明日からできない
React.js, Redux入門

落合隆行

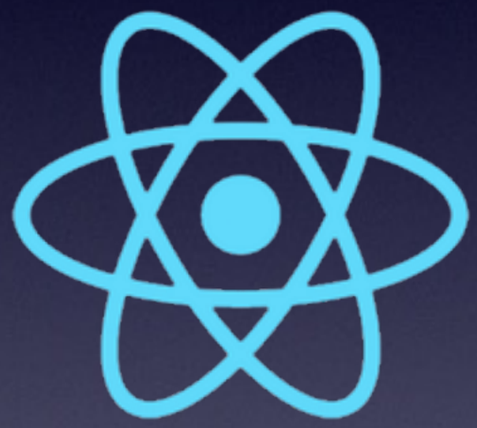
今日伝えること

- React.jsの概要・メリット
- FluxとReduxの概要・メリット
- 今後興味を持って自習したりするときに必要な全体の俯瞰図

伝えないこと

- React.jsやReduxでぱーふぇくとぷりちーなアプリを作る方法

React



4行でわかるReact

・Facebookがメインで開発を進めているOSS

・画面の描画に関わるJavaScriptのViewライブラリ
（フレームワークではない！）

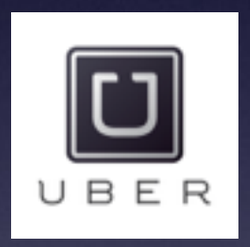
・Google先生が担いでいるAngularと双璧をなす完成度と資料の豊富さで、採用実績も◎

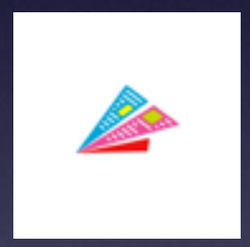
・ライブラリであって、フレームワークではない（大事なことなのでry

採用事例

海外

国内




参考）https://github.com/facebook/react/wiki/Sites-Using-React

最近のフロントエンド

・スマホの普及

・クライアント端末の性能向上

・ネイティブアプリの普及

ユーザーの要求は大絶賛増速中

当然Webアプリの画面側への要求も増える

・もっとリアルタイムに動かしたい

・ネイティブアプリよろしくページ遷移でアニメーションとかつけたい！

・もっとレスポンス早く！

・というかリロードするたびに画面が白くなるのはユーザー体験途切れるからヤダ

これらの要求はAjax + JavaScript(JQuery)でも解決できる

解決できるが、保守が難しくて実装は困難を極める

フロントエンドをリッチにするための課題

・JavaScriptからの従来のDOM操作は難しい

・見た目とデータが分離できていない

・データが変更されたときの画面の変更は自分で書かないと
いけない

・いつどこで誰が持ってるデータが変わって、画面のどこが
変わるのか管理しきれてない

Reactが目指す世界

・データ（JSONで保存）と画面を表示する部品（コンポーネント）を明確に切り離す

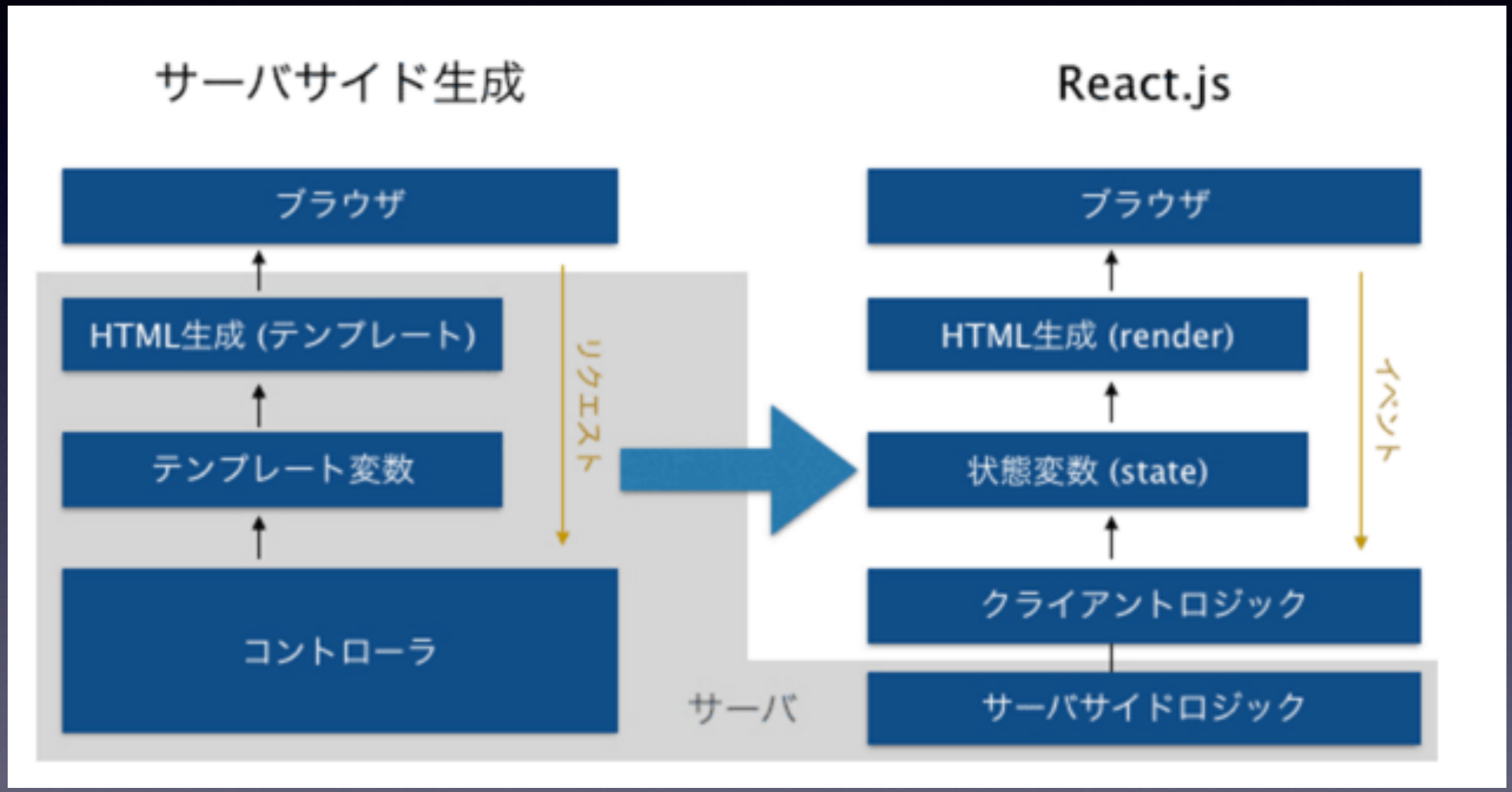
・画面を描画するときは、今のデータの状態からHTMLをクライアント側で作成して表示する

・保存されたデータが書き換えられたときは、まず保存しているデータを書き換えて、書き換え結果をもとにもう一回新しいHTMLを全部作り直す

保存されたデータが書き換えられたときは、まず保存しているデータを書き換えて、書き換え結果をもとにもう一回新しいHTMLを全部作り直す

?

サーバーサイド生成と同じ仕組みをクライアントで

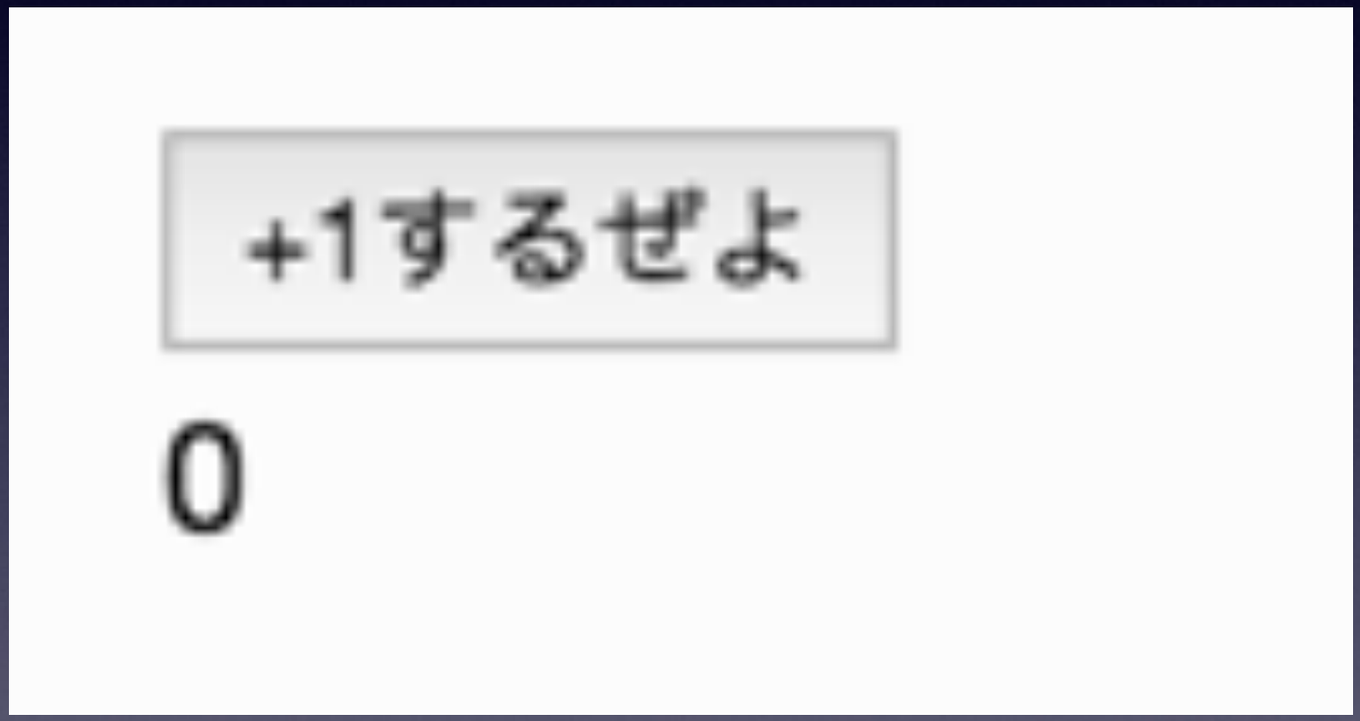


引用) http://blog.masuidrive.jp/2015/03/03/react/

具体的にはどーすんのか？

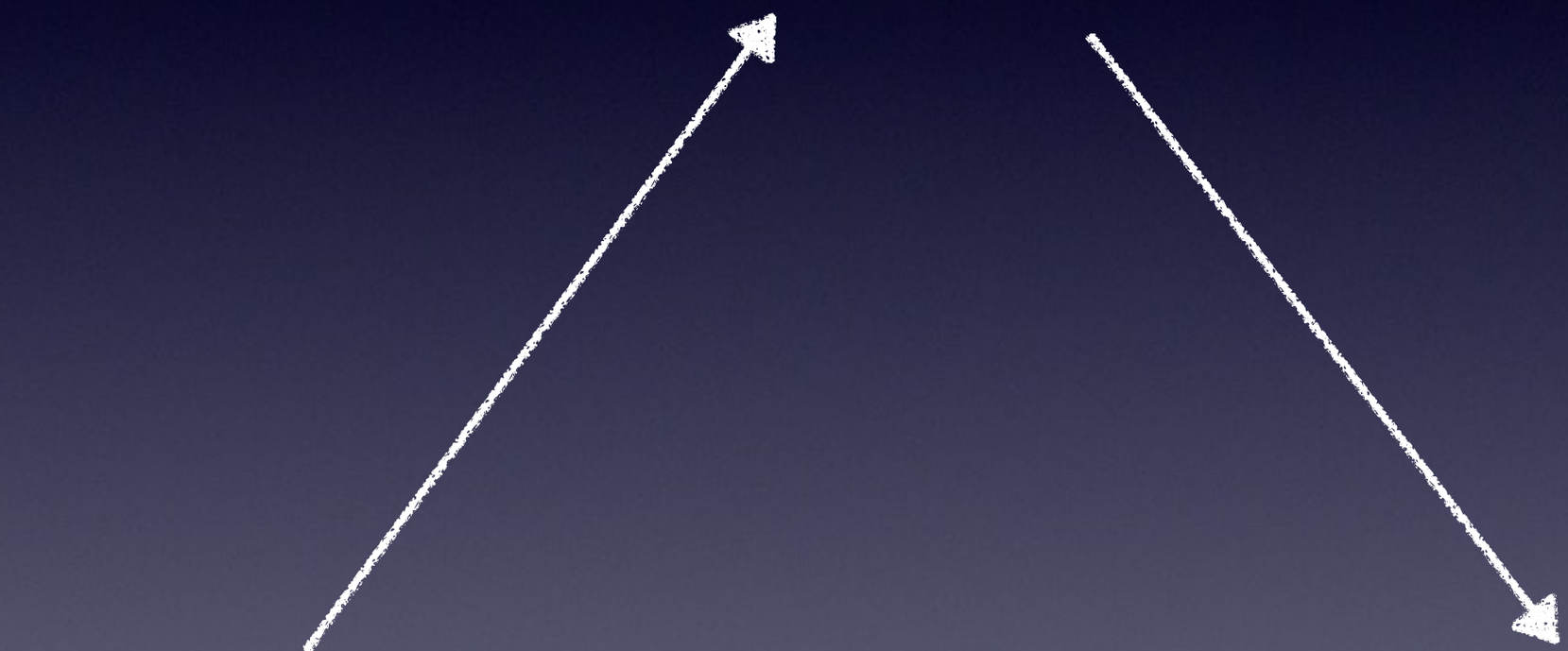
触ってみましょう

カウントアップ！



登場人物

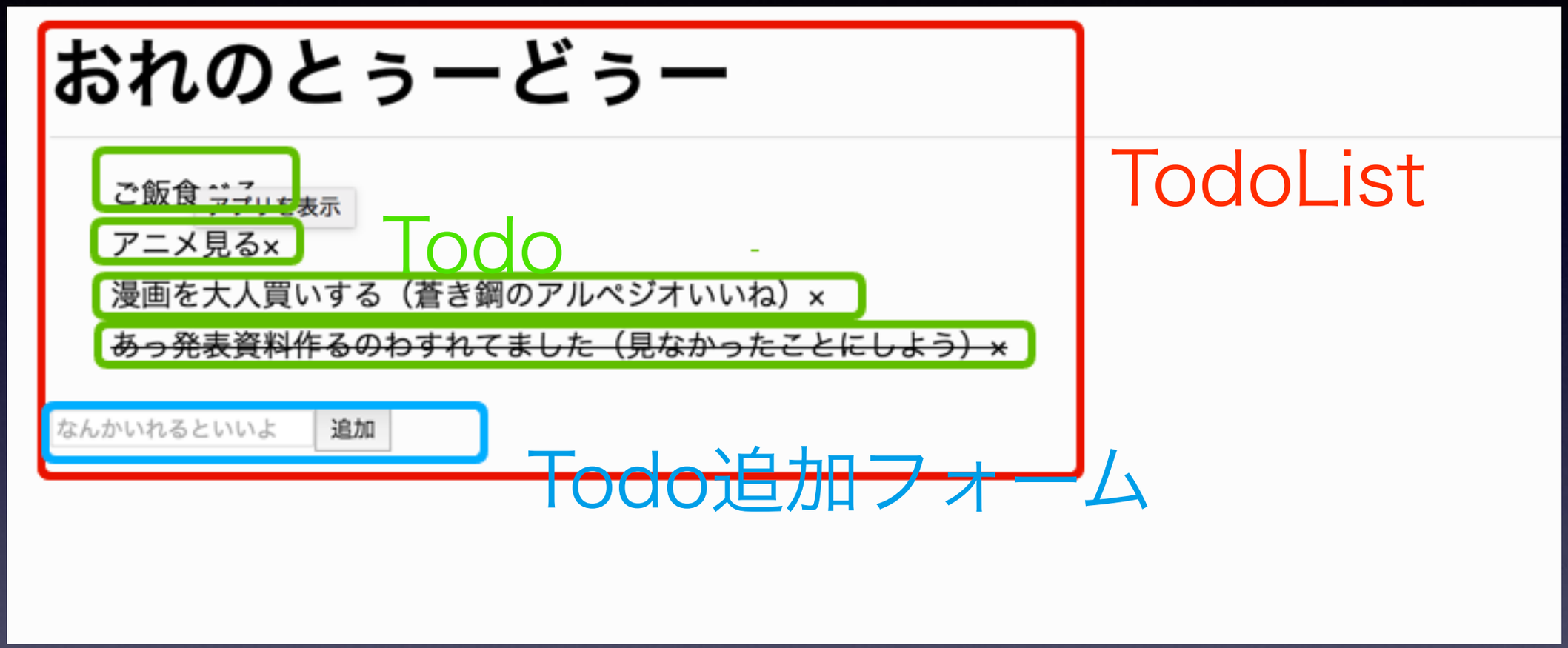
State



setState()

render()

応用のサンプル（おれのとぅーどぅー）



- Todoを表示できる
- 完了したTodoを取り消し線で表現できる
- 新しくTodoを追加できる

HTMLの構成イメージ

TodoList

Todo

TodoForm

データと描画する部品を分離

表示に使っている元データ

```
// コンポーネント内でStateの初期値を定義する関数
getInitialState: function() {
  return {
    todos: [
      { value:'ご飯食べる', done: false },
      { value:'アニメ見る', done: false },
      { value:'漫画を大人買いする（蒼き鋼のアルペジオいいね）', done: false },
      { value:'あっ発表資料作るのわすれてました（見なかったことにしよう）', done: true }
    ]
  }
},
```

Reactではデータはstateかpropのどちらかの状態で管理される

データと描画する部品を分離

描画に使っているテンプレート

Todo

```
// ToDoListの子コンポーネント。親からもらったデータを、変化させることができないpropsという状態でうけとる
var ToDo = React.createClass({
  render: function() {
    var defaultClass = 'callout';

    defaultClass += this.props.done ? ' callout-success' : ' callout-info';
    return (
      <div className={defaultClass}>
        <i className='ficon ficon-checkmark mark-done' onClick={this.props.onClickDone}></i>
        <span>{this.props.value}</span>
        <i className='close' onClick={this.props.onClickClose}>&times;</i>
      </div>
    )
  }
});
```

Reactではこうしたテンプレートをコンポーネントという

複数のコンポーネントを作って別のコンポーネントを作る

ことも可能

HTMLの構成イメージ

TodoList

・stateのデータ（書き換えられる親データ）

・TodoListコンポーネント

Todo

・Todoコンポーネント

・TodoListからもらったTodoを表示するために必要なpropのデータ（このデータは直接書き換えはできない）

TodoForm

・TodoFormコンポーネント

TodoList

```
var todos = this.state.todos.map(function(todo, index) {
  return (
    <ToDo
    key={index}
    value={todo.value}
    done={todo.done}
    onClickClose={this.removeTodo.bind(this, index)}
    onClickDone={this.markTodoDone.bind(this, index)}
  /> );
}.bind(this));
```

```
return (
  <div className='container'>
    <div className='col-xs-6 col-xs-offset-3'>
      <h1>おれのとぅーどぅー</h1>
      {todos}
      <form
        className='form-inline todo-form col-xs-8 col-xs-offset-2'
        role='form'
        onSubmit={this.addTodo}>
        <div className='input-group'>
          <label className='sr-only' htmlFor='todoInput'></label>
          <input type='text' value={this.state.inputValue}
            onChange={this.handleChange}
            className='form-control'
            placeholder='なんかいれるといいよ'
          />
          <span className='input-group-btn'>
            <button className='btn btn-default'>追加</button>
          </span>
        </div>
      </form>
    </div>
  </div>
```

・Todoを４個分作る

・TodoListはTodoに対して自分のstateをpropとして渡す

・作ったTodo4つ分を埋め込み

・TodoForm

・Todo4つ分+TodoForm＝TodoList

データの流れ

TodoList

・stateのデータ（書き換えられる親データ）

・TodoListコンポーネント

②stateが変わったのでもう一度データを渡し直す

①初期表示のためのstateデータ

①doneのbool値の変更

①'入力データ

Todo

・Todoコンポーネント

・TodoListからもらったTodoを表示するために必要なpropのデータ（このデータは直接書き換えはできない）

TodoForm

・TodoFormコンポーネント

ポイント

・データはstateかpropとして保管する

・コンポーネントは自分の持ってるstateかpropを使ってHTMLを表示する

・親コンポーネントは子コンポーネントに自分のstateかpropを渡すことができる

・画面を書き換えたいときは、一回stateを書き換える必要がある。propは直接書き換えることができない！

表で比べるフロントエンド

	手軽さ	大規模・複雑	枯れているか
JQuery	◎	×	○
Angular	○	○	△
React + Flux	△	◎	△

Reactおわり

Flux

Redux

Fluxは設計思想・Reduxはその実装

オブジェクト指向→Java, Ruby, Perl

MVC→Spring, Ruby on Rails, Angular

Flux→Redux

４行でわかるFlux

・データの流れを１方向にするのが目的

・プログラムをView, Action, Dispatcher, Storeの４つの役割で分割する

・この４つをObserverパターンでつなげる*1

- View → Action → Dispatcher → Store → View

*1あるプログラムの部品の状態が変わったとき、状態が変わったことを知らせるためのテクニック。
オブジェクト指向プログラミングのデザインパターンの一種

つまり、FluxとはObeserver
パターンだったんだよ！！！

**Facebookがやったのは、Observerパターンを連鎖させる設計に
Fluxと名付けて枠組みをつくったこと**

参考) http://inside.pixiv.net/entry/2015/04/27/170944

おさらい

フロントエンドをリッチにするための課題

- JavaScriptからの従来のDOM操作は難しい
- 見た目とデータが分離できていない
- データが変更されたときの画面の変更は自分で書かないといけない
- いつどこで誰が持ってるデータが変わって、画面のどこが変わるのか管理しきれてない

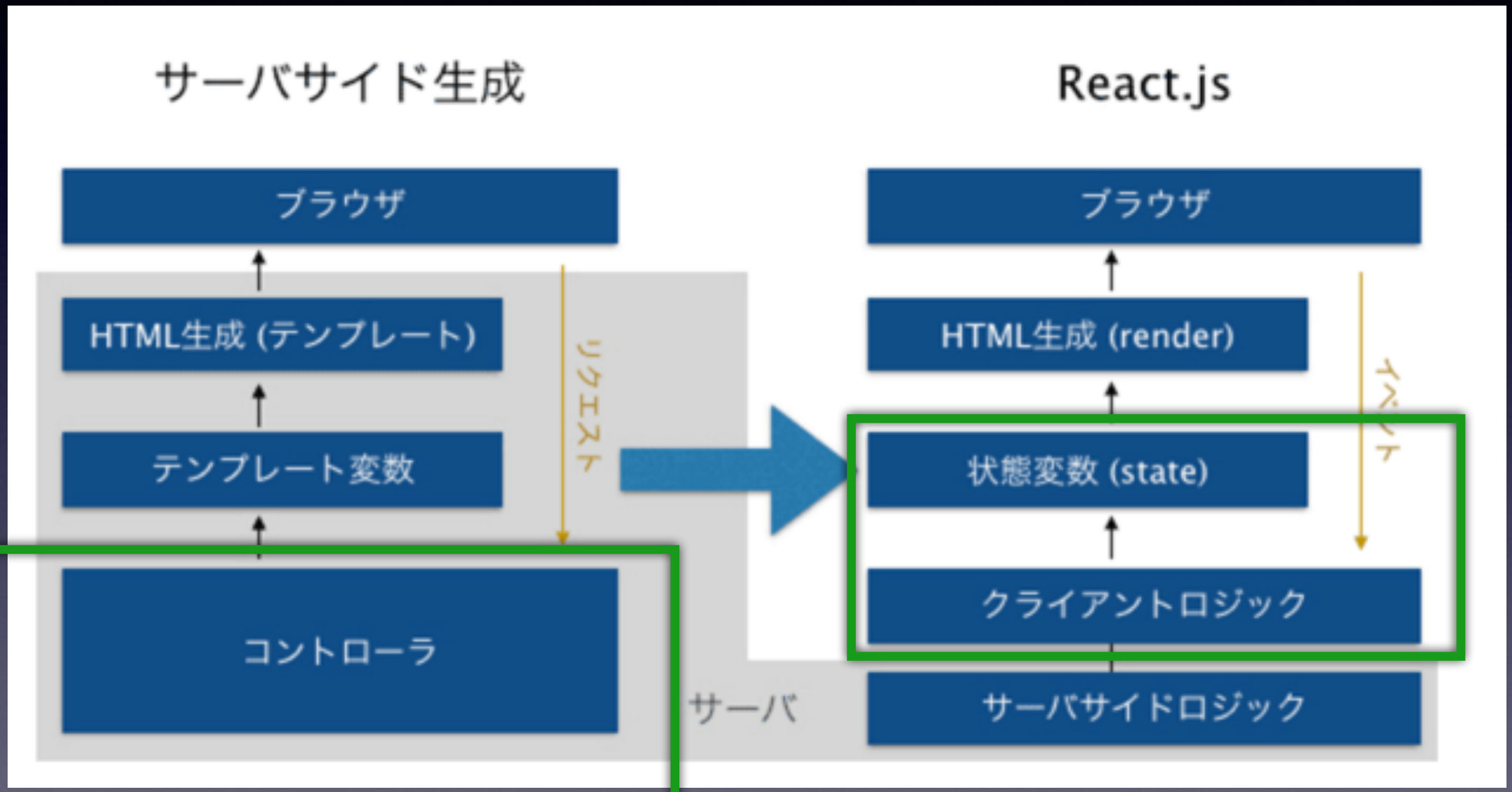
おさらい

フロントエンドをリッチにするための課題

- JavaScriptからの従来のDOM操作は難しい
  →DOM操作の部分はReact.jsがやってくれる
- 見た目とデータが分離できていない
  →データはJSONとして保持・供給される
- データが変更されたときの画面の変更は自分で書かないといけない
  →データが変わるとReactが勝手にrenderメソッドを呼んで再描画してくれる
- いつどこで誰が持ってるデータが変わって、画面のどこが変わるのか管理しきれてない
  →そもそもReactはビューライブラリなので管轄外

サーバーサイド生成と同じ仕組みをクライアントで

状態管理もクライアントで



引用) http://blog.masuidrive.jp/2015/03/03/react/

具体例

今回SPAで作ったTodoアプリが管理しなければならない画面側の状態

- Todoの内容や数、完了状態
- Todoの表示条件（全部、未完了、完了）
- どのページを見ているか（表示しているページとURL)
- Todo入力コンポーネントが開いているか閉じているか
- Todo入力フォームに入っているデータの内容
- どのアニメーションを使って遷移するか

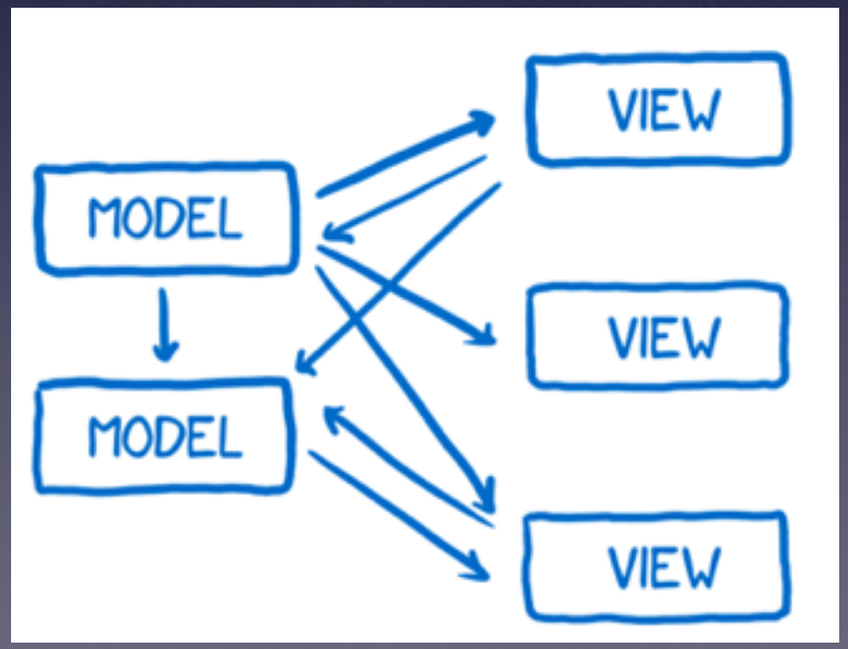
単純なアプリでも、管理しなければならない状態はたくさん。

なぜFluxが必要だったのか？

・Facebookはコードがでかくなりすぎた

・１行直すと3つバグるハウルの動く城ができあがっていた（らしい）

・しかもどのコードがどういう順序で状態や画面を書き換えているのかわからないことが原因



参考) https://medium.com/@sotayamashita/%E6%BC%AB%E7%94%BB%E3%81%A7%E8%AA%AC%E6%98%8E%E3%81%99%E3%82%8B-flux-1a219e50232b#.mlqdh9tkg

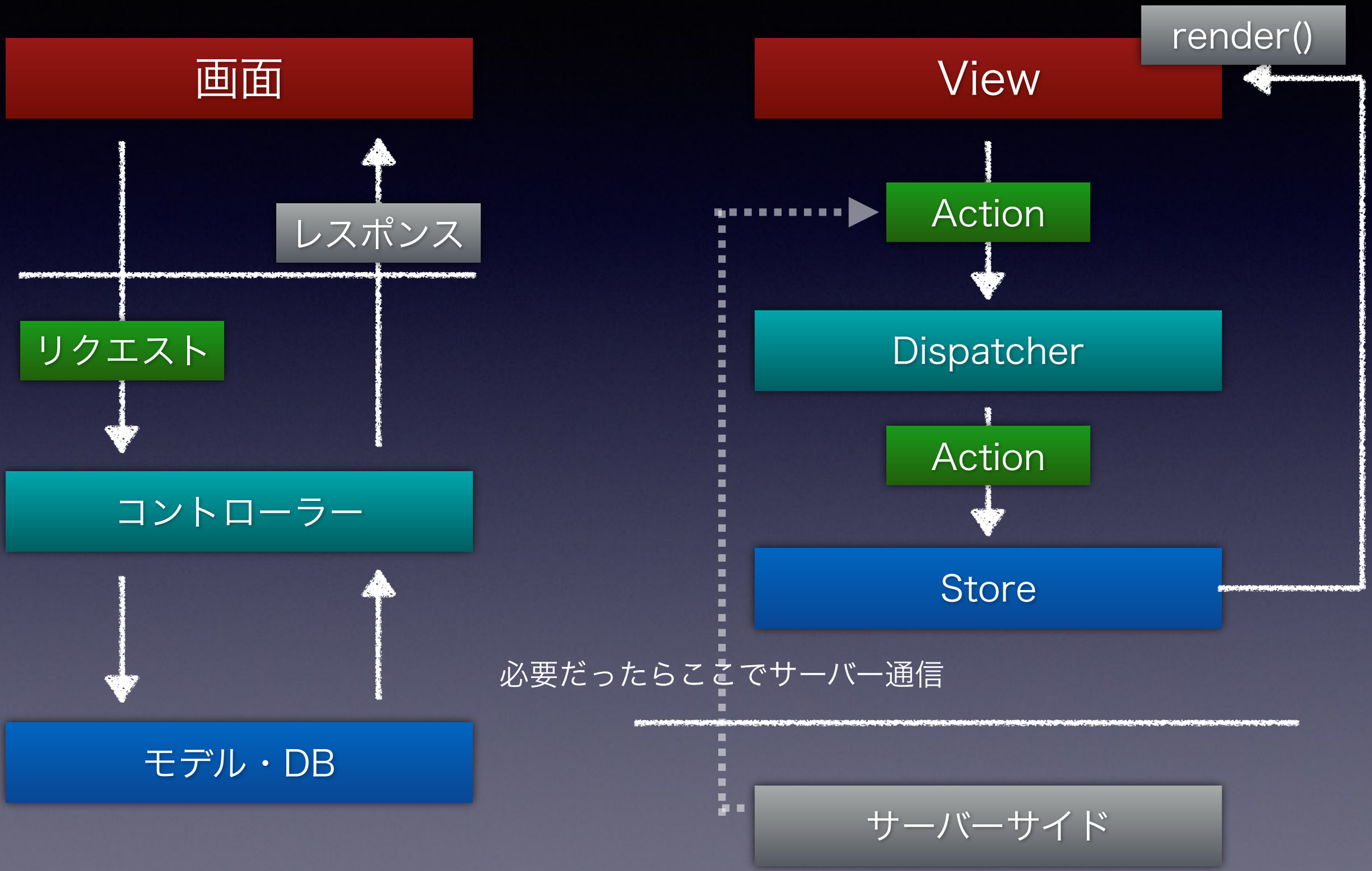
双方向にデータのやり取りが
あるからわかりにくい

データの流れを一方向にしちゃえばどこで何を書き換えたかわかりやすいしトレースしやすいよね！

各部品の役割分担

・View　→　渡されるデータに基づいて画面を描画(React.jsが担当)
　　Storeの変化をlistenしている。
・Action →　アプリケーションの状態の変更や画面の変更時に発火するイベント
　　とどんな変更が起きるかの内容
・Dispatcher →　Actionが発行されたかをStoreに伝える。
　　どのStoreに伝えるべきかはDispatcherが知っている。
・Store → データや状態の保管場所。Dispatherからデータを受け取ると、
　　自分のデータを書き換えて、Viewに「データが変わったから
　　再取得して再描画オナシャス」する

MVCとFluxのざっくり比較



もっと簡単にお願いします

Viewさん：「カート追加したいんでAction発行します」

Dispatcherさん：「カート追加したいってAction来たんで、追加内容を新しくカートデータの保管庫に追加してください」

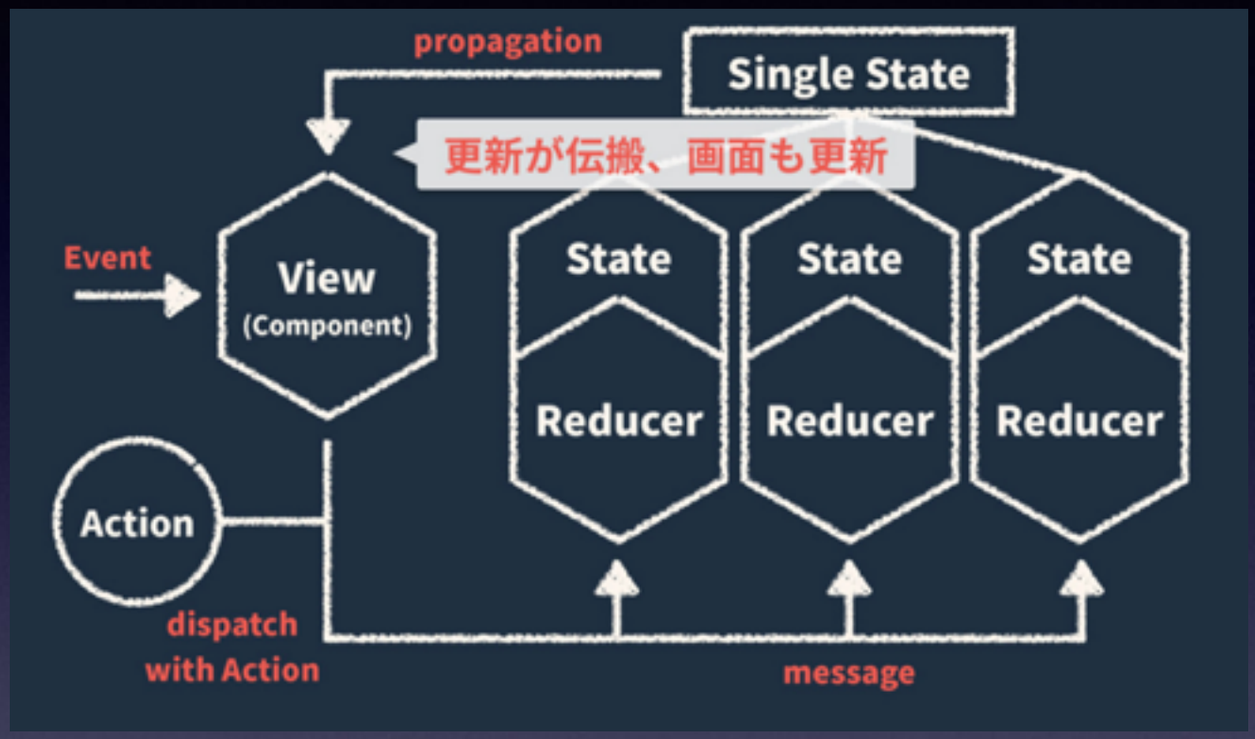
カート担当のStoreさん：「言われた通りカートデータを追加したから、Viewさんはデータを取って再描画してください」

Reduxとは

- Fluxの考え方を実装したフレームワークです
- Flux実装の中では本家Facebookのものを抜いて現在一番人気
- 登場人物はAction, Reducer, Store, View
- Dispatcherはオブジェクトからただのメソッドに格下げ
- Actionは全てReducerが受け取る。受け取った内容を解釈してStoreを更新するのがReducerの仕事

参考）https://developers.eure.jp/tech/redux_feature/

Reduxもデータの流れは一方向



- Viewがイベントに応じてActionを作るよ
- dispatchメソッドでReducerにActionの発行を知らせるよ
- Reducerは受け取ったActionをもとにStoreの状態を変えるよ
- Storeは変更があったらVIewに変更を伝えるよ

引用) https://speakerdeck.com/axross/introduction-to-redux

実際に動かしてみます

実際使ってみて感じたメリット

・どこで入力が起きたら次はどこにデータが流れるか決まっているので、問題の認識や切り分けが楽

・Fluxという枠組みの中で、どこにどんな処理を書くべきかが整理されているため、あんま悩まなくて済む

実際使ってみて感じたデメリット

・書いているとやっぱり冗長になるケースがある（コンポーネントのある部分の表示非表示のトグル処理を書くためにFluxを一周書くのはキツイ）

・2way-bindingのほうが書くコードは少なくて済む

・JSの荒野から必要でメンテされててメジャーなライブラリを自分で選定しないといけない

React + Fluxが向いているもの

- 大規模なSPAの作成
- 画面側のあらゆるところでイベントが発生するアプリ
- リアルタイムにサーバーとやりとりしてその結果を表示するようなアプリ

要するに大きくて複雑なフロントエンド機能を持ったアプリを破綻なく作ることに向いている

質問タイム

Material DesignとかCSS Moduleとかデザイン実装関連の話もどうぞ

最後に

フロントエンドは戦国時代

技術の変化が凄く早いし１年

前に流行った技術が廃れてる

なんてこともある

でも、その歴史は繋がっているし、考え方がただただ捨てられているわけではないし、
現在に影響を与えている

その技術がなぜ出来たのか、何を解決するのか正しく理解した上で、今必要な技術を選択できるといいですね！