**Материалы к вебинару "Разработка SPA на React, NodeJS, Express и MongoDB"**

**🎉🎉🎉**[**6-недельный онлайн-курс по React.js**](http://frontend-science.com/react/)**🎉🎉🎉**

**План вебинара**

* Краткое введение в NodeJS и npm
* Написание простого сервера на NodeJS
* Кратко о MongoDB, взаимодействие с базой данных
* Написание компонентов на ReactJS
* Кратко об архитектуре Flux
* Получение и обработка данных с API
* Распределение данных внутри приложения
* Динамика в приложении

**Вспомогательные материлы**

В рамках данного вебинара мы напишем небольшое приложение для создания и хранения заметок ([такое](https://github.com/krambertech/spa-webinar/blob/master/cc%D1%8B%D0%BB%D0%BA%D0%B0)) на NodeJS, Express, MongoDB и React.

**1. Введение в Node.js и npm**

Node.js - программная платформа, основанная на движке Javascript V8. Она позволяет выполнять Javascript код на сервере.

Первое, что нужно сделать - это установить Node.js. Вместе с ним в комплекте идет также и npm (Node Package Manager) - менеджер пакетов. В зависимости от используемой операционной системы, у Вас естьтакие варианты:

* Для Windows: [скачать .msi установщик](https://nodejs.org/en/download)
* Для Linux: запустить curl -L https://npmjs.org/install.sh | sh в терминале
* Для Mac: [скачать .pkg установщик](https://nodejs.org/en/download) или brew install node, если используете [Homebrew](http://brew.sh/)

После установки, можно проверить наличие Node.js и npm на Вашем компьютере.

node -v

v5.1.0

npm -v

3.3.12

**2. Hello world на Node.js**

Давайте приступим к делу и напишем наше первое Node.js приложение «Hello world».

Откройте любой редактор и создайте файл под названием helloworld.js. В нем мы хотим вывести строку «Hello world» в консоль, для этого пишем следующий код:

console.log("Hello World");

Сохраняем файл и выполняем его посредством Node.js из терминала:

node helloworld.js

Эта команда должна вывести Hello World в вашем терминале.

**3. Постановка задачи**

Мы будем создавать приложения для работы с заметками с таким функционалом:

1. Добавить заметку

2. Просмотреть заметки

3. Удалить заметку

В самом начале, нам нужно спроектировать API сервера. Исходя из определенного выше функционала и опираясь на методолгию REST, нам понадобятся такие вызовы:

`GET /notes` - получить все заметки

`POST /notes` - создать новую заметку

`DELETE /notes/:id` - удалить заметку

**4. Использование Express**

[Express](http://expressjs.com/) - это минималистичный и гибкий веб-фреймворк для приложений Node.js, предоставляющий обширный набор функций. Он значительно упрощает маршрутизацию, использование промежуточных обработчиков (middleware), обработку запросов и отладку.

Создадим новую папку для проекта. Добавим в нее файл package.json приблизительно такого содержания:

{

"name": "NotesApp",

"version": "0.0.1",

"description": "Notes application",

"scripts": {

"server": "babel-node server/app.js"

},

"author": {

"name": "Kateryna Porshnieva",

"email": "k.porshnieva@gmail.com",

"url": "https://github.com/krambertech"

},

"dependencies": {

"body-parser": "\*",

"cors": "\*",

"express": "\*",

"mongoose": "\*"

},

"devDependencies": {

"babel": "5.x",

"babel-loader": "5.x"

}

}

Создадим папку /server - в ней будет храниться весь код, относящийся к серверной части приложения (бекенд). В ней создадим файл app.js - главный файл нашего сервера.

**Примечание:** Весь последующий код будет написан в стандарте ES2015 (ES6).

import express from 'express';

const app = express();

const server = app.listen(8080, () => {

console.log(`Server is up and running on port 8080`);

});

Здесь мы создали express-приложение и запустили веб-сервер на порте 8080. Для того, чтобы запустить его нужно выполнить в терминале babel-node server/app.js.

Пока что наш сервер ничего не умеет делать, давайте создадим для него новый маршрут:

import express from 'express';

const app = express();

app.get('/', (req, res) => {

res.send('Hello World!');

});

const server = app.listen(8080, () => {

console.log(`Server is up and running on port 8080`);

});

Теперь если открыть http://localhost:8080/ то вы увидите надпись "Hello world". Маршруты в Express описываются очень просто:

app.get('/grapes', (req, res) => {});

app.post('/grapes', (req, res) => {});

app.put('/grapes', (req, res) => {});

app.delete('/grapes', (req, res) => {});

Вы пишете app.метод(маршрут, функция). Маршрут затем будет трансформирован в резулярное выражение, чтобы сервер понимал какие запросы как нужно обработать. Функция принимает объекты req ([объект запроса](http://expressjs.com/ru/4x/api.html#req)), res ([объект ответа](http://expressjs.com/ru/4x/api.html#res)) и next (функцию для вызова следующего обработчика).

**5. MongoDB и mongoose**

MongoDB - это документо-ориентированная СУБД. Данные в MongoDB хранятся в документах, которые объединяются в коллекции. Каждый документ представляет собой JSON-подобную структуру (BSON). Проведя аналогию с реляционными СУБД, можно сказать, что коллекциям соответствуют таблицы, а документам — строки в таблицах. В отличие от РСУБД MongoDB не требует какого-либо описания схемы базы данных — она может постепенно меняться по мере развития приложения, что есть удобно.

Cначала нужно установить MongoDB на свой компьютер. Cсылка для скачивания: <https://www.mongodb.org/downloads>

**Инструкция по установке на Windows:** <http://metanit.com/nosql/mongodb/1.2.php>

Mongoose — самый популярный модуль для работы с mongodb на javascript. Он позволяет очень удобно работать с базой данных, в нем все строится на схемах данных. То есть вы создаете модель хранимых данных в базе, а он уже помогает их типизировать, валидировать, строить бизнес логику поверх них, создавать запросы и т.д.

В базе данных мы будем хранить заметки. Так как в mongoose все строится на моделях данных, нам нужно создать модель (схему) для заметок. Делается это очень просто:

import mongoose from "mongoose";

const Schema = mongoose.Schema;

const NoteSchema = new Schema({

title : { type: String },

text : { type: String, required: true },

color : { type: String },

createdAt : { type: Date }

});

const Note = mongoose.model('Note', NoteSchema);

Есть огромное множество встроенных типов и приемов для работы с данными - все их вы можете найти в документации - <http://mongoosejs.com/docs/>

Сейчас мы создали схему для заметки, а потом, на основе этой схемы, модель. Модель - это класс, с помощью которого будут строиться документы в коллекции. В этом случае, каждый документ будет заметкой с параметрами и поведением, которое мы определили. Например, у каждой заметки есть заголовок, текст, цвет и дата создания. Из всех полей, только тект является обязательным - он помечен required.

Также к схемам можно добавлять методы:

var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/kittens');

var kittySchema = mongoose.Schema({

name: String

});

kittySchema.methods.speak = function () {

var greeting = this.name

? "Meow name is " + this.name

: "I don't have a name";

console.log(greeting);

}

var Kitten = mongoose.model('Kitten', kittySchema);

var fluffy = new Kitten({ name: 'fluffy' });

fluffy.speak();

Мы с вами создали говорящих котят. Для того, чтобы сохранить Флаффи в базу данных, нам нужно просто написать:

fluffy.save(function (err, fluffy) {

if (err) return console.error(err);

fluffy.speak();

});

Теперь для того, чтобы получить всех котят, мы можем использовать метод find:

Kitten.find(function (err, kittens) {

if (err) return console.error(err);

console.log(kittens);

})

Чтобы найти всех котят по имени fluffy:

Kitten.find({ name: "fluffy" }, callback);

Теперь давайте отойдем от котят и вернемся к нашим заметкам. Сохраним модель заметки в папке/server/models/Note.js. Теперь нужно связать взаимодействие с базой данных и запросы. Для работы с базой мы создадим утилиту server/utils/DataBaseUtils.js, в которой просто изолируем методы для работы с базой.

Сначала нам нужно настроить соединение:

import mongoose from "mongoose";

import '../models/Note';

const Note = mongoose.model('Note');

export function setUpConnection() {

mongoose.connect(`mongodb://localhost/notes`);

}

И добавим несколько методов для взаимодействия с базой:

export function listNotes(id) {

return Note.find();

}

export function createNote(data) {

const note = new Note({

title: data.title,

text: data.text,

color: data.color,

createdAt: new Date()

});

return note.save();

}

export function deleteNote(id) {

return Note.findById(id).remove();

}

Теперь нужно создать соответствующие маршруты. Вернемся к файлу /server/app.js.

Данные мы будем получать в формате json, для того, чтобы удобно с ними взаимодействовать мы будем использовать промежуточный обработчик [body-parser](https://github.com/expressjs/body-parser). Он будет вызван всякий раз, когда прийдет запрос - он сначала преобразует данные, а затем передаст управление нашим обработчикам.

Используется он таким образом:

import bodyParser from 'body-parser';

// ...

app.use( bodyParser.json() );

Теперь данные, полученые в json формате будут обработаны корректно.

В итоге, наш файл app.js выглядит вот так:

import express from 'express';

import bodyParser from 'body-parser';

import \* as db from './utils/DataBaseUtils';

// Initialization of express application

const app = express();

// Set up connection of database

db.setUpConnection();

// Using bodyParser middleware

app.use( bodyParser.json() );

// RESTful api handlers

app.get('/notes', (req, res) => {

db.listNotes().then(data => res.send(data));

});

app.post('/notes', (req, res) => {

db.createNote(req.body).then(data => res.send(data));

});

app.delete('/notes/:id', (req, res) => {

db.deleteNote(req.params.id).then(data => res.send(data));

});

const server = app.listen(8080, function() {

console.log(`Server is up and running on port 8080`);

После обращения к базе мы используем промисы (Promise), они предоставляют удобный способ организации асинхронного кода.

На этом почти все, мы создали серверное приложение на node.js. Еще было бы неплохо создать для него конфигурацию, то есть вынести некоторые переменные в конфиг. Создадим файл /etc/config.json:

{

"apiPrefix": "http://localhost:8080",

"serverPort": "8080",

"db":{

"name": "notes",

"host": "localhost",

"port": 27017

}

}

Теперь при использовании этих переменных, будем обращаться к этому файлу.

**app.js**

import express from 'express';

import cors from 'cors';

import bodyParser from 'body-parser';

import { serverPort } from '../etc/config.json';

import \* as db from './utils/DataBaseUtils';

// Initialization of express application

const app = express();

// Set up connection of database

db.setUpConnection();

// Using bodyParser middleware

app.use( bodyParser.json() );

// Allow requests from any origin

app.use(cors({ origin: '\*' }));

// RESTful api handlers

app.get('/notes', (req, res) => {

db.listNotes().then(data => res.send(data));

});

app.post('/notes', (req, res) => {

db.createNote(req.body).then(data => res.send(data));

});

app.delete('/notes/:id', (req, res) => {

db.deleteNote(req.params.id).then(data => res.send(data));

});

const server = app.listen(serverPort, function() {

console.log(`Server is up and running on port ${serverPort}`);

});

**DataBaseUtils.js**

import mongoose from "mongoose";

import config from '../../etc/config.json';

import '../models/Note';

const Note = mongoose.model('Note');

export function setUpConnection() {

mongoose.connect(`mongodb://${config.db.host}:${config.db.port}/${config.db.name}`);

}

export function listNotes(id) {

return Note.find();

}

export function createNote(data) {

const note = new Note({

title: data.title,

text: data.text,

color: data.color,

createdAt: new Date()

});

return note.save();

}

export function deleteNote(id) {

return Note.findById(id).remove();

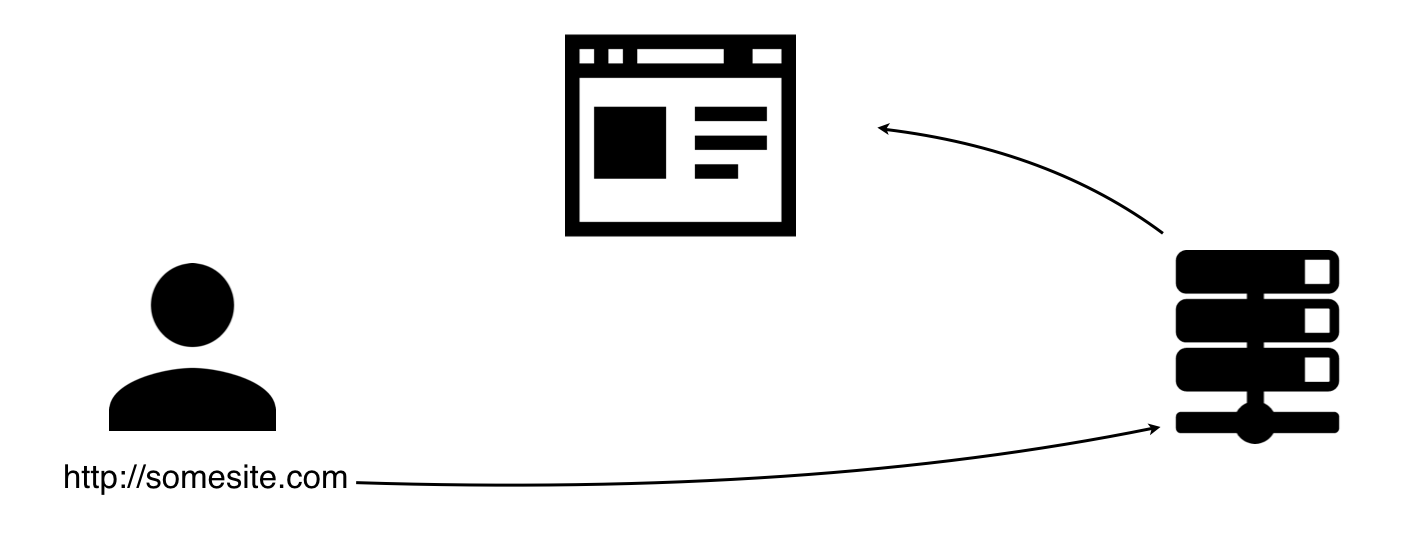
}

**6. Введение в React**

ReactJS - это JаvaScript библиотека для построения пользовательских интерфейсов. Это не MVC фреймворк. К нему можно применить только V из этой аббревиатуры. Такая узкая сфера применения дает свободу использования React в различных системах в комбинации с другими библиотеками.

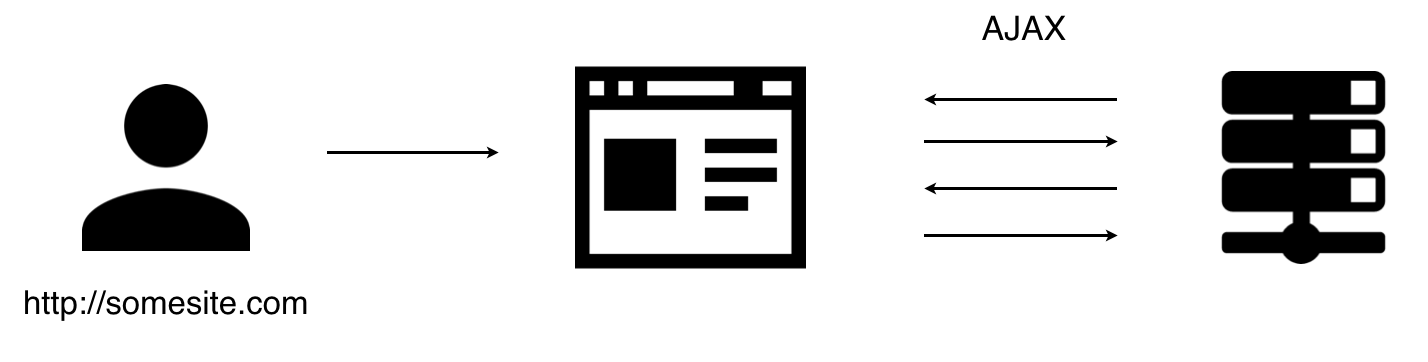
React был представлен Facebook в 2013 году, и очень быстро обрел популярность. Сегодня его используют многие известные компании включая Instagram, Airbnb, Ebay, Netflix, Yahoo и другие.

Основным отличием React от других JavaScript фреймворков является то, как он управляет состоянием приложения. Если вспомнить, как пользователи взаимодействовали с веб-страницами еще 10-15 лет назад, то увидим такую картину:

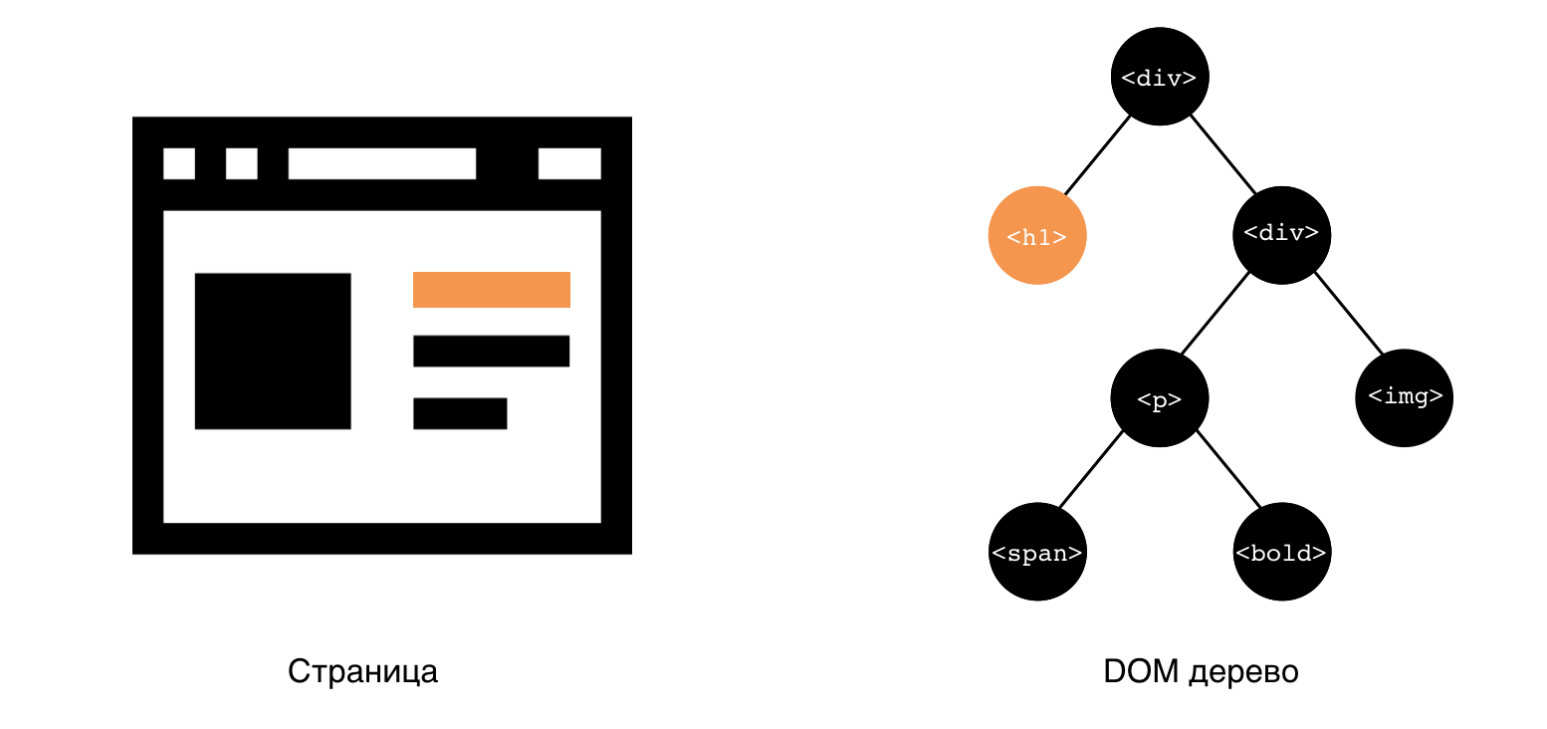
[](https://github.com/krambertech/react-essential-course/raw/master/01-introduction-to-react/images/001.png)

Сервер всегда возвращал статическую страницу, и реакцией на действия пользователя была полная перезагрузка страницы. Преимуществами такого подхода, была простота в реализации и понимании, недостатками - скорость работы, отзывчивость, UX и потеря состояния при каждой перезагрузке.

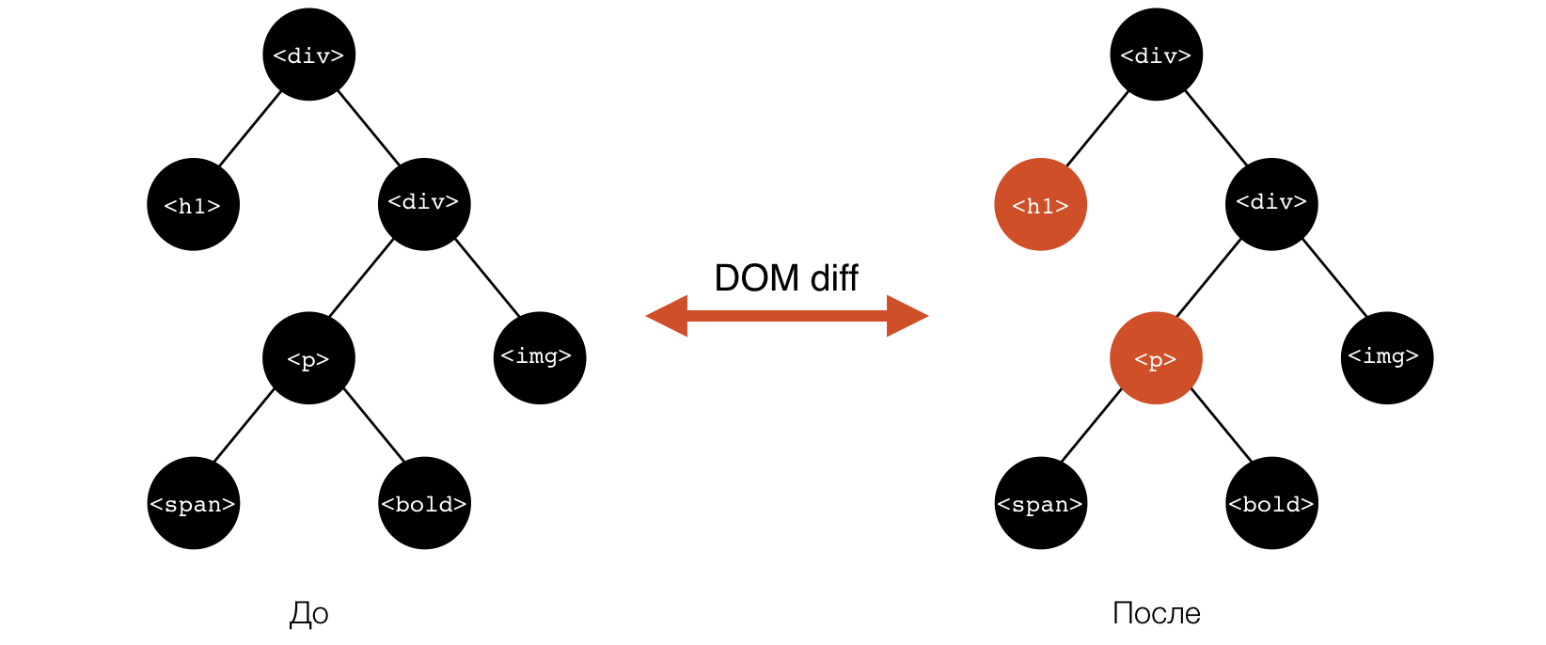
Все очень изменилось с появлением AJAX, это подход к построению интерактивных веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером. То есть, в фоновом режиме отправляются запросы на сервер, приходят с него ответы, изменяется состояние приложения и, соответственно, внешний вид. Именно такой подход породил понятие Single Page Application.

[](https://github.com/krambertech/react-essential-course/raw/master/01-introduction-to-react/images/002.png)

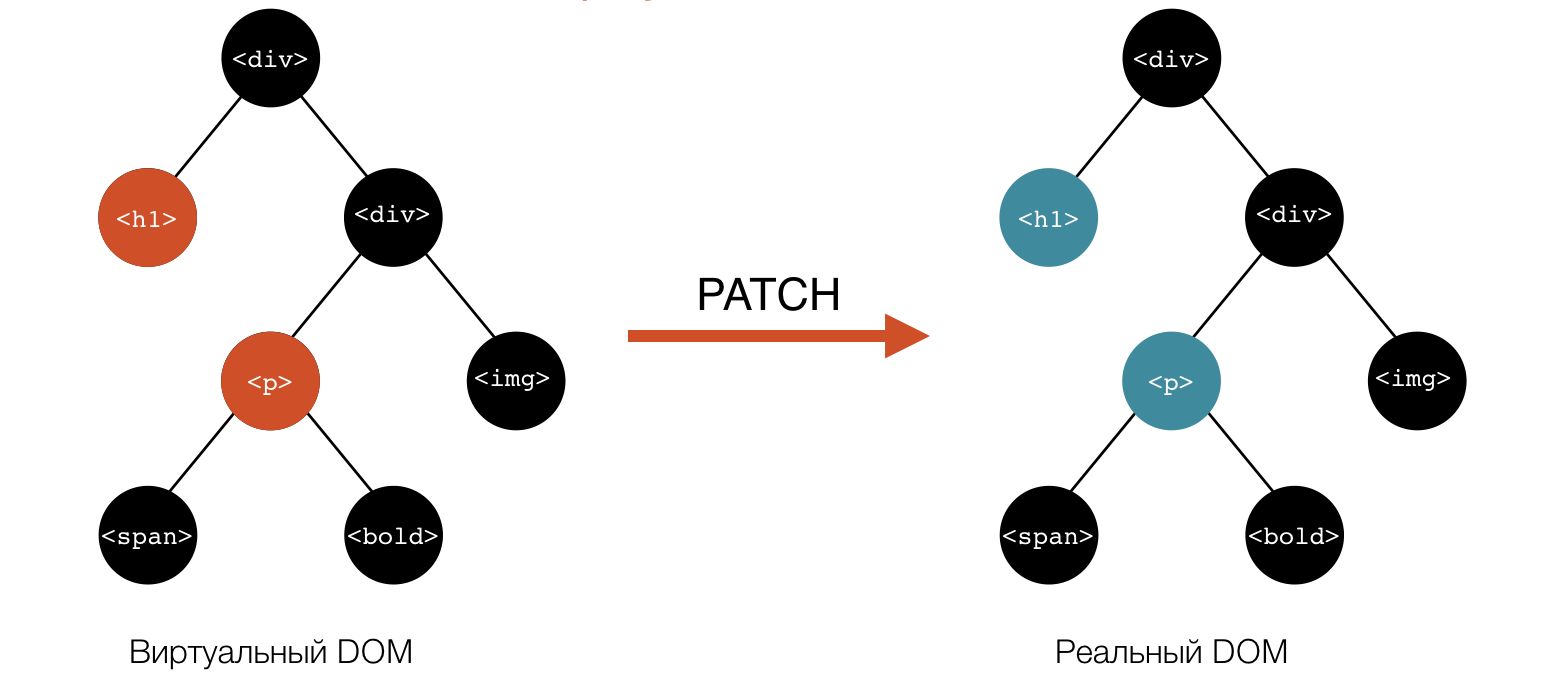
Но каждое визуальное изменение на странице соответствует изменению ее DOM дерева. Не секрет, что все манипуляции с DOM деревом являются очень ресурсоемкими операциями, т.к. изначально DOM дерево было статическим и никакой динамики не предусматривало.

[](https://github.com/krambertech/react-essential-course/raw/master/01-introduction-to-react/images/006.png)

Именно поэтому в React используется виртуальный DOM. Это такая легковесная копия реального DOM дерева на Javascript. Таким образом, React манипулирует не с реальным (синоним - медленным) DOM деревом, а с виртуальным.

[](https://github.com/krambertech/react-essential-course/raw/master/01-introduction-to-react/images/003.png)

Он сравнивает предыдущее состояние виртуального DOM дерева с его следующим состоянием и находит минимальное количество манипуляций, которые можно произвести уже с реальным DOM, чтобы обновить вид приложения согласно его новому состоянию.

[](https://github.com/krambertech/react-essential-course/raw/master/01-introduction-to-react/images/004.png)

И это действительно быстро работает. А все что вам нужно делать - это просто менять состояние вашего приложения, а все остальное React сделает уже за вас!

Все приложение на React строится из небольших кирпичиков - компонентов. Такое построение приложения упрощает поддержку и работу над кодом в команде. В идеале, все компоненты должны быть независимыми и каждый должен делать свою задачу и делать ее хорошо.

**JSX**

Для рендеринга данных в React используется JSX. JSX нужен для JavaScript XML — разметки в стиле XML внутри компонентов React. React работает и без JSX, но именно JSX поможет сделать ваши компоненты более читаемыми, поэтому рекомендуется использовать его.

// с JSX

var app = <Nav color=“blue">

<Profile>click</Profile>

</Nav>;

// без JSX

var app = React.createElement(

Nav,

{color:"blue"},

React.createElement(

Profile,

null,

"click"

)

);

JSX позволяет вам описывать структуру компонентов с помощью понятного синтаксиса, а затем все написанное вами преобразуется в цепочку javascript функций.

В JSX можно использовать переменные, условные конструкции и вызывать функции. Для этого нужно использовать фугурные скобки.

var myName = 'Katya';

var app = <h1> My name is {myName}! </h1>;

function getMyName() {

return 'Katya';

}

var app = <h1>

My name is {getMyName()}!

</h1>;

var age = 20;

var app = <h1>

Hi! { age > 18 ? 'Your age is more than 18!' : 'Your age is less than 18!' }

</h1>;

**Компоненты**

React использует компонентную модель, то есть все большое приложение делится на небольшие независимые компоненты, которыми гораздо легче управлять.

// простой компонент

var HelloWorld = React.createClass({

render: function() {

return (

<h1> Hello world! </h1>

);

}

});

У каждого компонента есть один обязательный метод - render, который возвращает JSX разметку, соответствующую виду компонента.

**Параметры aka props**

Каждый компонент может принимать параметры. Они передаются из выше стоящих компонентов. К параметрам компонента можно обращаться используя this.props.propName.

var Heading = React.createClass({

render: function() {

return <h1> My name is {this.props.name}!</h1>;

}

});

var Hello = React.createClass({

render: function() {

return <Heading name="Katya" />;

}

});

Запомните: Модифицировать this.props **крайне нежелательно**\*!

**Состояние aka state**

Также каждый компонент может хранить свое состояние. К нему можно получить доступ, обратившись к this.state. В состоянии компонента стоит хранить данные, от которых компонент напрямую зависит внешний вид компонента, и при изменении которых его внешний вид тоже должен меняться (будет вызываться метод render). Cостояние компонента доступно только внутри самого компонента. Для объявления начального состояния компонента, нужно использовать метод getInitialState. Этот метод вызывается до того, как компонент отобразиться в доме и определяет первоначальное значение состояния компонента. Для того, чтобы модифицировать состояние компонента нужно вызвать метод this.setState({ /\* новое состояние \*/ }), тогда состояние компонента измениться и вызовется метод render.

var Component = React.createClass({

getInitialState : function() {

return {

name : "Katya"

};

},

handleClick : function() {

this.setState({

name : "Vasya"

});

},

render : function() {

return <div onClick={this.handleClick}>

Hello, {this.state.name}

</div>;

}

});

**Состояние нужно использовать только там, где это действительно необходимо!**

**НИКОГДА** не нужно модифицировать this.state напрямую!

**Обработка событий**

Если вы захотите сделать свои компоненты динамическими, то вам не обойтись без использования событий. Обычно, для каждого события описывается обработчик, в котором вы можете произвести какие-то действия.

var HelloComponent = React.createClass({

handleClick : function() {

alert('Hello stranger!');

},

render : function() {

return <button onClick={this.handleClick}> Say hello </button>;

}

});

Во все такие обработчики событий в качестве аргумента приходит объект SyntheticEvent. Это объект, который является кроссбраузерной оберткой над стандартными событиями.

var EventComponent = React.createClass({

handleClick : function(event) {

alert('Event handled - ' + event.type); // Event handled - click

},

render : function() {

return <button onClick={this.handleClick}> Click me </button>;

}

});

О том, какие события поддерживаются можно прочитать [тут](https://facebook.github.io/react/docs/events.html).

**ReactJS Hello world**

Начнем с простого подключения React в html документе

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Hello World!</title>

<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.6.15/browser.js"></script>

</head>

<body>

<div id='content'></div>

<script type="text/babel">

const HelloWorld = React.createClass({

render() {

return (

<h1> Hello world! </h1>

);

}

});

ReactDOM.render(

<HelloWorld />,

document.getElementById('content')

);

</script>

</body>

</html>

В параметрах мы можем передавать значения компоненту, которые он будет уже использовать внутри себя, обращаясь к this.props.

const HelloWorld = React.createClass({

render() {

return (

<h1> Hello, {this.props.name}! </h1>

);

}

});

ReactDOM.render(

<HelloWorld name='Vasya' />,

document.getElementById('content')

);

Также мы можем добавить динамику в компонент с помощью использования состояния и обработки событий.

const HelloWorld = React.createClass({

getInitialState() {

return {

name: 'Vasya'

};

},

handleNameChange(e) {

this.setState({

name: e.target.value

});

},

render() {

return (

<div>

<h1> Hello {this.state.name}! </h1>

<input type='text' value={this.state.name} onChange={this.handleNameChange} />

</div>

);

}

});

ReactDOM.render(

<HelloWorld />,

document.getElementById('content')

);

**Жизненный цикл компонента**

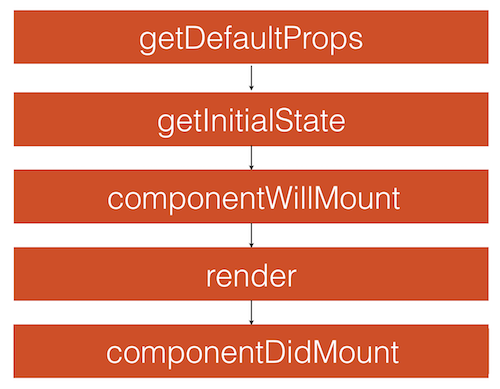
В React есть относительно немного методов жизненного цикла, но все они очень мощные. React дает вам все необходимые методы для контроля свойств и состояния приложения в процессе его жизни.

Есть всего 4 сценария, когда методы жизненного цикла могут быть использованы:

1. Инициализация компонента
2. Изменение его параметров (props)
3. Изменение его состояния (вызов setState)
4. Удаление компонента

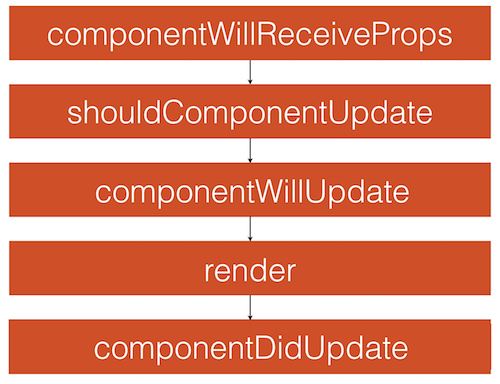
**Инициализация компонента (первый render)**

При первом render компонента методы жизненного цикла будут вызваны в таком порядке:

[](https://github.com/krambertech/react-essential-course/raw/master/02-deep-in-components/images/001.png)

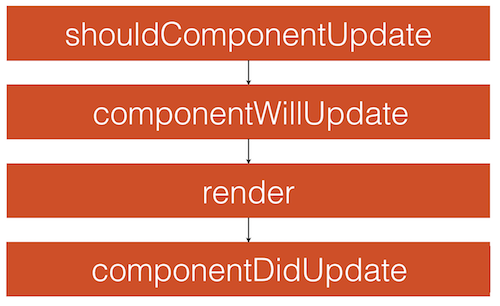
**Изменение параметров (props)**

Когда от родительского компонента приходят измененные параметры, последовательность вызова методов жизненного цикла такая:

[](https://github.com/krambertech/react-essential-course/raw/master/02-deep-in-components/images/002.png)

**Изменение состояния (вызов setState)**

Когда в компоненте изменяется состояние, то методы жизненного цикла вызываются в таком порядке:

[](https://github.com/krambertech/react-essential-course/raw/master/02-deep-in-components/images/003.png)

**Удаление компонента**

Перед удаление компонента из DOM будет вызван один единственный метод сomponentDidMount

[](https://github.com/krambertech/react-essential-course/raw/master/02-deep-in-components/images/004.png)

**Методы жизненного цикла**

**getDefaultProps**

Вызывается единожды при инициализации класса. Отвечает за значения параметров по умолчанию.

getDefaultProps: function() {

return {

name: ‘’,

age: 0

};

}

**getInitialState**

Создан для определения начального состояния компонента.

getInitialState: function() {

return {

isOpened: true

};

}

**componentWillMount**

Вызывается один раз прямо **перед** тем, как состоится первый render компонента. Вызов setState в рамках данного метода дополнительного рендера не вызовет.

componentWillMount: function() {

// ...

}

**componentDidMount**

Вызывается один раз прямо сразу **после** того, как состоялся первый render компонента.

componentDidMount: function() {

// компонент уже находится в DOM

// здесь можно уже взаимодействовать с DOM напрямую

// например, использовать jQuery или какие-то сторонние библиотеки

}

**componentWillReceiveProps**

Вызывается каждый раз, когда компонент получает **новые параметры**. Не вызывается для первого рендера. ВызовsetState в рамках данного метода дополнительного рендера не вызовет.

componentWillReceiveProps: function(nextProps) {

// в nextProps содержится объект с новыми параметрами

// старые параметры можно получить использование this.props

this.setState({

likesIncreasing: nextProps.likeCount > this.props.likeCount

});

}

**shouldComponentUpdate**

Вызывается при изменении параметров или состояния. Возвращает true (если изменение должно вызвать перерисовку компонента) или false (если изменение не влияет на отображение компонента).

shouldComponentUpdate: function(nextProps, nextState) {

return nextProps.id !== this.props.id;

}

Если shouldComponentUpdate возвращает false, то метод render() будет пропущен до следующего изменения параметров или состояния. По умолчанию (если не определен), всегда возвращает true. Может быть использован для улучшения бустродействия приложения (чтобы избежать лишних перерисовок), особенно, если используется огромное количество компонентов.

**componentWillUpdate**

Вызывается **перед** вызовом метода render() при изменении параметров или состояния компонента.

componentWillUpdate: function(nextProps, nextState) {

// в nextProps содержится объект с новыми параметрами

// в nextState содержится объект с измененным состоянием

}

**!!!** Не используйте setState() в этом методе! Так у вас может произойти зацикливание!

**componentDidUpdate**

Вызывается **сразу после** вызова метода render() при изменении параметров или состояния компонента.

componentDidUpdate: function(prevProps, prevState) {

// в prevProps содержится объект с предыдущими параметрами

// в prevState содержится объект с состоянием до изменения

// измененные параметры и состояние могут быть получены через this.props и this.state

}

Произведенные изменения уже отображены в DOM дереве. Обычно, в данном методе производят какие-то операции с DOM елементами согдасно изменениям.

**!!!** Не используйте setState() в этом методе! Так у вас может произойти зацикливание!

**componentWillUnmount**

Вызывается перед тем, как компонент будет удален из DOM.

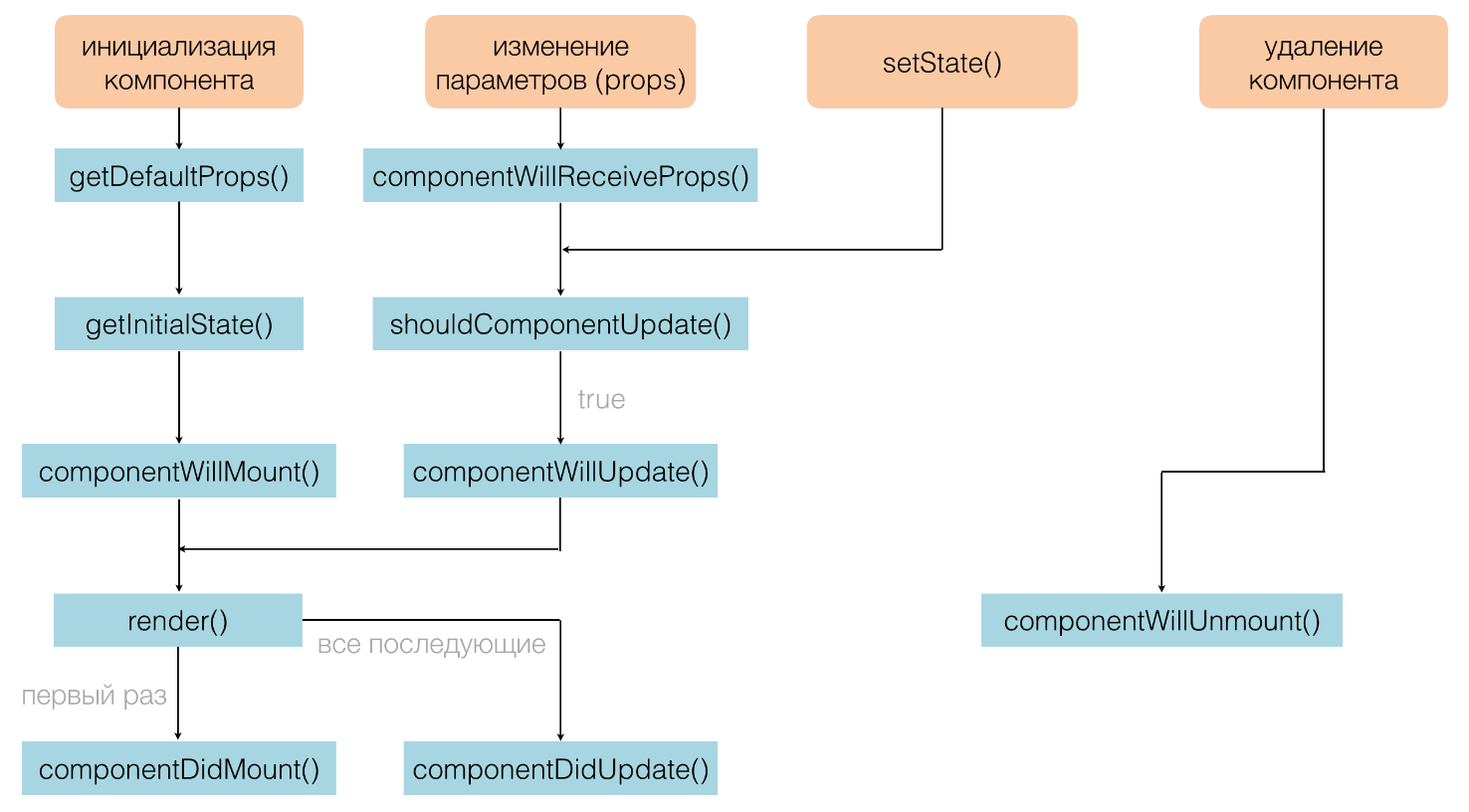
componentWillUnmount: function() {

// обычно, в данном методе происходит некая уборка за компонентом

// остановка таймеров, удаление ссылок на DOM елементы и т.д.

}

**Весь жизненный цикл компонента можно представить в виде такой схемы**

[](https://github.com/krambertech/react-essential-course/raw/master/02-deep-in-components/images/005.png)

**Сборка приложения**

Для более сложных проектов уже не выйдет все компоненты описать в index.html. Обычно создается отдельный файл для компонента, то компонент Button будет находиться в файле Button.jsx, а если у него еще есть отдельный файл со стилями, то мы создадим файл Button.css и расположим рядом с файлом компонента и подключим в нем. Основная идея такого подхода стоит в том, чтобы максимально изолировать компонент и все относящиеся к нему зависимости.

Тогда создадим в нащем проекте папку /client, в которой будет располагаться весь код, относящийся к клиентской части приложения. Также мы должны создать главный файл проекта, так называемую "точку входа". Из которого мы будем уже подключать другие модули, а они следующие и так далее. Создадим файл main.js.

import React from 'react';

import ReactDOM from 'react-dom';

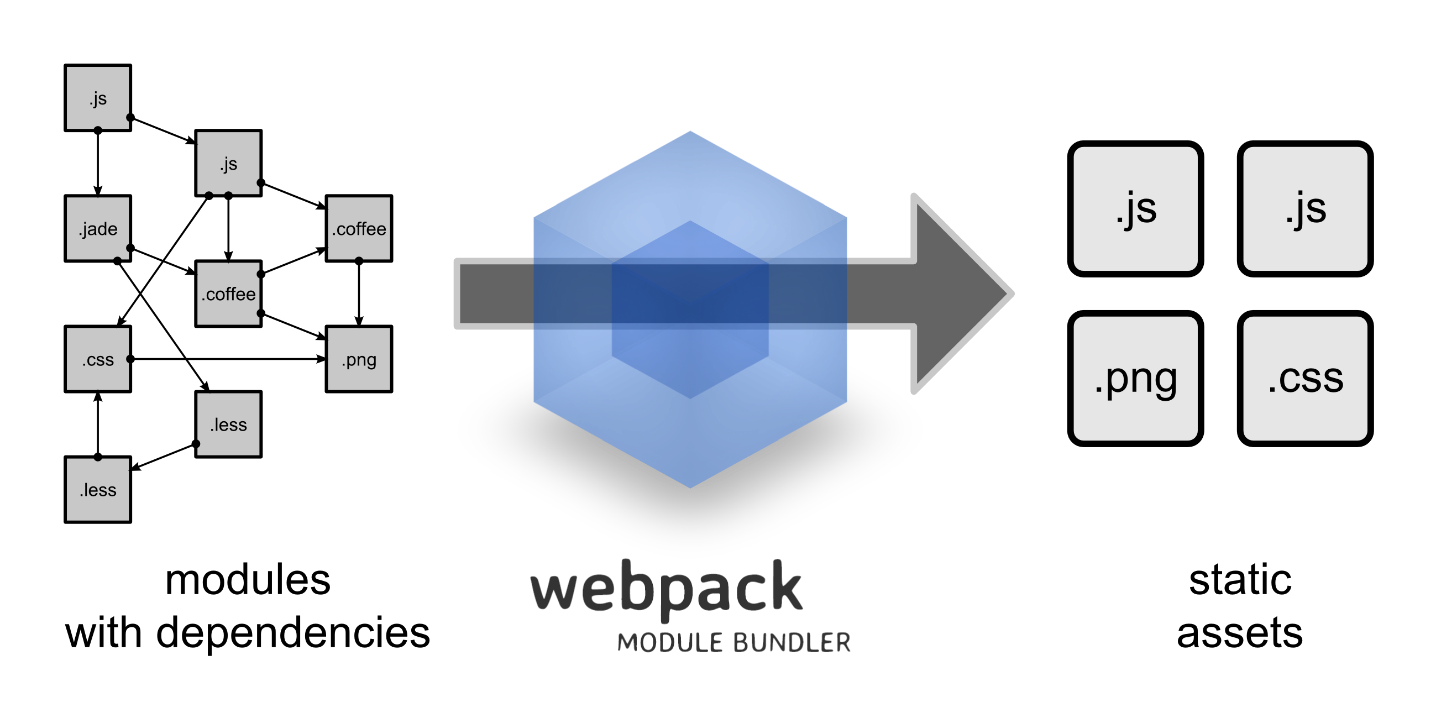
ReactDOM.render(

<h1>Notes</h1>,

document.getElementById('mount-point')

);

Для того, чтобы кучу совершенно разных кусочков проекта (стили, js, json, jsx и т.д.) собрать воедино существуют системы сборки. Их сейчас есть достаточно много: gulp, grunt, broccoli, browserify и т.п. Но мы будем говорить о webpack. Почему именно он? Все очень просто - это самое универсальное и комплексное решение.

[](https://camo.githubusercontent.com/0d3c9687ac274c17b7cf573b518bf06abd88e7ce/68747470733a2f2f7765627061636b2e6769746875622e696f2f6173736574732f776861742d69732d7765627061636b2e706e67)

Установим webpack глобально:

npm install webpack -g

При этом, в нем весь процесс сборки проекты описывается достаточно просто. Вот, например, как будет выглядеть наш конфиг:

var webpack = require('webpack');

module.exports = {

entry: "./client/main.js",

output: {

path: \_\_dirname + '/public/build/',

publicPath: "build/",

filename: "bundle.js"

},

module: {

loaders: [

{

test: /\.js$/,

loader: "babel",

exclude: [/node\_modules/, /public/]

},

{

test: /\.less$/,

loader: "style-loader!css-loader!autoprefixer-loader!less",

exclude: [/node\_modules/, /public/]

},

{

test: /\.jsx$/,

loader: "react-hot!babel",

exclude: [/node\_modules/, /public/]

},

{

test: /\.json$/,

loader: "json-loader"

}

]

}

}

Мы описывает то, что должно быть на входе и то, что мы хотим получить на выходе. Для осуществления преобразований файлов используются загрузчики.

Чтобы запустить сборку, нужно в терминале выполнить команду

webpack -p

Тогда появится файл bundle.js cо всем уже минифицированным кодом.

Также для webpack есть утилита webpack-dev-server для запуска development-сервера и отслеживания изменений в файлах. В него можно подключить как hot-reload - автоматическую перезагрузку страницы браузера при изменении файлов проекта, так и hot-module-replacement - возможность при изменении исходников очень быстро отобразить изменения в браузере без перезагрузки страницы, выглядит как магия.

Для запуска webpack-dev-server нам нужно выполнить такую команду:

webpack-dev-server --debug --hot --devtool eval-source-map --output-pathinfo --watch --colors --inline --content-base public --port 8090 --host 0.0.0.0

Но чтобы такое безобразие каждый раз неписать в консоли, давайте добавим ее в scripts в package.json.

"scripts": {

"webpack-devserver": "webpack-dev-server --debug --hot --devtool eval-source-map --output-pathinfo --watch --colors --inline --content-base public --port 8090 --host 0.0.0.0"

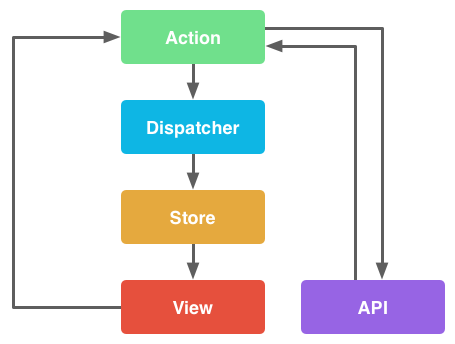
}

Теперь мы можем просто писать npm run webpack-devserver и данная команда будет выполнена.

**Flux**

Flux — это архитектура, которую команда Facebook использует при работе с React. Это не фреймворк, или библиотека, это новый архитектурный подход, который дополняет React и принцип однонаправленного потока данных.

Типичная реализация архитектуры Flux может использовать эту библиотеку вместе с классом EventEmitter из NodeJS, чтобы построить событийно-ориентированную систему, которая поможет управлять состоянием приложения.

[](https://camo.githubusercontent.com/77f80b4b46851a6cc86cf94be8a7f1becfb8bbaa/68747470733a2f2f686162726173746f726167652e6f72672f66696c65732f3464632f6131322f3934612f34646361313239346134633334616465613438626638646136316537613639322e706e67)

Facebook предоставляет библиотеку (npm i flux), которая содержит реализацию Dispatcher. Dispatcher по своей сути является event-системой. Он траслирует события и регистрирует колбэки. Есть только один глобальный dispatcher. Он очень легко инициализируется:

**client/dispatcher/AppDispatcher.js**

import { Dispatcher } from 'flux';

export default new Dispatcher();

В сущности, Диспетчер — это менеджер всего этого процесса. Это центральный узел вашего приложения. Диспетчер получает на вход действия и рассылает эти действия (и связанные с ними данные) зарегистрированным обработчикам.

Actions — хелперы, упрощающие передачу данных Диспетчеру. Это набор методов, которые вызываются из Представлений (или из любых других мест), чтобы отправить Действия Диспетчеру. В реализации Facebook Действия различаются по типу — константе, которая посылается вместе с данными действия. В зависимости от типа, Действия могут быть соответствующим образом обработаны в зарегистрированных обработчиках, при этом данные из этих Действий используются как аргументы внутренних методов.

Объявление констант:

**client/constants/AppConstants.js**

import keyMirror from 'keymirror';

export default keyMirror({

LOAD\_NOTES\_REQUEST: null,

LOAD\_NOTES\_SUCCESS: null,

LOAD\_NOTES\_FAIL: null

});

Мы использовали библиотеку keyMirror чтобы создать объект со значениями, идентичными своим ключам.

Действия же выглядят таким образом:

**client/actions/NotesActions.js**

import AppDispatcher from '../dispatcher/AppDispatcher';

import Constants from '../constants/AppConstants';

import api from '../api';

const NoteActions = {

loadNotes() {

AppDispatcher.dispatch({

type: Constants.LOAD\_NOTES\_REQUEST

});

api.listNotes()

.then(({ data }) =>

AppDispatcher.dispatch({

type: Constants.LOAD\_NOTES\_SUCCESS,

notes: data

})

)

.catch(err =>

AppDispatcher.dispatch({

type: Constants.LOAD\_NOTES\_FAIL,

error: err

})

);

},

createNote(note) {

api.createNote(note)

.then(() =>

this.loadNotes()

)

.catch(err =>

console.error(err)

);

},

deleteNote(noteId) {

api.deleteNote(noteId)

.then(() =>

this.loadNotes()

)

.catch(err =>

console.error(err)

);

}

};

export default NoteActions;

Хранилища в Flux управляют состоянием определенных частей предметной области вашего приложения. На более высоком уровне это означает, что Хранилища хранят данные, методы получения этих данных и зарегистрированные в Диспетчере обработчики Действий.

**client/stores/NotesStore.js**

import { EventEmitter } from 'events';

import AppDispatcher from '../dispatcher/AppDispatcher';

import AppConstants from '../constants/AppConstants';

const CHANGE\_EVENT = 'change';

let \_notes = [];

let \_loadingError = null;

let \_isLoading = true;

function formatNote(note) {

return {

id: note.\_id,

title: note.title,

text: note.text,

color: note.color || '#ffffff',

createdAt: note.createdAt

};

}

const NotesStore = Object.assign({}, EventEmitter.prototype, {

isLoading() {

return \_isLoading;

},

getNotes() {

return \_notes;

},

emitChange: function() {

this.emit(CHANGE\_EVENT);

},

addChangeListener: function(callback) {

this.on(CHANGE\_EVENT, callback);

},

removeChangeListener: function(callback) {

this.removeListener(CHANGE\_EVENT, callback);

}

});

AppDispatcher.register(function(action) {

switch(action.type) {

case AppConstants.LOAD\_NOTES\_REQUEST: {

\_isLoading = true;

NotesStore.emitChange();

break;

}

case AppConstants.LOAD\_NOTES\_SUCCESS: {

\_isLoading = false;

\_notes = action.notes.map( formatNote );

\_loadingError = null;

NotesStore.emitChange();

break;

}

case AppConstants.LOAD\_NOTES\_FAIL: {

\_loadingError = action.error;

NotesStore.emitChange();

break;

}

default: {

console.log('No such handler');

}

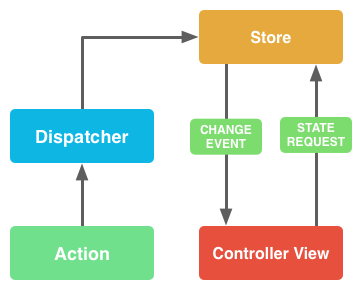
}

});

export default NotesStore;

Самое важное, что мы сделали — добавили к нашему хранилищу возможности EventEmitter из NodeJS. Это позволяет хранилищам слушать и рассылать события, что, в свою очередь, позволяет компонентам представления обновляться, отталкиваясь от этих событий. Так как наше представление слушает событие «change», создаваемое Хранилищами, оно узнаёт о том, что состояние приложения изменилось, и пора получить (и отобразить) актуальное состояние.

Также мы зарегистрировали обработчик в нашем AppDispatcher с помощью его метода register. Это означает, что теперь наше Хранилище теперь слушает оповещения от AppDispatcher. Исходя из полученных данных, оператор switch решает, можем ли мы обработать Действие. Если действие было обработано, создается событие «change», и Представления, подписавшиеся на это событие, реагируют на него обновлением своего состояния:

[](https://camo.githubusercontent.com/86b8718505c3020eff76d72ab762b39aaa735db6/68747470733a2f2f686162726173746f726167652e6f72672f66696c65732f6534632f6537612f6138382f65346365376161383831663934613534396133623663393837643465336531632e706e67)

Представление использует метод getNotes интерфейса Хранилища для того, чтобы получить все notes из внутреннего объекта \_notes и передать эти данные в компоненты. Это очень простой пример, однако такая архитектура позволяет компонентам оставаться достаточно аккуратными, даже если вместо Представлений использовать более сложную логику.

Тогда в компоненте нам нужно подписаться на изменения в хранилище и вызывать действия.

**client/components/App.jsx**

import React from 'react';

import NotesStore from '../stores/NotesStore';

import NotesActions from '../actions/NotesActions';

import NoteEditor from './NoteEditor.jsx';

import NotesGrid from './NotesGrid.jsx';

import './App.less';

function getStateFromFlux() {

return {

isLoading: NotesStore.isLoading(),

notes: NotesStore.getNotes()

};

}

const App = React.createClass({

getInitialState() {

return getStateFromFlux();

},

componentWillMount() {

NotesActions.loadNotes();

},

componentDidMount() {

NotesStore.addChangeListener(this.\_onChange);

},

componentWillUnmount() {

NotesStore.removeChangeListener(this.\_onChange);

},

handleNoteDelete(note) {

NotesActions.deleteNote(note.id);

},

handleNoteAdd(noteData) {

NotesActions.createNote(noteData);

},

render() {

return (

<div className='App'>

<h2 className='App\_\_header'>NotesApp</h2>

<NoteEditor onNoteAdd={this.handleNoteAdd} />

<NotesGrid notes={this.state.notes} onNoteDelete={this.handleNoteDelete} />

</div>

);

},

\_onChange() {

this.setState(getStateFromFlux());

}

});

export default App;

А уже этот компонент будет передавать данные всем дочерним компонентам через props.

**Запуск проекта**

1. Clone this repo
2. cd spa-webinar
3. npm install
4. Скопируйте etc/config.js.sample в etc/config.js
5. npm run server
6. npm run webpack-devserver
7. Откройте http://localhost:8090 в браузере

--------------------------------------------------------------------------------------------------------------------------------------------

**Мои заметки**

//Тема 1 – Основы

1)Работа с состояниями

2)Виртуальное DOM дерево - определяет минимальное кол-во манипуляций с DOM деревом дял внесения изменений и применяет его на реальном

3)

<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.js"></script>

либы реакта

4)

<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.6.15/browser.js"></script> - необходим для конвертирования непонятного для веб-страницы кода в нативный ЖС

5)

<script type="text/babel"> - type="text/babel" дает понять что все что находиться внутри скрипта должно конверитироваться бабилем

6)

var HelloWorld = React.createClass - создаем класс для реакта и присваиваем его переменной

7) Обязательный метод render отвечает за то, что будет отображаться на нашей странице

8) JSX - спициальный XML подобный синтаксис для ЖС, который позволяет писать наши компоненты в виде HTML тегов, но в итоге все написаное конвертируется в ЖС с помощью Бабеля (JSX не обязателен, но такой формат наиболее испольуемый)

9) Для отображения написанной функции на Реакте, испольозуется ReactDOM.render

ReactDOM.render(

<HelloWorld />, - название элемента, в нашем случае HelloWorld

document.getElementById("content") - место где отобразить

);

10) Множество обектов перед рендером нужно оборачивать в какой-то общий элемента

ReactDOM.render(

<div>

<HelloWorld />

<HelloWorld />

<HelloWorld />

<HelloWorld />

<HelloWorld />

</div>,

document.getElementById("content")

);

11)

<script type="text/babel">

var HelloWorld = React.createClass({

render: function(){

return <h1> My name is {this.props.name} and I love {this.props.action}! </h1> - использовать параметры name и action

}

});

ReactDOM.render(

<div>

<HelloWorld name="Kolya" action="eating"/> - присваивание параметрам значений

<HelloWorld name="Vanya" action="football"/>

<HelloWorld name="Sveta" action="walking"/>

<HelloWorld name="Jenya" action="swimming"/>

</div>,

document.getElementById('content')

);

</script>

12) В JSX при обьявлении класса нужно использовать ключевое слово className вместо className

<li className="contact">

13) Синтетичекие события в Реакте, это события поверх стандартных событий браузера, который работают одинаково во всех браузерах

14) Для обновления состояния компонента используется метод setState. Он изменяет свойства компонента и говорит Реакту сделать перерендер компонентов

this.setState({

displayedContacts: displayedContacts

});

//Тема 2 - Компоненты

1) Жизненный цикл компонента (Инициализация ===> Изменение ===> Удаление)

Его жизнь начинается с того, что он появляется в ДОМ дереве (происходит его инициализация), а заканчивается тем, что он будет удален из ДОМ дерева

В течении жизни с ним могут происходить некоторые изменения: изменение параметров или состояния

Инициализацией компонента можно считать первый рендер

При инициализации поочередно вызываются след методы:

1. getDefaultProps - отвечает за значения параметров по умолчанию (вызывается единожды)

2. getInitialState - отвечает за первоначальное состояние компонента

3. componentWillMount - вызывается перед рендером компонента. Подготовка к рендерингу

4. render

5. componentDidMount - если нужно использовать какой-то джеквери, обратиться к дом елементам это все делается в этом методе

2) Когда происходит изменения параметров компонента (т.е. тех параметров, которые приходят к нашему от родительского) порядок вызова методов такой

1. componentWillReceiveProps - в этом методе модифицируется состояние в зависимости от тех параметров, которые пришли. Если это делается в этом методе то это не вызовет дополнительного рендера

2. shouldComponentUpdate - смотрит нужно ли перерисовывать наш компонент или нет. В него приходят новые параметры и состояние. Используюется для повышения производительности и быстродействия

3.componentWillUpdate - сюда приходят новые параметры и состояние, которые можно предварительно обработать перед рендером если того требует ситуации. Используюется для отслеживания изменения состояния компонента

4. render

5. componentDidUpdate - сюда приходят преыдущие параметры и состояние

3) При изменении состояния

1. shouldComponentUpdate

2.componentWillUpdate

3. render

4. componentDidUpdate

4) Деонтаж компонента

//Тема - FLUX

FLUX - архитектура позволяющая упростить настройку взаимодействия всех данных внутри приложения. Удобна для управления всеми данными внутри приложения

FLUX архитектура :

VIEW - это все компоненты в нашем приложении. При каких-то действиях пользователя из компонента посылается какое-то событие в ACTION. Отсюда мы можем в самом ACTION обращаться к внешнему API с помощью каких-то утилит. К нам могут прийти какие-то данные из API в ACTION и затем через DISPATCHE мы отправляем все это в STORE, а каждый из компонентов может подписываться на изменения этого STORE. STORE это по сути хранилище данных внутри нашего приложения. При изменении состояния в STORE - это влияет на VIEW состояния в приложении.