {count: state.count - amount}
    end

    def up(state:, amount:)
      return {count: state.count + amount}
    end
  end

  class MainComponent < Ovto::Component
    def render(state)
      o "div" do
        o "h1", state.count
        o "button", onclick: ->{ actions.down(amount: 1) }, "-"
        o "button", onclick: ->{ actions.up(amount: 1) }, "+"
      end
    end
  end
end
```

~ Ovtoの実装を読む上でもう一つ大事なのが、WiredActionsです。 ~ 例えばさっきのカウンターのサンプルを見てみましょう。 ~ まず、ここにボタンがクリックされたときの処理があって、そこではdownアクションを呼んでいます。 ~ downアクションの実装、downメソッドはここにあります。

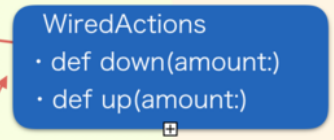
wired_actions.rb(2/4)

```
class MyApp < Ovto::App
  class State < Ovto::State
    item :count, default: 0
  end

  module Actions < Ovto::Actions
    def down(state:, amount:)
      return {count: state.count - amount}
    end

    def up(state:, amount:)
      return {count: state.count + amount}
    end
  end

  class MainComponent < Ovto::Component
    def render(state)
      o "div" do
        o "h1", state.count
        o "button", onclick: ->{ actions.down(amount: 1) }, "-"
        o "button", onclick: ->{ actions.up(amount: 1) }, "+"
      end
    end
  end
end
```

~ そうすると下のdownは上のdownを呼んでいるように見えますが:

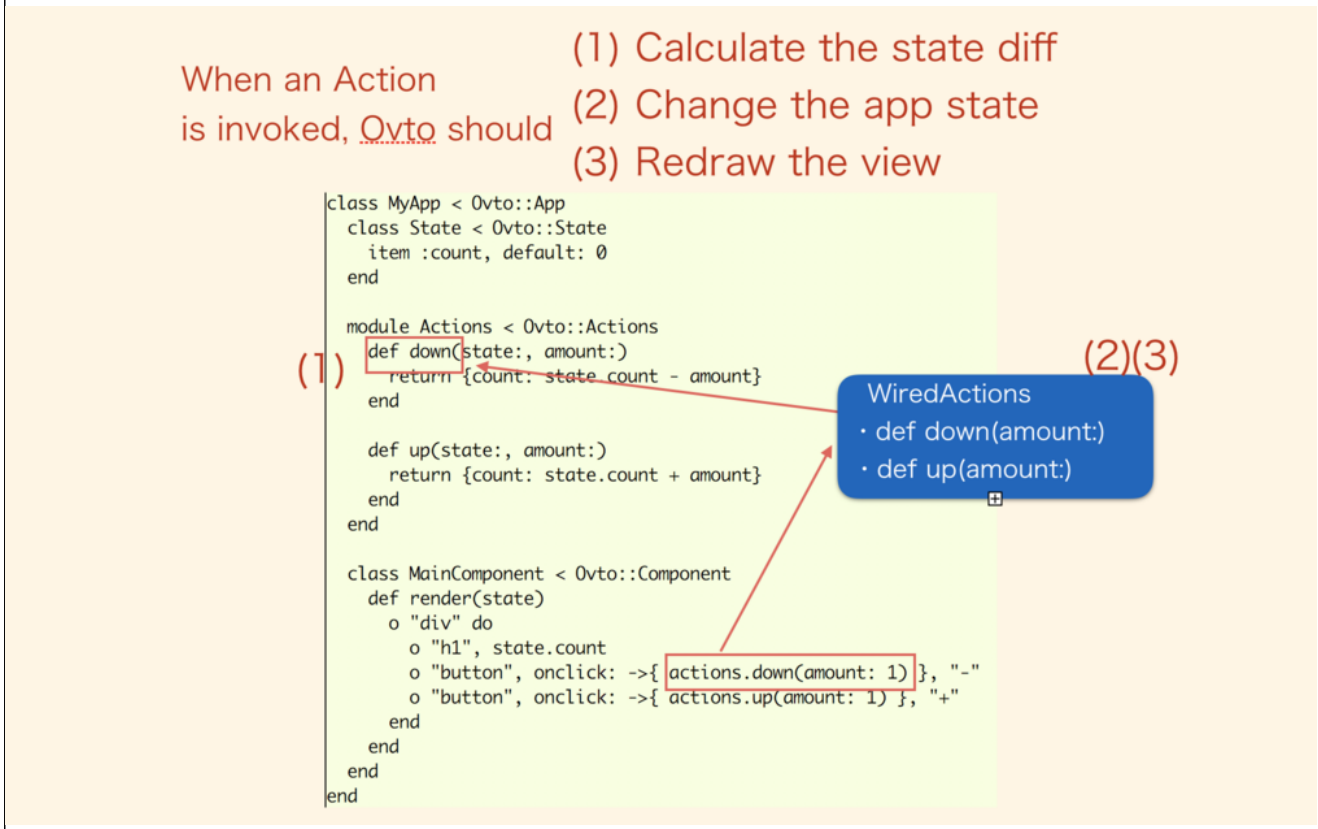
wired_actions.rb(3/4)

```
class MyApp < Ovto::App
  class State < Ovto::State
    item :count, default: 0
  end

  module Actions < Ovto::Actions
    def down(state:, amount:)
      return {count: state.count - amount}
    end

    def up(state:, amount:)
      return {count: state.count + amount}
    end
  end

  class MainComponent < Ovto::Component
    def render(state)
      o "div" do
        o "h1", state.count
        o "button", onclick: ->{ actions.down(amount: 1) }, "-"
        o "button", onclick: ->{ actions.up(amount: 1) }, "+"
      end
    end
  end
end
```



~ 実はそうではなくて、間にWiredActionsという仲介役のようなオブジェクトが存在しています。

wired_actions.rb(4/4)



~ なぜかというと。アクションを実行するときって、以下の3つのことをやる必要があります。 ~ 1. Stateの差分を計算する ~ 2. the application stateを更新する ~ 3. ビューを再描画する ~ 上のdownメソッドがやるのは1の部分だけで、残りの2と3はWiredActionsがやっています。 ~ このことを知っておくと、より理解が深まると思います。

Implementation of OvtoShow(1/2)

- https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

You Might Not Need Redux

Dan Abramov Follow
Sep 20, 2016 · 3 min read

People often choose Redux before they need it. “What if our app doesn’t scale without it?” Later, developers frown at the indirection Redux introduced to their code. “Why do I have to touch three files to get a simple feature working?” Why indeed!

People blame Redux, React, functional programming, immutability, and many other things for their woes, and I understand them. It is natural to compare Redux to an approach that doesn’t require “boilerplate” code to update the state, and to conclude that Redux is just complicated. In a way it is, and by design so.

~ OvtoShow、このプレゼンツールの実装についても少し解説します。

~ redux関係者の人がブログに書いているのですが、彼が言うには、reduxは万能ではなくて、シングルステートが向いたアプリとそうでないアプリがあると。でも、シングルステートだといろいろ興味深いことができるよ、と。 ~ 例えばあなたのアプリにアンドゥ・リドゥ機能を追加したくなったとしましょう。普通のアプリだとアンドゥを実装するためにはいろいろ考えることがありますが、シングルステートの場合、stateをもとの状態に戻すだけでアンドゥが作れる。

Implementation of OvtoShow(2/2)

- Send action via WebSocket

```
// Send Ovto action call to the channel
send_action: function(name, kwargs) {
  return this.perform('send_action', {ovto_action: name, kwargs: kwargs});
},

received: function(data) {
  //console.log("received", data);
  if (data['action'] == 'send_action') {
    // Perform Ovto action sent to the channel
    var action_name = data['ovto_action'];
    var args_hash = Opal.hash(data['kwargs']);
    Opal.OvtoApp.$instance().$actions().$send("invoke_action", action_name, args_hash);
  }
},
```

~ あるいはあるブラウザで行った操作をリアルタイムに別のブラウザに反映させたいとしましょう。これはまさにこのプレゼンツールでやっていることです。こうやって管理画面でキーを押すと、スクリーンにも、あなたの手元にも反映されます。

~ というとなんだか難しそうですが、実は通信部分のコードはほんの少ししかありません。「アクションを送信する」という処理です。~ たとえばゲームやチャットアプリを作る場合も、このコードをコピペすればそれだけでブラウザ間通信に対応できます。~ (もちろん任意のアクションが実行できてはまずいので、サーバ側でホワイトリストしてやるべきですが。OvtoShowはそのようになっています)

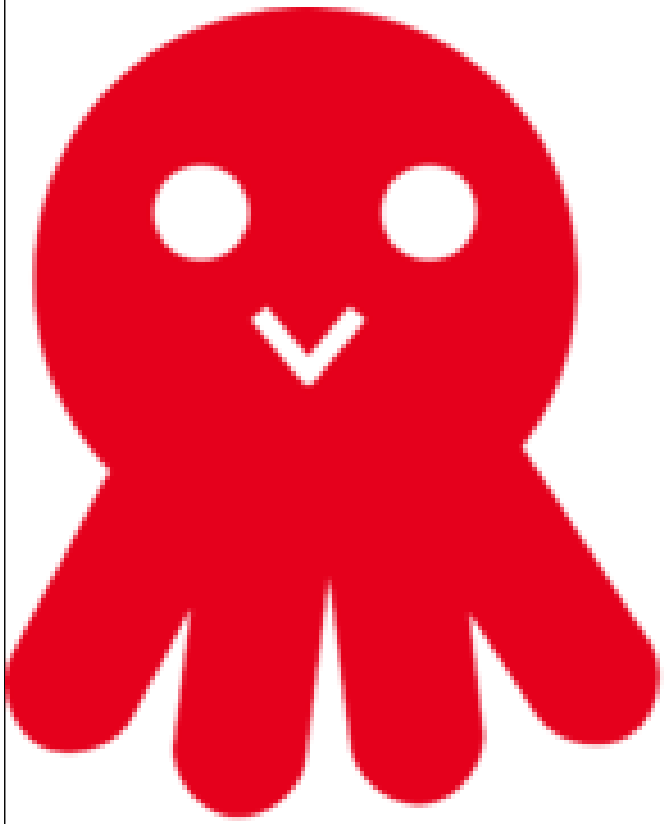
Contributing to Ovto

- Try it!
  - Write posts to your blog, dev.to, Qiita, etc.
- Future plan
  - Server-side rendering
  - Router

~ 最後に、何をしてほしいかです。 ~ まず、実際に触ってみてほしい。そして、ブログとかに感想を書いてほしいです。 ~ 今後としては、サーバーサイドレンダリングとか、ルーティングのサポートとかがありますけど、リクエストがあればやろうと思うんで、リクエストしてください。

~ Ovtoが何らかの形で皆さんの役に立てば幸いです。

Thank you!



~ (Tips: xを押すとスライドが回転します)