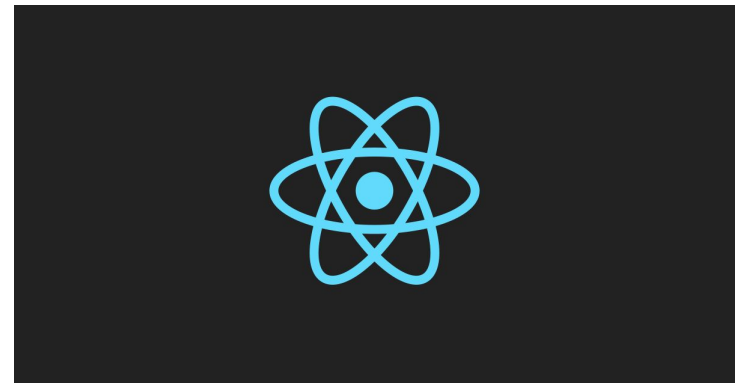


# ReactJS

[bleizard@cs.ifmo.ru](mailto:bleizard@cs.ifmo.ru)  
[ivanuskov@cs.ifmo.ru](mailto:ivanuskov@cs.ifmo.ru)

# React - js библиотека для построения UI

- open-source (<https://github.com/facebook/react>)
- by Facebook
- Текущая версия ~~v15.5.4~~ ~~v15.6.1~~ ~~v16.0.0~~ ~~v16.1.0~~ ~~v16.1.1~~ ~~v16.2.0~~ v16.6.3
- Используют: facebook, instagram, periscope, imdb, twitch, uber, bbc и др.



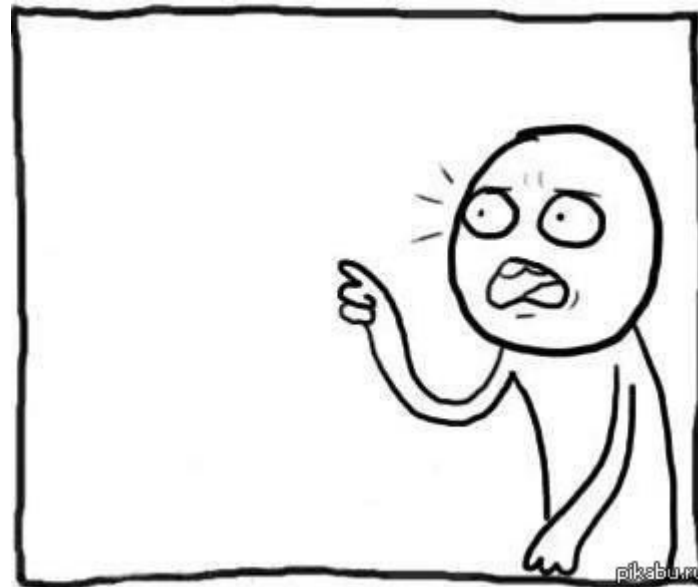
# Зачем?

- Декомпозиция
- Переиспользование
- Желание мыслить в ООП стиле
- Делегирование обязанностей по управлению DOM
- Единый интерфейс взаимодействия
- Единый жизненный цикл компонента

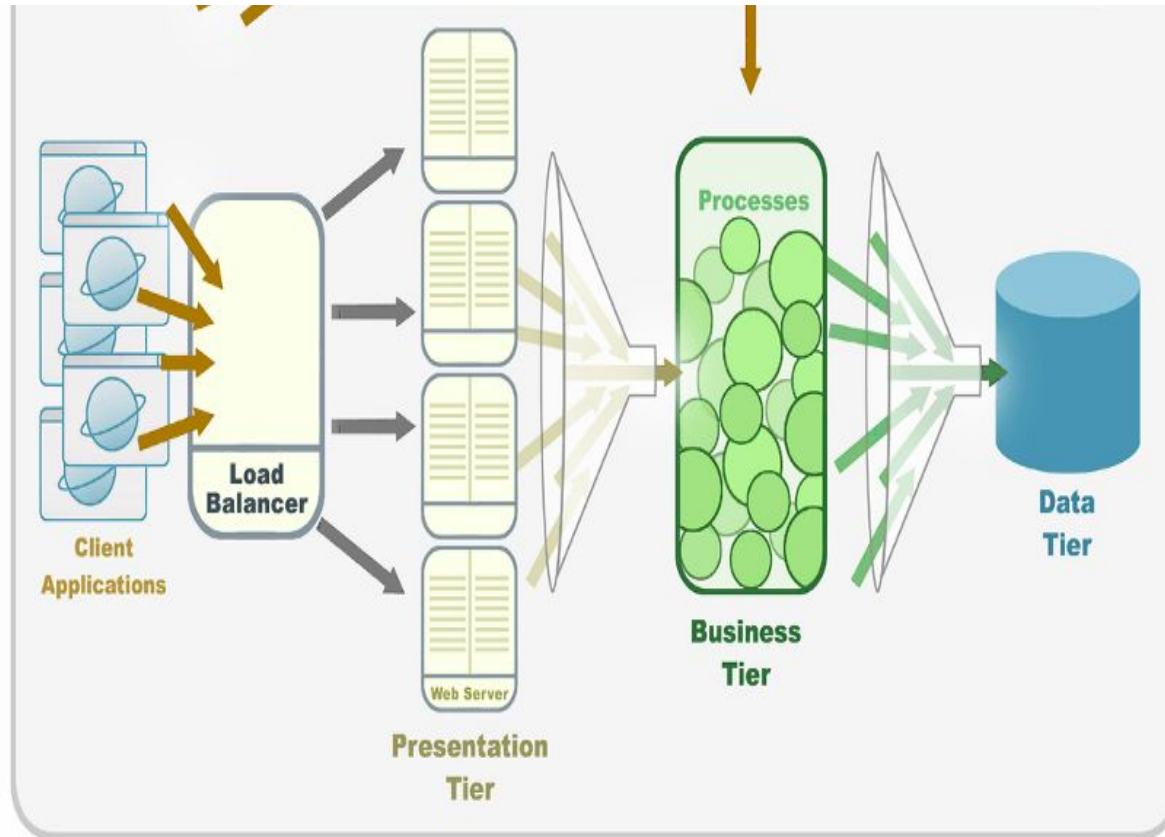
# Но есть же JQuery

JQuery делает проще API для:

- работы с DOM
- использования AJAX
- обработки событий
- анимаций



# Трехуровневая архитектура



# SPA - Single Page Application

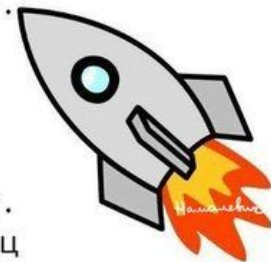
Основные идеи:

- Оффлайн режим
- Перенос части работы с сервера на клиента
- Единый html документ
- Routing через JS (html5 history api)
- Page State



Твой мобильный телефон имеет больше вычислительной мощности, чем всё оборудование NASA в 1969 году.

Они запустили человека на луну.  
Мы запускаем птиц в свиней.



# Browser history API

## Зачем?

Управлять историей браузера через JS

## А что он умеет?

- `history pushState()` и `replaceState()`
- `location forward()` и `back()`
- `location.href`

## Он Откуда?

BOM: `window.location` и `window.history`

<https://habrahabr.ru/post/123106/>

# Web storage API

## Зачем?

Чтобы хранить чего-нибудь на клиенте и не пользоваться cookie

## А что он умеет?

- две хеш-мапы: `localStorage` и `sessionStorage`
- `localStorage` - у каждого домена свой, пока js не почистим не удалится
- `sessionStorage` - выключил браузер, всё потерял

## Он Откуда?

BOM: `window.localStorage` и `window.sessionStorage`

[https://www.w3schools.com/html/html5\\_webstorage.asp](https://www.w3schools.com/html/html5_webstorage.asp)

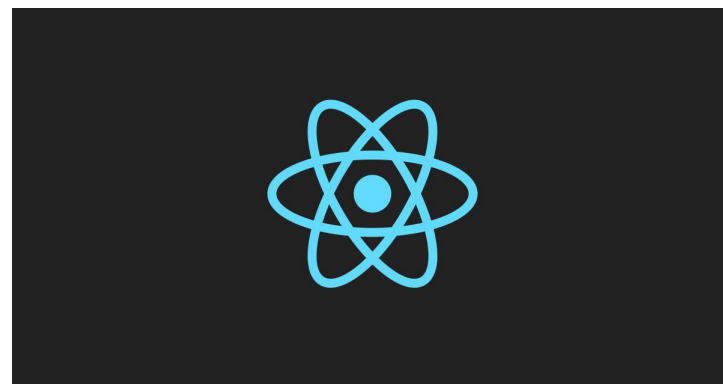


# React - js библиотека для построения UI

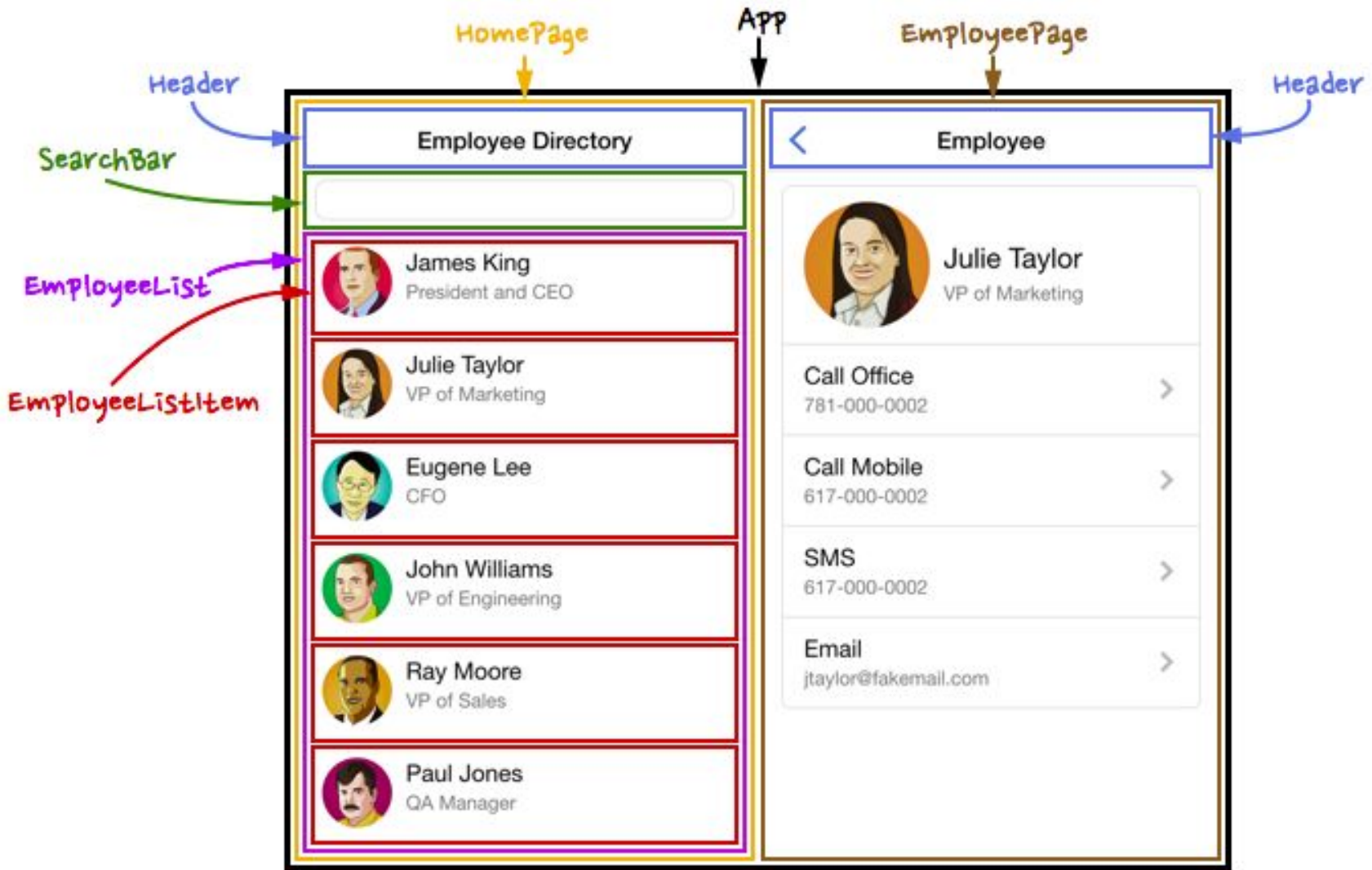
(не только)

Основные особенности:

- основан на компонентах
- декларативный
- one-way dataflow
- Virtual DOM (на самом деле не совсем DOM)

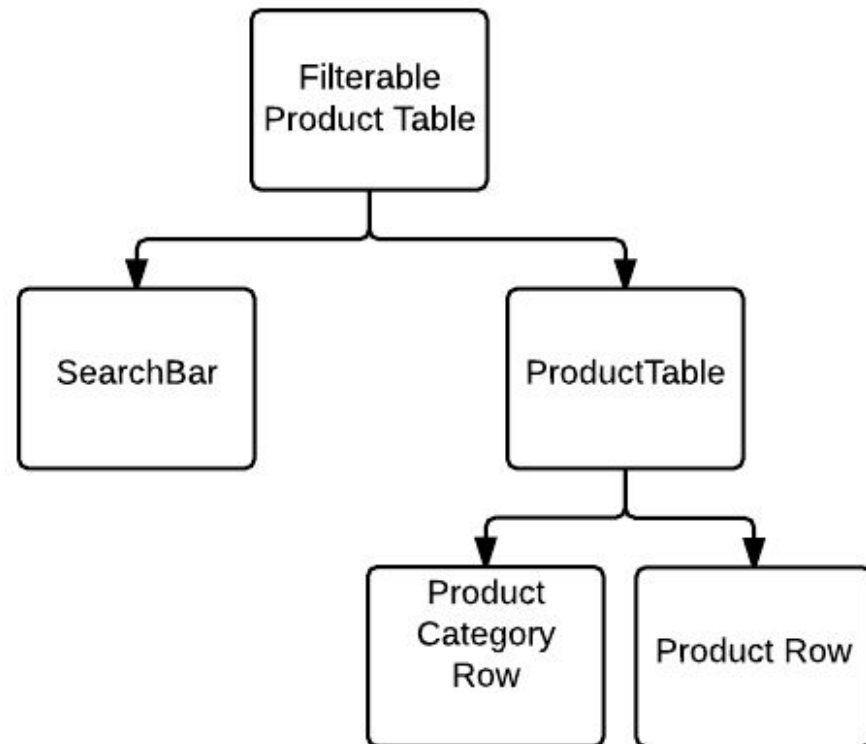


# React (компоненты)



# React (компоненты)

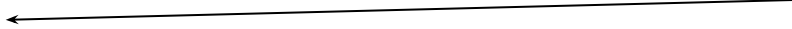
Name	Price
<b>Sporting Goods</b>	
Football	\$49.99
Baseball	\$9.99
<b>Basketball</b>	\$29.99
<b>Electronics</b>	
iPod Touch	\$99.99
<b>iPhone 5</b>	\$399.99
Nexus 7	\$199.99



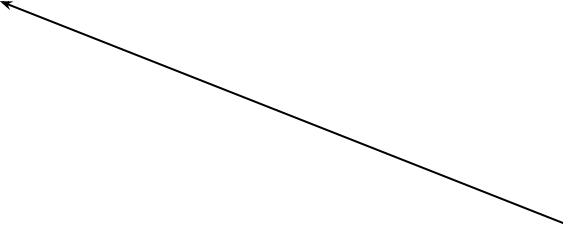
# React (Hello world)

```
class Hello extends React.Component {  
  render() {  
    return React.createElement('div', null, `Hello ${this.props.name}`);  
  }  
}
```

Объявление пользовательского компонента (ES6 class)



Создание div'а и помещение в него некоторого содержимого



```
ReactDOM.render(  
  React.createElement(Hello, {name: 'Вася'}, null),  
  document.getElementById('root')  
);
```

Отображение пользовательского компонента на странице

# JSX

```
const element = <h1>Hello, world!</h1>;
```

JSX:

- Расширение языка JavaScript
- Сахар для `React.createElement(component, props, ...children)`
- Компилируется Babel'ом в JS
- визуально близок к HTML

# React (Hello world) c JSX

```
class Hello extends React.Component {  
  render() {  
    return <div> Hello ${this.props.name}</div>  
  }  
}
```

```
ReactDOM.render(  
  <Hello name="Вася"/>,  
  document.getElementById('root')  
);
```

# JSX — переиспользование КОМПОНЕНТОВ

```
class NameForm extends React.Component {
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name {this.name}:
          <input type="text" value={this.state.value}
            onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

# JSX — переиспользование компонентов

```
class ComponentThatUseAnotherYourComponent extends  
React.Component {  
  render() {  
    return (  
      <div>  
        We use component from previous slide  
        <NameForm/>  
      </div>  
    );  
  }  
}
```



Компонент *NameForm* объявлен на  
прошлом слайде.  
Здесь происходит его подключение.



# JSX

```
render() {
  return (
    <div>
      <h3>TODO</h3>
      <TodoList items={this.state.items} />
      <form onSubmit={this.handleSubmit}>
        <input onChange={this.handleChange} value={this.state.text}
      />
        <button>{'Add #' + (this.state.items.length + 1)}</button>
      </form>
    </div>
  );
}
```

*TodoList* - пользовательский компонент

*div, h3, form, input, button* - встроенные компоненты

Важно: Пользовательские компоненты должны быть написаны с большой буквы

# JSX

```
render() {  
  let name = 'Vasya';  
  return (  
    <div>  
      <h3>Name {this.name}</h3>  
      <TodoList items={this.state.items} />  
    </div>  
  );  
}
```

В фигурных скобках '{}' JS выражения. Т.к. это выражения никаких if, for, объявлений и прочего.

<https://habrahabr.ru/post/319270/>

# Условный рендер в JSX

```
class Greeting extends React.Component {  
  render() {  
  
    const isLoggedIn = props.isLoggedIn;  
    if (isLoggedIn) {  
      return <UserGreeting />;  
    }  
  
    return <GuestGreeting />;  
  }  
}
```

# Условный рендер в JSX

```
render() {  
  const isLoggedIn = this.state.isLoggedIn;  
  
  let button = null;  
  if (isLoggedIn) {  
    button = <LogoutButton onClick={this.handleLogoutClick} />;  
  } else {  
    button = <LoginButton onClick={this.handleLoginClick} />;  
  }  
  return (  
    <div>  
      <Greeting isLoggedIn={isLoggedIn} />  
      {button}  
    </div>  
  );  
}
```

# ЦИКЛЫ В JSX

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to
review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message}
                                message={message} />
      )}
    </ul>
  );
}
```

# JSX

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';
```

```
const components = {
  photo: PhotoStory,
  video: VideoStory
};
```

```
function Story(props) {
  // Wrong! JSX type can't be
an expression.

  return <components[props.storyType]
story={props.story} />;
}
```

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';
```

```
const components = {
  photo: PhotoStory,
  video: VideoStory
};
```

```
function Story(props) {
  // Correct! JSX type can be
a capitalized variable.

  const SpecificStory =
components[props.storyType];
  return <SpecificStory
story={props.story} />;
}
```

# Компоненты: классы и функции

## Классы:

- Наследуются от `React.Component`
- Можно переопределять методы жизненного цикла
- Обладают внутренним состоянием (`state`)
- Принимают `props` от родительских компонентов

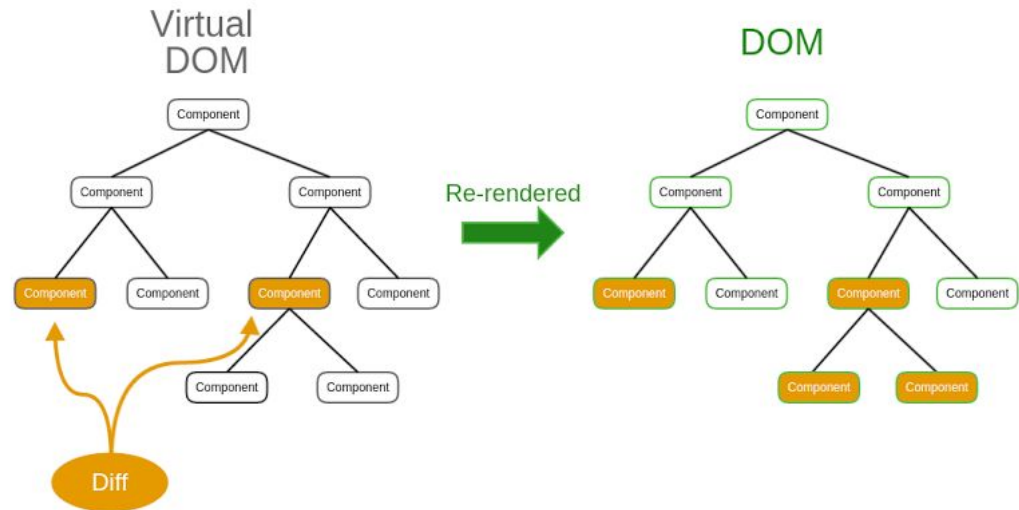
## Функции:

- “Чистые” функции от `props`
- Нет состояния
- Жизненный цикл есть, а привязки к нему нет

# VirtualDOM

Virtual ~~DOM~~ — это техника и набор библиотек / алгоритмов, которые позволяют нам улучшить производительность на клиентской стороне, избегая прямой работы с ~~DOM~~ путем работы с легким JavaScript-объектом, имитирующем DOM-дерево.

Virtual ~~DOM~~ представляет из себя дерево абстрактных компонентов, как их отобразить — задача рендереров (React DOM, React Native, React PDF — тысячи их!).





# state & props

Каждый компонент обладает состоянием 'state' и свойствами 'props'.

Свойства служат для передачи данных между родительским компонентом и дочерним, напоминают атрибуты тегов в XML/HTML

Состояние является внутренним, потому недоступно другим компонентам (но его можно передать через props).

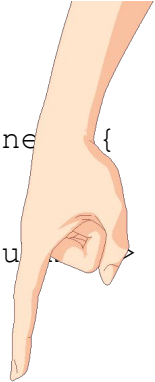
Изменения state/props приводят к перерисовке компонента.

# state & props

```
class MyComp extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {myName: 'Vasya'}  
  
  render() {  
    return (  
      <div>  
        We use component from previous slide  
        <NameForm name={this.state.myName}/>  
      </div>  
    );  
  }  
}
```



```
class NameForm extends React.Component {  
  render() {  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <label>  
          Name {this.props.name}:  
          <input type="text"  
            value={this.state.value}  
            onChange={this.handleChange} />  
        </label>  
        <input type="submit"  
          value="Submit" />  
      </form>  
    );  
  }  
}
```



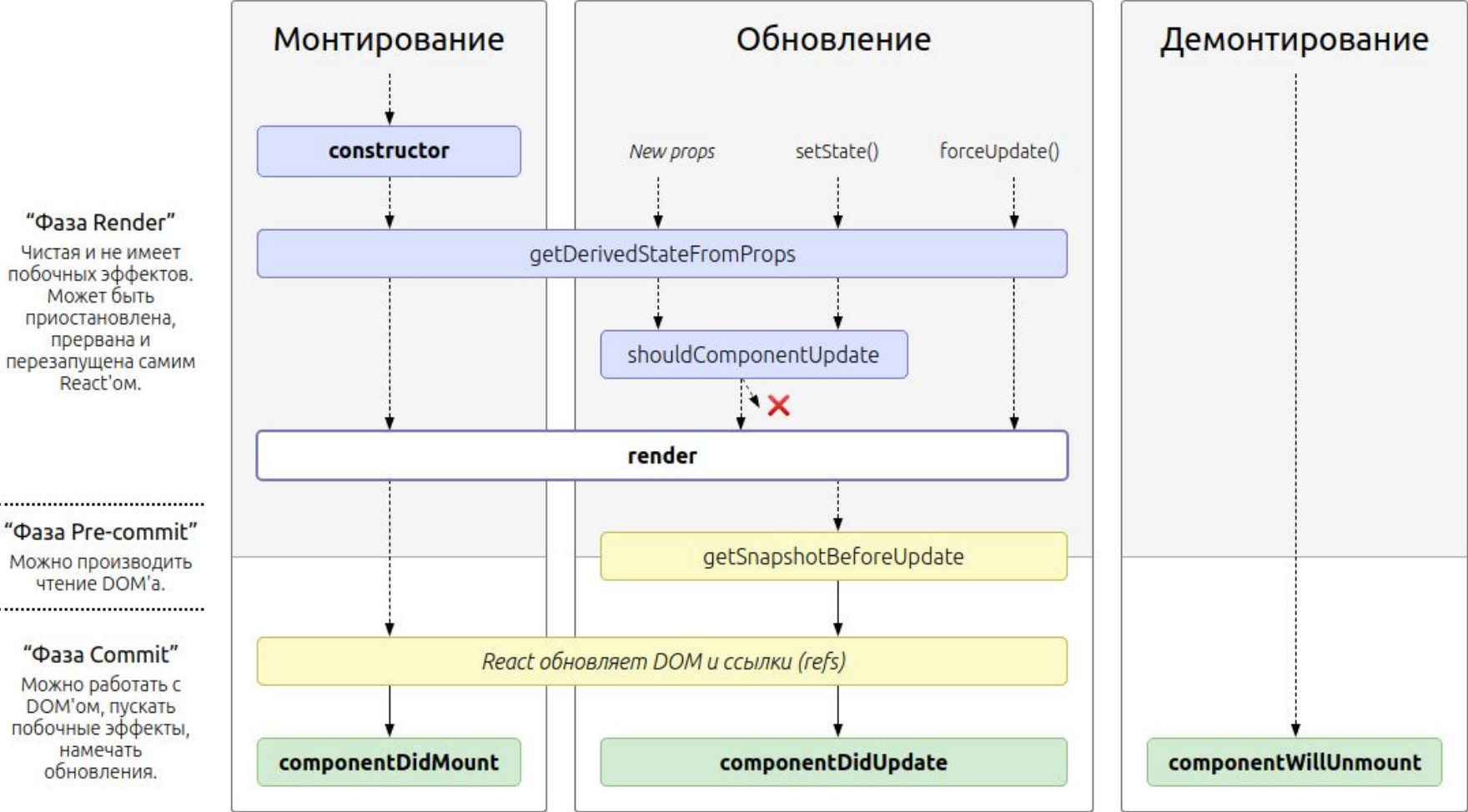
# state

Классы могут обладать своими полями, но это не имеет никакого отношения к state'у (с точки зрения React).

Единственный правильный способ изменения состояния — вызов метода `setState(newState)`.

`setState` не меняет состояние целиком, а объединяет старое и новое.

# Component lifecycle



# Обработка событий

- Обработка событий есть
- В обработчики событий передается не действительное событие DOM'а, а обертка SyntheticEvent
- Обработчики событий именуются в camelCase

```
<button type='submit' onMouseOver={this.onMouseEnter}  
      onClick={this.props.onClick}  
      onMouseDown={(event) => alert('Mouse down!')} />
```

# Обработка событий

```
class Tsopa extends Component {
  constructor(props) {
    super(props);
    this.state = { answer: 'Ответ неверный!' }
  }
  handleStudentAnswer() {
    alert(this.state.answer);
  }
  render() { return (
    <span className='question'>...</span>
    <input name='answer' type='text' />
    <button type='button' onClick={this.handleStudentAnswer}>ОТВЕТИТЬ!
    </button>
  );}
}
```

Ошибка!

Uncaught TypeError: Cannot read property of undefined

# Обработка событий

```
class Tsopa extends Component {
  constructor(props) {
    super(props);
    this.state = { answer: 'Ответ неверный!' };
    //Возможное решение №1:
    this.handleStudentAnswer = this.handleStudentAnswer.bind(this);
  }
  handleStudentAnswer() {
    alert(this.state.answer);
  }
  render() { return (
    <span className='question'>...</span>
    <input name='answer' type='text' />
    //Возможное решение №2:
    <button type='button' onClick={ () => this.handleStudentAnswer() }>
      Ответить!
    </button>
  );}
} В выражении {this.handleStudentAnswer} был потерян контекст выполнения this.
```

# Контролируемые компоненты

```
class NameForm extends React.Component {
  constructor(props) {
    //...
  }
  handleChange(event) {
    event.preventDefault();
    this.setState({value:
event.target.value});
  }
  handleSubmit(event) {
    alert('A name was submitted:
${this.state.value}');
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text"
            value={this.state.value}
            onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Ключевая концепция: реакт полностью управляет отображаемыми данными (в том числе и пользовательским вводом. value у всех input'ов получается из state'a. На изменение значения input'ов навешиваются обработчики событий, которые изменяют state.



# Простейшая форма

```
class Input extends Component {
  inputChange(event) { event.preventDefault(); this.setState({value:
    event.target.value}); }

  render() { return (
    <label htmlFor={this.props.id} className='label'>{this.props.label}</label>
    <input className='input' name={this.props.id} id={this.props.id}
      type={this.props.inputType} value={this.state.value}
      onChange={this.inputChange}/>
    ); }
}
```

**Input хранит во внутреннем состоянии введенное пользователем значение**

```
class Form extends Component {
  submit(submitEvent) { //put your submit here }

  render() { return (
    <form onSubmit={this.submit}>
      <Input id='sample-1' label='Sample input' inputType='text' />
      <Input id='sample-2' label='Yet another input' inputType='number' />
      <button type='submit' onClick={this.submit}>Click me!</button>
    </form>
  ); }
}
```

Форма использует несколько Input'ов и должна что-то сделать с введенными в них значениями (сформировать JSON и отправить на сервер).

Но как получить значения из Input'ов? Они же хранятся во внутреннем состоянии!

# Вверх!

```
const Input = (props) => (  
  <label htmlFor={this.props.id} className='label'>{this.props.label}</label>  
  <input className='input' name={this.props.id} id={this.props.id}  
    type={this.props.inputType} value={this.state.value}  
    onChange={this.props.handleChange}/>  
);  
  
class Form extends Component {  
  submit(submitEvent) { //put your submit here }  
  handleChange(event) {  
    this.setState({ [event.target.name] :  
      event.target.value})  
  }  
  render() { return (  
    <form onSubmit={this.submit}>  
      <Input id='sample-1' label='Sample input' inputType='text'  
        handleChange={this.handleChange} />  
      <Input id='sample-2' label='Yet another input' inputType='number'  
        handleChange={this.handleChange} />  
      <button type='submit' onClick={this.submit}>Click me!</button>  
    </form>);  
  }  
}
```

Состояние выносится как можно выше в иерархии компонентов. Компоненты для пользовательского ввода через props получают callback-функции, через которые они могут изменить состояние родительского компонента.

# Навигация в SPA. React Router.

*React Router* - библиотека для декларативной маршрутизации в React приложениях.

Добавляем в приложение:

```
npm install --save react-router-dom
```

Документация и tutorиалы:

<https://github.com/ReactTraining/react-router>

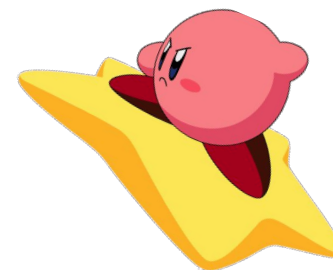
<https://maxfarseer.gitbooks.io/react-router-course-ru/>

<https://habr.com/post/329996/>

# Router & Routes

```
import { BrowserRouter, Route, IndexRoute, } from
'react-router-dom'

render (
  <BrowserRouter>
    <Route path="/" component={App}>
      <IndexRoute component={Home} />
      <Route path='admin' component={Admin}>
        <Route path='banlist' component={BanList} />
      </Route>
      <Route path='lab' component={Lab} />
    </Route>
  </BrowserRouter>,
  document.getElementById('root')
)
```



Вложенные пути

# Dynamic routes

```
<Route path='/users' component={UserList}>  
  <Route path='/users/:userId' component={UserInfo} />  
</Route>
```

Достаём внутри компонента UserInfo

```
const { userId } = this.props.match.params
```

# <Link> вместо <a>

```
import { Link } from 'react-router'

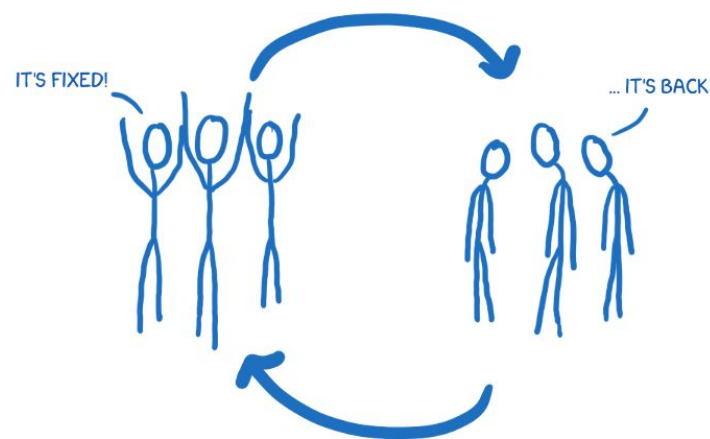
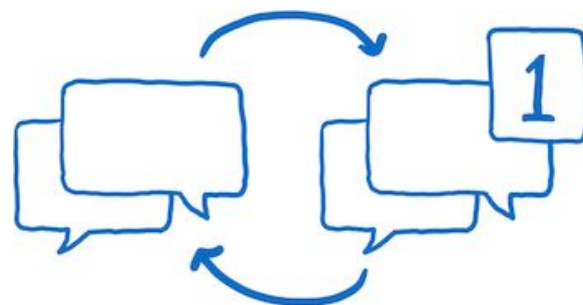
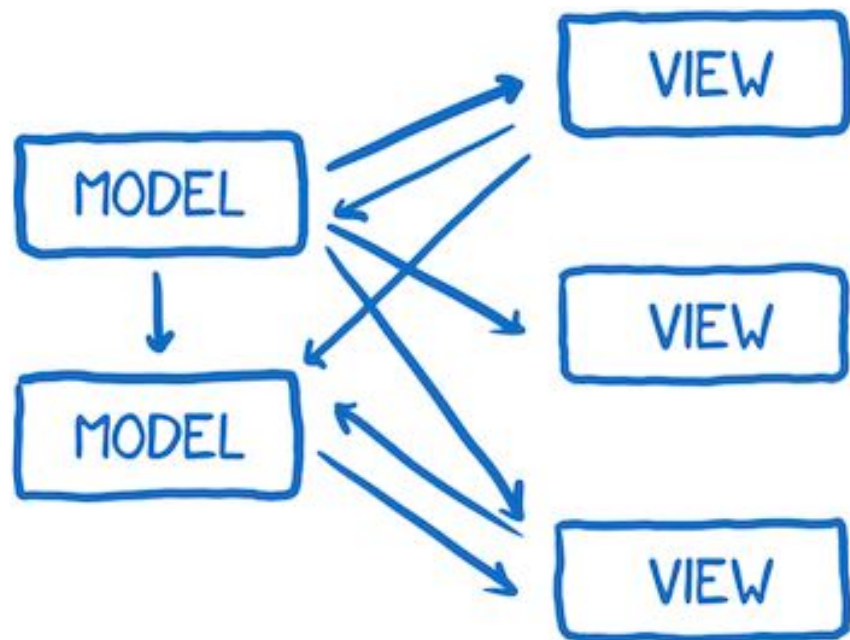
class App extends Component {
  render() {
    return (
      <div className='container'>
        <h1>App</h1>
        <ul>
          <li><Link to='/admin'>Admin</Link></li>
          <li><Link to='/lab'>Lab 4</Link></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
```

P.S. Если делаете навигацию, то используйте <NavLink>.

# Переход без Link (программный)

```
class UserList extends Component {
  handleSubmit(e) {
    e.preventDefault()
    const value = e.target.elements[0].value.toLowerCase()
    this.props.history.push(`/user/${value}`)
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input type='text' placeholder='user id' />
        <button type='submit'>Перейти</button>
      </form>
    )
  }
}
```

# Redux — введение

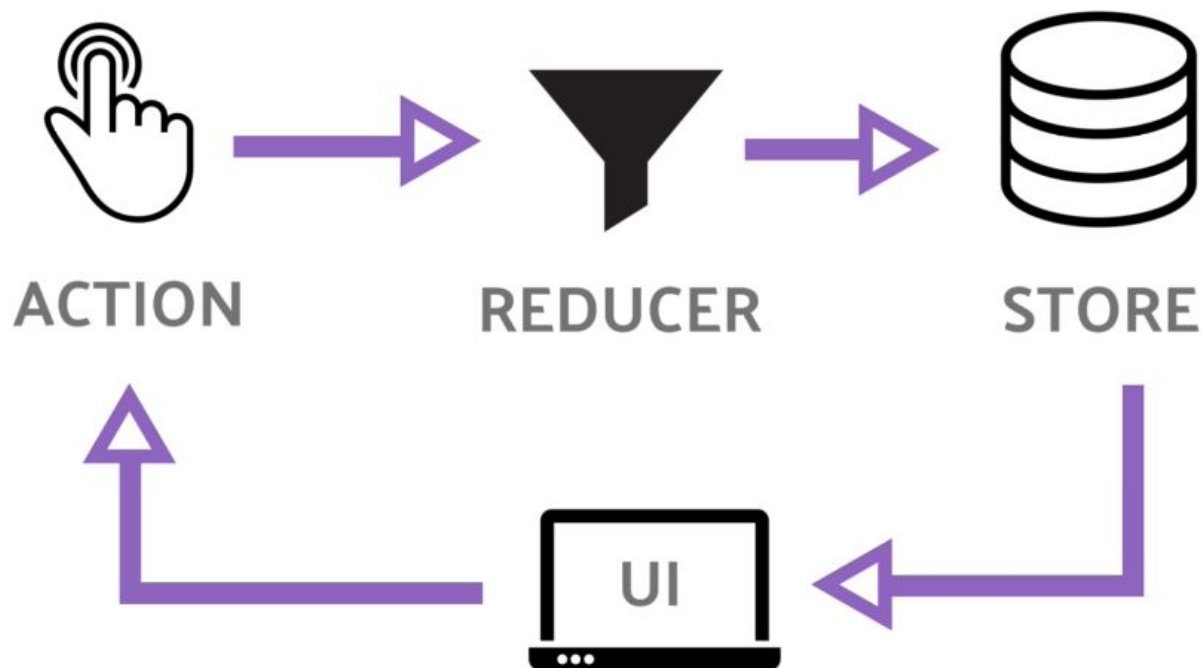


Проблема: многие представления меняют многие модели. Многие модели отражаются во многих других представлениях.

Как синхронизировать состояние?



# Redux



Redux — менеджер состояния приложения, единственный “источник истины” о данных.

# Redux

Заявленные преимущества:

- Возможность сохранения истории всех изменений, а как следствие...
- простота отладки приложения
- Единственный "источник правды" — все компоненты отображают одно и то же состояние
- Компоненты не имеют состояния — их легче тестировать

Недостатки (незаявленные):

- Простейшие вещи требуют написания большого количества кода
- To be continued...

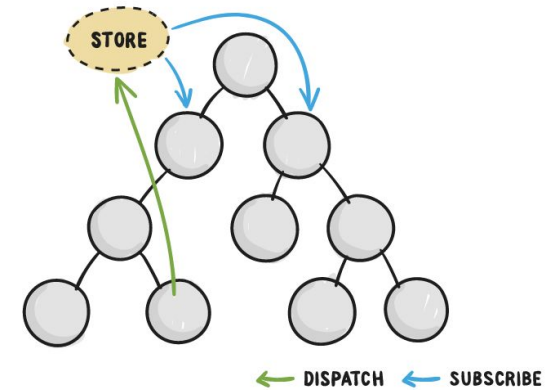
# Redux

API из 9 функций:

- createStore
- combineReducers
- bindActionCreators
- applyMiddleware
- compose
  
- Store
  - getState()
  - dispatch(action)
  - subscribe(listener)
  - replaceReducer(nextReducer)

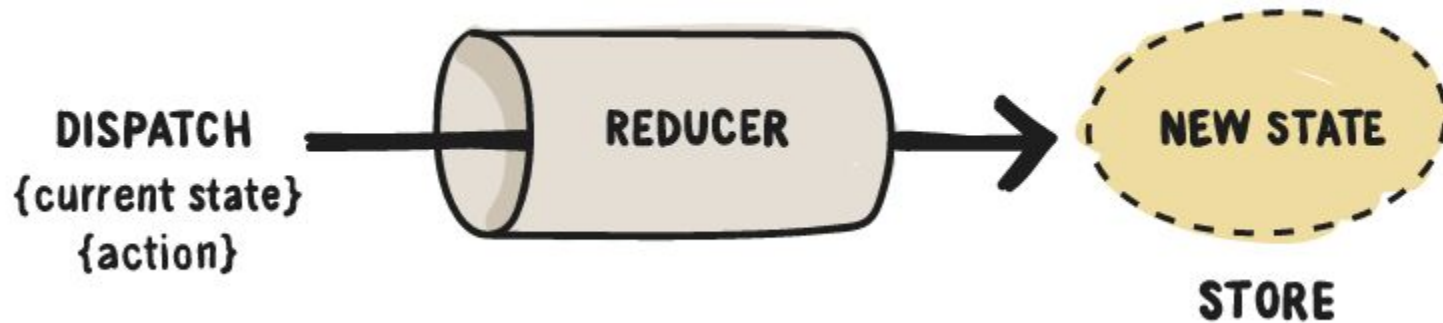
# Store

Redux предоставляет единое хранилище состояния. Ключевой его особенностью является неизменяемость (привет, функциональщина!) — все изменения в него вносятся посредством создания нового объекта состояния.



В “правильном” React-приложении с использованием Redux у компонентов нет state’a

# Reducer



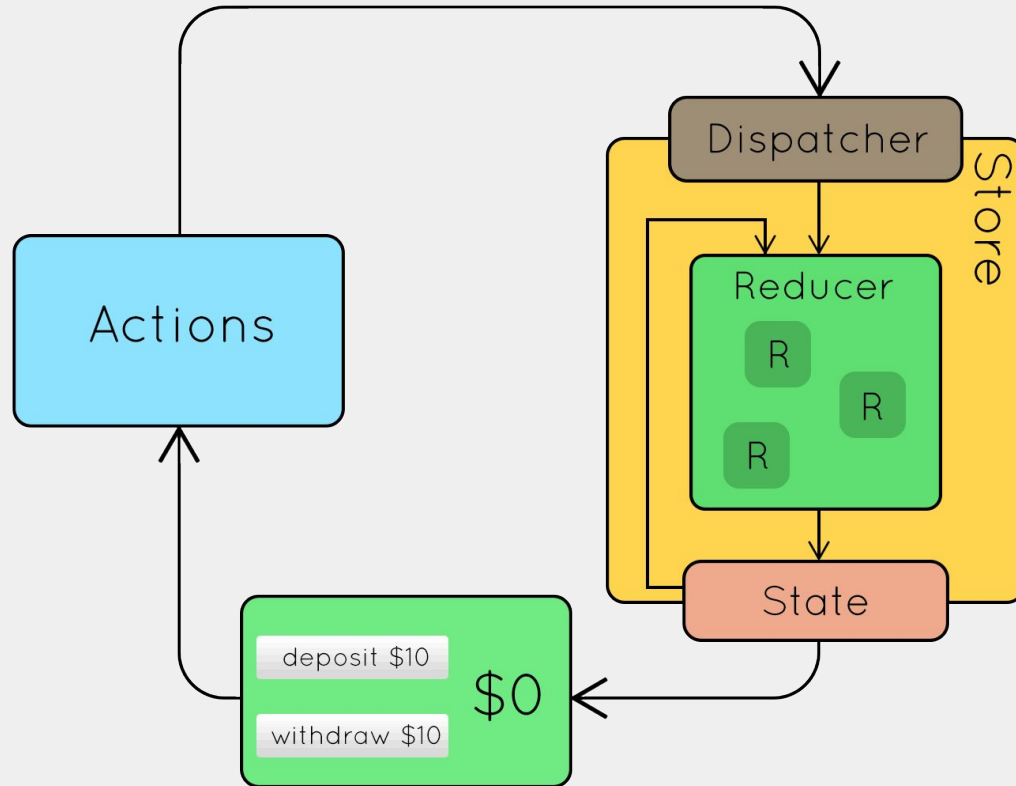
- Чистая функция, её выходное значение зависит только от входных
- Необходимо возвращать новое состояние, состояние должно быть неизменяемо

# Action

Некоторое событие, которое должно привести к изменению состояния (созданию нового объекта состояния), будь то ответ от сервера или нажатие на кнопку в интерфейсе.

С точки зрения библиотеки — любой JS объект, у которого есть идентификатор типа события (обычно строковое свойство `type`) и некоторая полезная нагрузка.

# Redux



# createStore()

function createStore(reducer, preloadedState, enhancer)

- reducer - на основе action и state генерирует новый state
- preloadedState - начальное состояние приложения
- enhancer - цепочка middleware



# Reducer

```
export default function user(state = initialState, action) {
  switch (action.type) {
    case USER_INFO_REQUEST: {
      return Object.assign({}, state, {fetchingInfo: true});
    }
    case USER_INFO_COMPLETE: {
      return fetchingUserInfoComplete(state, action);
    }
    case USER_INFO_FAIL: {
      return Object.assign({}, state, {
        fetchingInfo: false,
        fetchInfoError: action.error,
      });
    }
  }
}
```

# combineReducers()

```
export default function combineReducers(reducers);
```

Пример комбинирования Reducer'а

```
import {combineReducers} from 'redux';
import page from './objects';
import {routerReducer} from 'react-router-redux';
import user from './user';
const rootReducer = combineReducers({
  page,
  user,
  routing: routerReducer,
});

export default rootReducer;
```

# Action

```
export const addTodo = (name, date, comment) => ({
  type: types.ADD_TODO,
  payload: {
    name,
    date,
    comment,
  },
});
```

# dispatch

```
import {addTodo} from '../actions/todo';  
import dispatch from 'redux';  
  
func() {  
    dispatch(addTodo);  
}
```

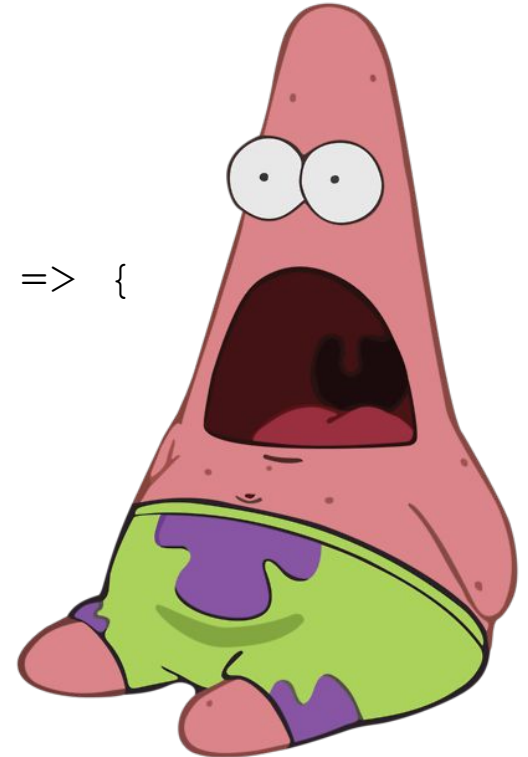
# Middleware

**Каррирование** или **карринг** (англ. *currying*) в информатике — преобразование функции от многих аргументов в набор функций, каждая из которых с одним аргументом. Возможность такого преобразования впервые отмечена в трудах Готтлоба Фреге, систематически изучена Моисеем Шейнфинкелем в 1920-е годы, а наименование получило по имени Хаскелла Карри — разработчика комбинаторной логики, в которой сведение к функциям одного аргумента носит основополагающий характер.

# Middleware

```
function applyMiddleware(...middlewares)  
  
  middleware = (store) => (next) => (action) => {  
  
  }  
  
}
```

Можно воспринимать как функцию от трех аргументов (почти)



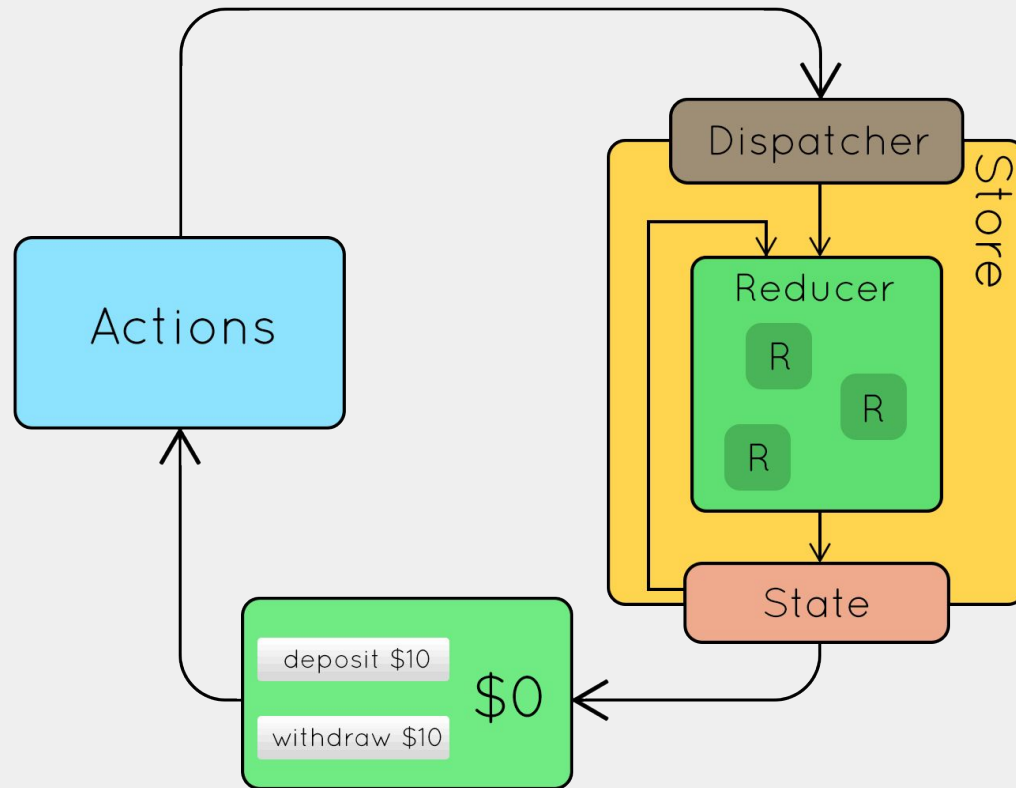
# Middleware

А зачем?

reducer - чистая функция

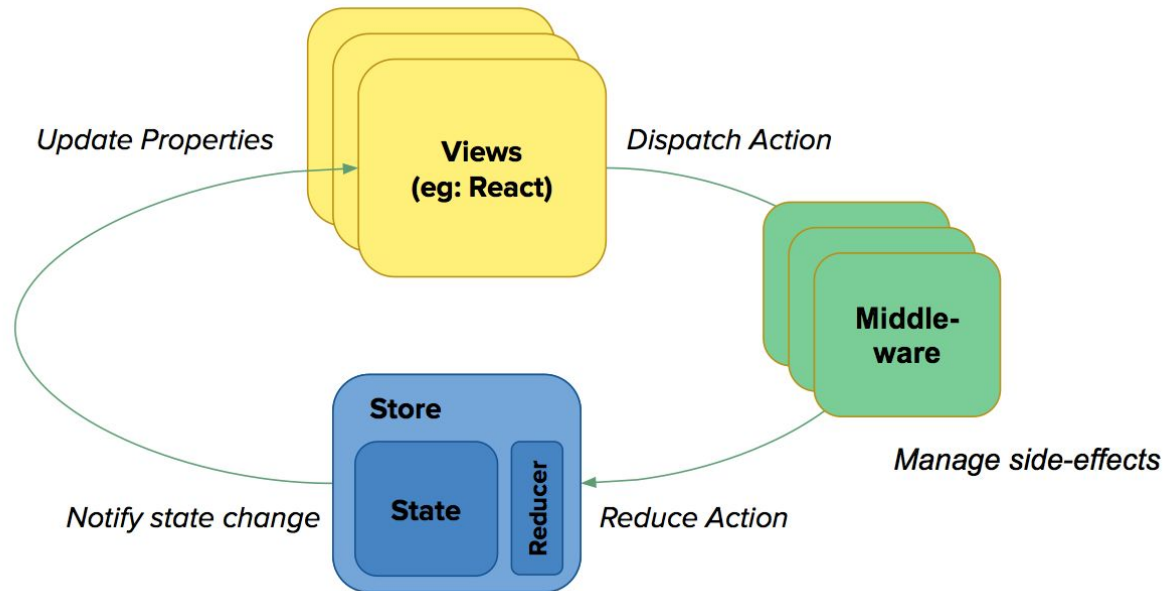
action - объект

# Redux

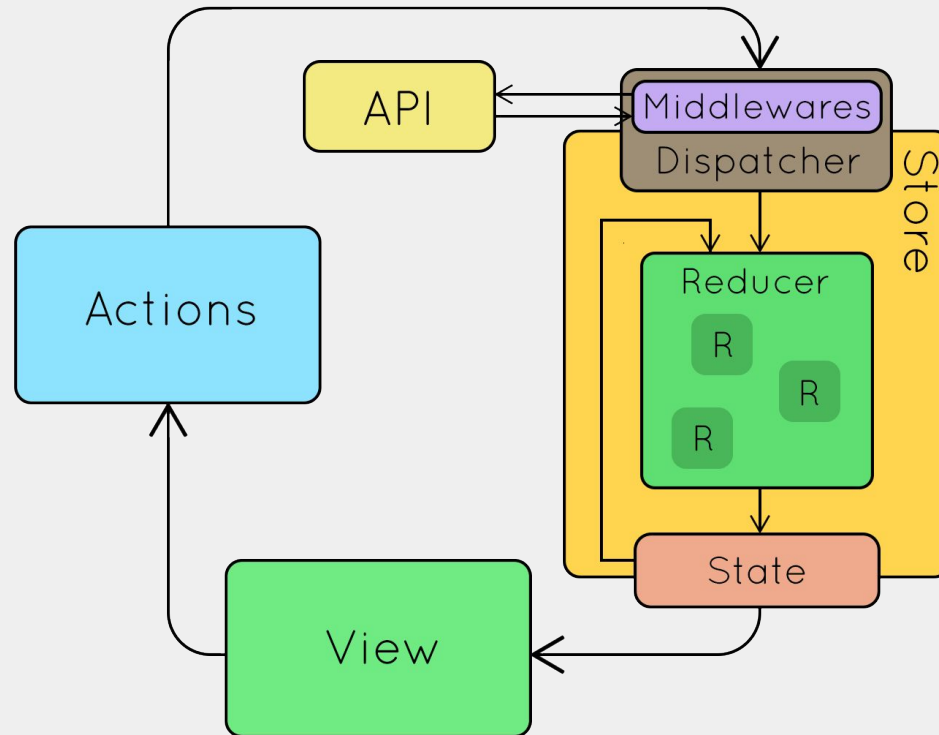




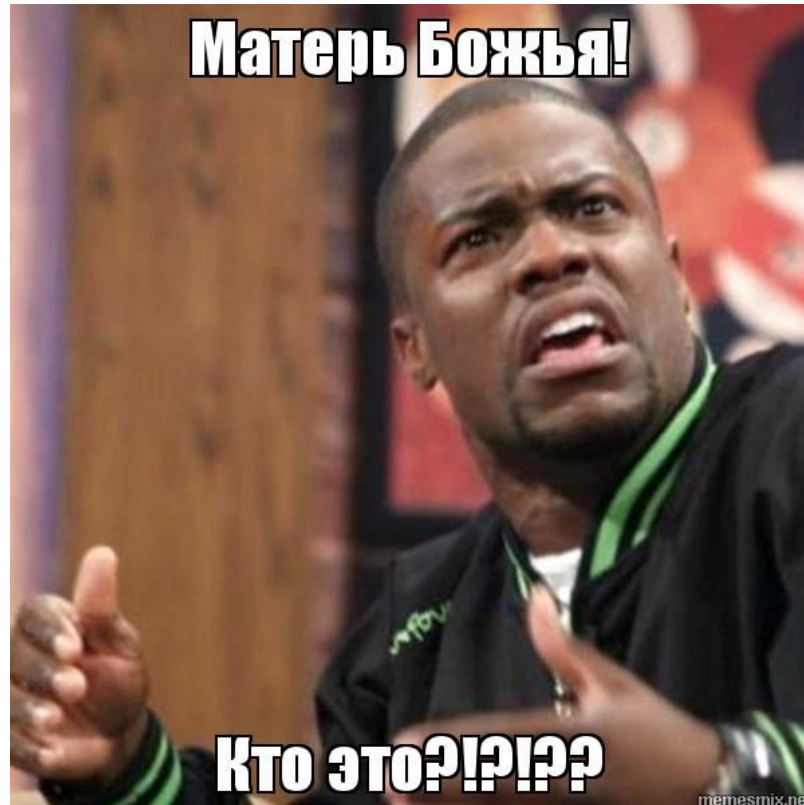
# middleware



# Redux



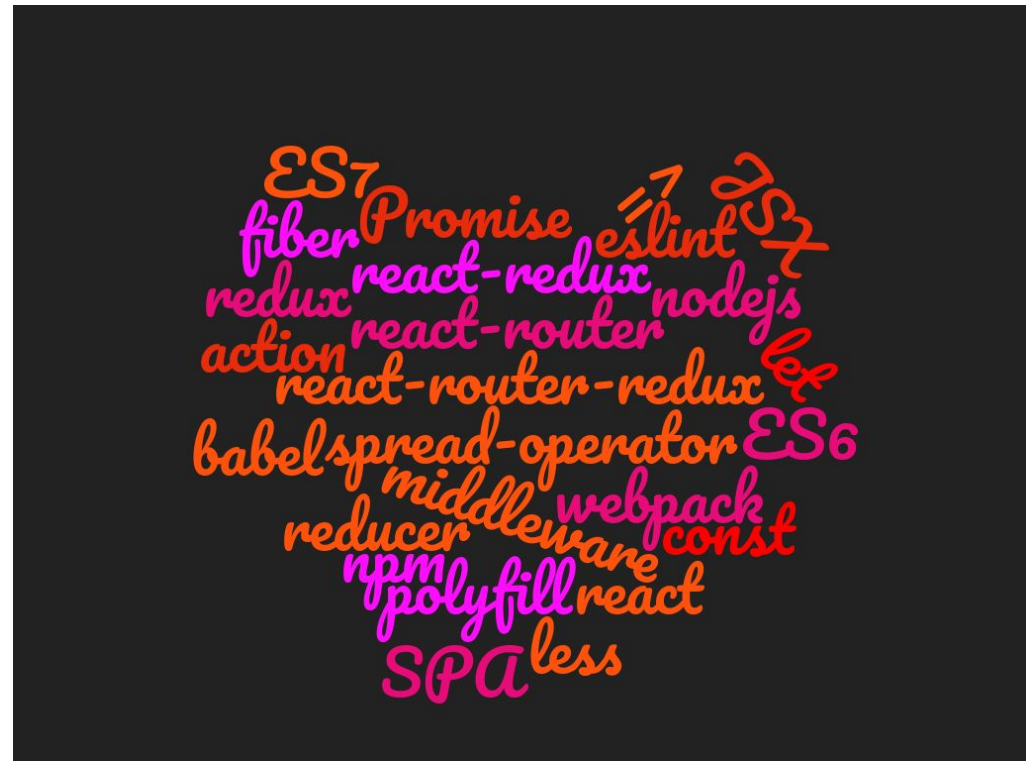
Babel? JSX? React?



# Библиотеки и технологии

С React используют следующее:

- ES6
- npm
- babel
- webpack
- react-router
- redux



# ES6

ECMAScript (ES) - язык программирования  
ECMA-262 - спецификация языка ECMAScript

ES 2016 года, 7ое издание => ES2016 = ES7, а ES2015 = ES6

JavaScript - реализация ES

<http://www.ecma-international.org/ecma-262/6.0/>

<https://github.com/tc39/ecma262#ecmascript>

# NPM

NPM - пакетный менеджер для NodeJS

Раньше подключение JS библиотеки выглядело так:

1. Найти и скачать нужную библиотеку в интернете (jquery например)
2. Положить в проект
3. Подключить тегом script в html файле

Есть вариант с CDN, тогда скачивать ничего не нужно, но искать всё равно надо

# NPM

- Центральный репозиторий <https://www.npmjs.com/>
- package.json для описания зависимостей проекта
- node\_modules - директория, куда npm кладёт скачанные для проекта зависимости



# NPM

- `npm install <packagename>`
- `npm i --save <pname>` (автоматически добавит в `package.json`)
- `npm i -g <pname>` (установить глобально)
- `npm run start` (запустить таргет `start`)
- `~/.npmrc` или `npm config` - конфиг (например для проху)

<https://github.com/npm/npm>

<https://habrahabr.ru/post/133363/>

<https://habrahabr.ru/post/243335/>

```
package.json
{
  "name": "my_package",
  "version": "1.0.0",
  "engine": {
    "node": ">=6",
    "npm": ">=3.10.1"
  },
  "dependencies": {
    "react": "^15.3.1",
    "react-router": "^2.8.1",
  },
  "devDependencies" : {
    "babel-core": "^6.13.2",
    "webpack": "^1.13.1",
  }
  "scripts": {
    "start": "\"npm run build\" && \"npm run watch\"",
    "build": "webpack --config webpack.config.js",
    "watch": "webpack --watch",
  }
}
```



# Babel

Babel - транспайлер (компилятор), который преобразует один язык в другой.

Например, можно писать код на ES6 и преобразовывать его в ES5, чтобы оно работало в IE.

```
npm install --save-dev babel-loader babel-core
```

<https://babeljs.io/>

<https://learn.javascript.ru/es-modern-usage#babel-js>

<https://habrahabr.ru/post/330018/>

# Babel

Зачем?

Браузеры внедряют новые языковые фичи медленно

А что он умеет?

ES2015, JSX, Typescript, Coffeescript, Clojurescript и др.



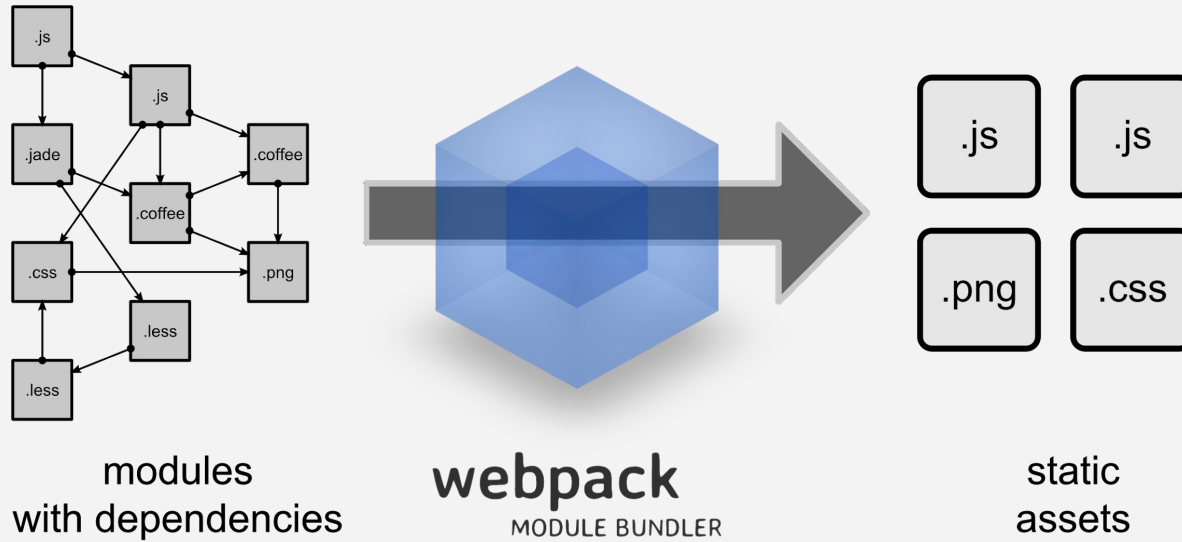
# Webpack

- Сборщик модулей
- Преобразует множество модулей(библиотек) в единый bundle
- Строит граф зависимостей
- `npm i -g webpack`
- Конфигурационный файл `webpack.config.js`
- Модули и плагины - точки расширения для управление сборкой и дополнительной обработкой модулей (минимизация, css-препроцессоры, `babel` и прочее)
- `webpack-dev-server`
- `hot-reload`

<https://webpack.js.org/concepts/>

<https://habrahabr.ru/post/245991/>

# Webpack



# Webpack

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via
npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');
const config = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'bundle.js'
  },
  module: {
    loaders:
      [ /* Следующий слайд */],

  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
module.exports = config;
```

# Webpack

```
module: {
  loaders:
    [{
      test: /\.jsx?$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: {
        presets: ['react', 'es2015', 'stage-0'],
      },
    },
    {
      test: /\.less$/,
      loader: ExtractTextPlugin .extract('style-loader',
      'css-loader!less-loader'),
    },
  ],
}
```