

REACTJS*, JAVASCRIPT*

Готовим плацдарм для react-приложения

ИЗ ПЕСОЧНИЦЫ

movax10h 17 марта 2017 в 16:46 👁 30,4k



Я хочу рассказать о процессе создание платформы для react приложения, которая использует [mobx](#) в качестве **Model**-и. Пройти путь от пустой директории проекта до рабочего примера. Рассмотреть основные моменты, на которые я обращал внимание в процессе разработки. Постараюсь насытить текст уточняющими

ссылками, дополнительные заметки будут выделены курсивом с пометкой «*Note:*».

Рассказ будет состоять из двух частей:

1. Готовим плацдарм для react приложения
2. Mobx + react, взгляд со стороны

Буду писать «как я вижу», поэтому предложения и замечания по улучшению приветствуются. Надеюсь, читатель знает, что такое npm, node.js и react.js, имеет базовые знания о props и state. На момент написания статьи, у меня стоит windows и нестабильная node.js 7.3.0 версии.

Готовим плацдарм для react приложения

Существуют тысячи [react skeleton-ов](#) и [boilerplate-ов](#), та что уж тут говорить, даже «fb» выпустил [свой](#) с блекджеком и hotreload-ом. Мы же не будем использовать готовый, а соберем все своими руками и увидим, как это работает. Самостоятельно пройдем этот путь и заглянем во все темные углы, чтобы понять всю механику процесса, так же разобраться в тех деталях, которые были непонятны ранее. Я не претендую на очередной велосипед, скорее разработка ради просвещения. Переполняемые энтузиазмом, открываем консоль в любимой IDE, создаем новую директорию для проекта и переходим внутрь. Погнали!

```
npm init
```

Тут все просто, вам предложат несколько общих вопросов, после чего npm создаст для нас `package.json` файл управления зависимостями.

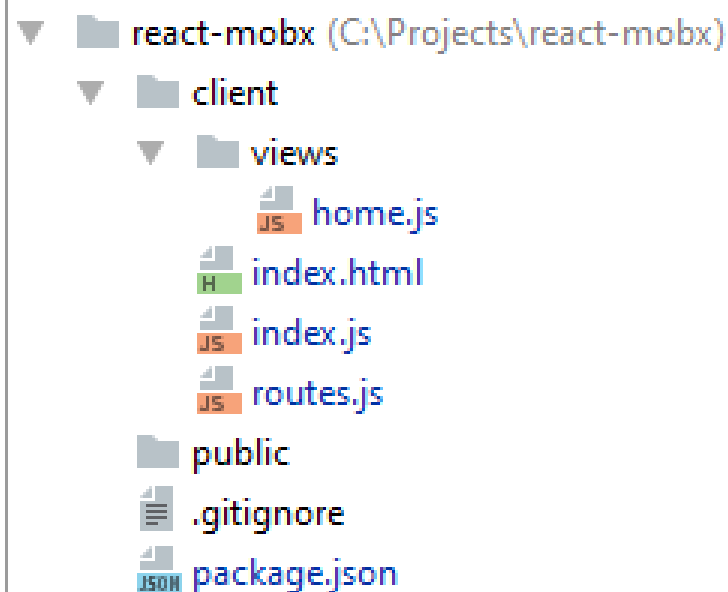
Note: Чтобы каждый раз не заполнять информацию о себе, можно прописать

```
npm set init.author.name "your name"  
npm set init.author.email "your email"  
npm set init.author.url "your site url"
```

Далее установим необходимые для работы react-а пакеты и сделаем о них запись в `package.json` в секции `dependencies`. Мы будем использовать `react-router`, поэтому сразу поставим и его:

```
npm i --save react react-dom react-router
```

Заложим структуру проекта и создадим пару директорий. В большинстве случаев требуется клиент-серверное приложение. Мне нравится иметь две отдельные директории для клиентской и серверной части, тут можно холиварить бесконечно долго, для примера я выбрал следующую структуру:



Для скелета нам понадобится:

- `index.js` – точка входа в клиентское приложение. Это первый файл в приложении, в который я смотрю, если вижу чужой проект в первый раз, это ниточка с которой начинаешь распутывать весь клубок;
- `routes.js` – настройка роутера. Для начала хватит одного роута, чтобы показать
- `home.js` – home page;
- `index.html` – мы будем делать SPA, `index.html` — это та единственная страничка;

[index.html](#)

Тут стоит обратить внимание на `div#app`, это контейнер для нашего будущего react приложения. Чуть позже мы добавим сюда скрипт.

[index.js](#)

Рендерим `<AppRouter />` в тот самый `div#app`.

[views/home.js es6](#)

Перед роутом давайте посмотрим на домашнюю (и пока единственную) вьюшку. Это react компонент, который просто выводит надпись приветствия.

Мы будем использовать ES6 way при создании react компонент. Как подружить react с ES6, можно почитать в [документе](#) или [тут](#) на русском. Рекомендую сразу пытаться писать на ES6, вы тут же почувствуете выгоду, тем более тема легкая для понимания.

Конечно же для удобства мы будем использовать jsx нотацию. Для того, чтобы браузер понял наш код, мы будем использовать [babel](#) транслятор, кроме того хочется идти в ногу со временем и использовать ES6/ES2015 [фичи](#), но не все браузеры поддерживают этот стандарт, поэтому я опять обращусь к babel за помощью. Получается, babel – транспайлер, переписывающий код, написанный в новых стандартах, в код стандарта es5, который понимаю почти все браузеры, а также может транслировать react jsx код, в код понятный браузеру. А еще, он поддерживает кучу плагинов. Это очень круто!

Note: Всю эту магию преобразований можно пощупать даже [онлайн](#).

Попробуйте вставить любой react или es6 код и увидите во что он трансформируется, например, код из `home.js`

Если вы проделали эту процедуру, то у вас могли заметить, что 9 строк react ES6 кода (~400 байт) превратились в 44! строки ES5 (~2200 байт)

```
1 import React from 'react';
2
3 export default class Home extends React.Component {
4   render() {
5     return (
6       <h1>Hello Kitty!</h1>
7     );
8   }
9 }

1 'use strict';
2
3 object.defineProperty(exports, "__esModule", {
4   value: true
5 });
6
7 var _createClass = function () { function defineProperties(target, props) { for (var i = 0; i < props.length; i++) { var descriptor = props[i]; descriptor.enumer
8
9 var _react = require('react');
10
11 var _react2 = _interopRequireDefault(_react);
12
13 function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { default: obj }; }
14
15 function _classCallCheck(instance, Constructor) { if (!(instance instanceof Constructor)) { throw new TypeError("Cannot call a class as a function"); } }
16
17 function _possibleConstructorReturn(self, call) { if (!self) { throw new ReferenceError("this hasn't been initialised - super() hasn't been called"); } return ca
18
19 function _inherits(subClass, superClass) { if (typeof superClass !== "function" && superClass !== null) { throw new TypeError("Super expression must either be n
20
21 var Home = function (_React$Component) {
22   _inherits(Home, _React$Component);
23
24   function Home() {
25     _classCallCheck(this, Home);
26
27     return _possibleConstructorReturn(this, (Home.__proto__ || Object.getPrototypeOf(Home)).apply(this, arguments));
28   }
29
30   _createClass(Home, [{
31     key: 'render',
32     value: function render() {
33       return _react2.default.createElement(
34         'h1',
35         null,
36         'Hello Kitty!'
37       );
38     }
39   }]);
40
41   return Home;
42 }(_React$Component);
43
44 exports.default = Home;
45 
```

Это расплата за синтаксический сахар, ведь, class-ов в javascript-е нет. Можно наблюдать как babel с легкой руки сделал из class-a функцию.

Наверно, на этом этапе нужно сказать пару строк о stateless компонентах. Грубо говоря, такими называются компоненты, у которых нет состояния. Наш Home компонент как раз не имеет состояния, поэтому мы можем переписать его как:

[stateless home.js](#)

Мы избавились от class-a, поэтому этот код будет гораздо короче в конечном ES5 синтаксисе, а его объем уменьшится более чем в 5 раз. Кроме того, мы можем сделать исходный код еще лаконичнее:

[stateless home.js](#)

Note: Мне нравится [эта](#) статья на тему stateless components, рекомендую.

[routes.js](#)

Наконец, в роутах пропишем только один path к Home компоненту. Тут вопросов не должно возникнуть, библиотека проста, но в тоже время обладает [мощным функционалом](#).

Подсунуть браузеру, читаемый код – только пол дела, так как проект состоит из множества файлов, а в конечном результате нам нужен только один минимизированный js файл(который мы подключим к index.html), то нам понадобится еще и сборщик модулей. Собирать мы будем с помощью [webpack](#).

Ставим его:

```
npm i --save-dev webpack
```

Note: Обратите внимание, что webpack ставится в devDependencies секцию.

Все, что связано с разработкой и не будет использоваться в производственной среде ставится с флагом --save-dev, зачастую это: сборщики и плагины к ним, тесты, линтеры, лоадеры, пост/препроцессоры и прочее.

Как было описано выше для всех преобразований кода нам нужен

babel и необходимые preset-ы(наборы плагинов) к нему:

```
npm i --save-dev babel-loader babel-core babel-preset-es2015 babel-preset-react
```

Webpack-у потребуется [файл конфигурации](#), создадим webpack.config.js в корне директории проекта.

[webpack.config.js](#)

Также babel рекомендует использовать .babelrc файл, тут опишем какие preset-ы мы хотим использовать.

```
{  
  "presets": ["es2015", "react"]  
}
```

Note: Пара полезных ссылок: [6 вещей](#), которые необходимо знать о babel 6 и какая разница [в порядке объявления preset-ов](#).

Тут мы говорим сборщику, что точка входа в наше приложение – client/index.js файл, webpack начнет свою работу с этого файла, нам не нужно указывать ему какие файлы должны войти в сборку, все это он сделает за нас сам. На выходе должен получиться один bundle.js файл в директории public. Грубо говоря, этим конфигом мы говорим babel-ю: «Эй, склей все необходимые файлы в один, начиная с index.js и позаботься о том, чтобы babel преобразовал все .js и .jsx файлы в код понятный браузерам.» Разве это не здорово? Настройка webpack-а готова, идем в консоль и запускаем

сборщик:

```
webpack
```

В директории `public` должен появиться файл `bundle.js`. `Public` — это наша публичная директория, после билда все «готовые» файлы(для нас это `index.html` + `bundle.js`) должны попасть сюда. Бандл готов, пришло время заняться `html`. Тут стоит понимать, что текущий `index.html` — это только заготовка, в дальнейшем нам, например, нужно прицепить `CSS` или `js` файлы, минимизировать или добавить какое-то содержимое, при этом для разных сборок производить разные операции. Для этих целей нам потребуется [HtmlWebpackPlugin](#). Ставим его:

```
npm i --save-dev html-webpack-plugin
```

После идем в файл конфигурации и настраиваем плагин:

[webpack.config.js](#)

Этим мы говорим `webpack`-у, чтобы он вставил `'script'` тег со ссылкой на сбилденый им `bundle.js` в нашу заготовку `index.html`. При этом «готовый» `index.html` будет лежать рядом с бандлом, т.е в `public`. Запустим `webpack` еще раз и убедимся в этом, проверив публичную директорию.

Давайте вернемся к нашему бандлу, который собрал нам `webpack`. Внимательный читатель заметит, что `~710KB` многовато

для 'Hello Kitty!'. Согласен, но у нас пока девелоп версия, которая предоставляет дополнительный функционал в помощь разработчику, например, показывает различные варнинги в консоли. Давайте попробуем намекнуть react-у, что мы хотим собрать проект под продукцию. Для этого нужно минимизировать конечный bundle.js и задать NODE_ENV переменной окружения значение «production». В конфиге добавляем плагины, при этом ничего дополнительного качать и устанавливать не нужно.

[webpack.config.js](#)

Ознакомиться с полным списком плагинов можно [тут](#).

Note: Если не задать NODE_ENV=production, а просто сжать файл, то react покажет предупреждение в консоли:

```
Warning: It looks like you're using a minified copy of the development build of React. When deploying React apps to production, make sure to use the production build which skips development warnings and is faster. See https://fb.me/react-minification for more details. bundle.js:1
```

Пересоберем проект с использованием плагинов и вновь посмотрим на наш свежий бандл.

Asset	Size	Chunks		Chunk Names
bundle.js	191 kB	0	[emitted]	main
index.html	184 bytes		[emitted]	

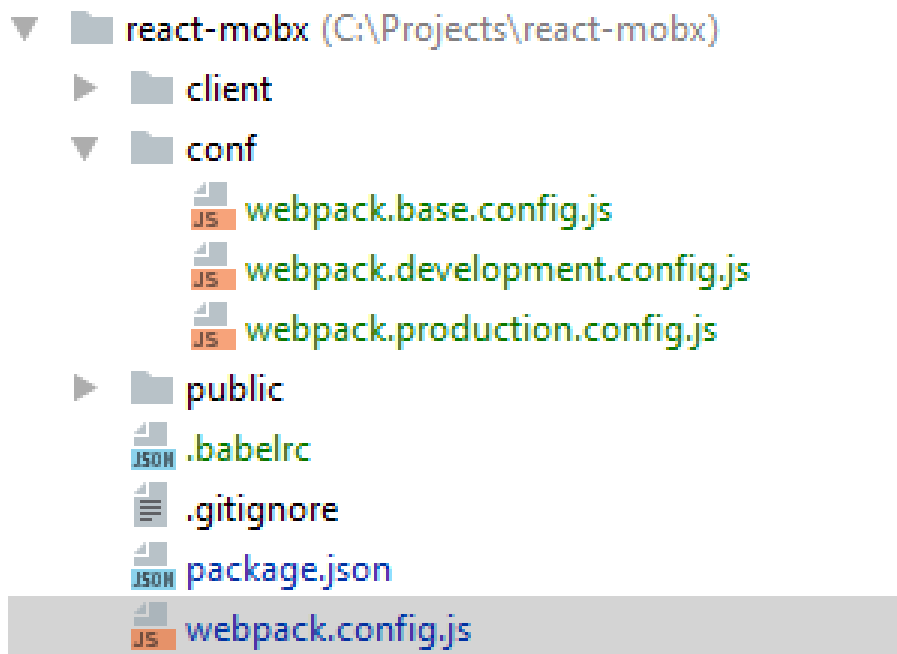
С этим уже можно работать, но это еще не предел, давайте посмотрим на еще одну настройку конфигурации webpack-а — "devtool". Эта опция также влияет на размер конечного файла и скорость сборки. Поэтому мы будем использовать разные

значения для продукции и девелопа. [Тут](#) можно почитать как работает каждая опция. Для себя я выбрал «source-map» для продакшена и «inline-source-map» для девелопа, хотя, наверно, для разных проектов эти значения могут варьироваться. Тут нужно поиграть и выбрать оптимальное для себя.

Настало время изменить файл конфигурации, ведь теперь мы с лёгкостью можем собирать наш проект под разные нужды, мне не нравится решение, когда в одном конфиге через условия регулируются настройки. При увеличении настроек и/или типов сборок, конфиг становится тяжело читаем, поэтому будем использовать [webpack-config](#).

```
npm install --save-dev webpack-config
```

Как видно из описания — это помощник для загрузки, расширения и мерджа конфигурационных файлов. В данном примере я бы хотел иметь возможность делать две сборки: development и production. Добавим директорию conf и три конфига, как показано на рисунке:



[webpack.base.config.js](#)

В базовом конфиге находятся общие настройки, которые справедливы для двух сборок.

[webpack.development.config.js](#)

В development конфиге мы указываем лишь имя конечного бандла — 'bundle.js'

[webpack.production.config.js](#)

Для production, мы добавляем плагин для минимизации, а так же меняем имя бандла. Как видно оба конфига расширяются от базового.

[webpack.config.js](#)

Теперь мы можем управлять сборками с помощью переменной среды `NODE_ENV`, в зависимости от её значения, `webpack-config` будет автоматически подтягивать нужный файл.

Note: `webpack.config.js` использует ES6 синтаксис, поэтому при попытке запустить `webpack`, вы увидите ошибку «`SyntaxError: Unexpected token import`». Для решения проблемы достаточно переименовать данный файл в `webpack.config.babel.js`. Этим мы пропускаем конфиг через `babel-loader`.

Добавим необходимые скрипты запуска `webpack`-а в `package.json` в секцию `scripts`:

```
"scripts": {  
  "build-dev": "set NODE_ENV=development&& webpack --progress",  
  "build-prod": "set NODE_ENV=production&& webpack --progress"  
},
```

С флагом `--progress` можно видеть прогресс выполнения и отчет по бандлам. Теперь мы можем собирать две разные сборки; для продукции:

```
npm run build-prod
```

и для разработки:

```
npm run build-dev
```


Note: Я работаю в windows, поэтому присвоение выглядит так «set NODE_ENV=production». Для других ОС присвоение выглядит по-другому.

Остался последний штрих — hot loader. Эта штука позволяет пересобирать проект на лету при изменении в исходных файлах. При этом страница не будет перезагружена и состояния не будут потеряны. Это в разы ускоряет разработку, а процесс девелопинга превращается в наслаждение. Подробнее можно послушать в [этом подкасте](#), так же там есть ссылки на интересные ресурсы по данной теме.

Для этого нам понадобятся: [react-hot-loader](#), [webpack-dev-middleware](#) и [webpack-hot-middleware](#) и, конечно же, сам сервер, будем использовать [express](#).

```
npm i --save express
```

```
npm i --save-dev react-hot-loader@next webpack-dev-middleware  
webpack-hot-middleware
```

Note: Обратите внимание, что необходимо установить react-hot-loader  next версии.

Добавим в корень проекта файл

[server.js](#)

Минимальный express сервер, единственный нюанс — это настройка `middleware` для development сборки. Как видно данные для `middleware` берутся из `webpack.config.babel`

Следующим шагом добавляем в `.babelrc` секцию `plugins`

```
"plugins": [  
  "react-hot-loader/babel"  
]
```

Конфигурационный файл для development теперь выглядит так:

[webpack.development.config.js](#)

Также изменения претерпел

[index.js](#)

И, наконец, скрипты в `package.json` должны выглядеть так

```
"scripts": {  
  "build-dev": "set NODE_ENV=development&& node server.js",  
  "build-prod": "set NODE_ENV=production&& webpack && node  
server.js"  
},
```

Note: если вы попытаете запустить скрипт, то опять увидите ошибку «SyntaxError: Unexpected token import». Потому что, `server.js` использует ES6 `import`-ы и пытается прочитать `webpack.config.babel.js`, в котором тоже используются `import`-ы. А

поддержку обещают только в 8й версии. Потребуется *babel* для командной строки *babel-cli*:

```
npm i --save-dev babel-cli
```

Будем использовать *babel-node*, вместо *node*, всё должно работать:

```
"scripts": {  
  "build-dev": "set NODE_ENV=development&& babel-node server.js",  
  "build-prod": "set NODE_ENV=production&& webpack && babel-node  
server.js"  
},
```

Пробуем собрать обе сборки, при этом для *production*, соберется минимизированный *bundle.min.js* и запустится сервер на 7700 порте, а для *development* будет работать горячая перезагрузка, при этом вы не увидите никаких файлов в *public* директории, весь процесс будет проходить *in memory*. Для теста усложним код *home.js*

[home.js](#)

Кстати, если вы запустили *development* сборку, то все изменения должны подтянуться сразу. Давайте кликнем по заголовку, тем самым изменим состояние *name* с «Kitty» на «Bunny», далее в коде заменим текст в заголовке с «Hello» на «Bye». Перейдем в браузер и увидим надпись «Bye Bunny», т.е. горячая перезагрузка сработала, при этом измененное состояние не сбросилось.

Сначала не хотел добавлять работу с CSS, но в процессе написания статьи понял, что все таки для полного комплекта нужно добавить процесс сборки стилей.

Наверно, у каждого был случай, когда правишь верстку в одном месте и незаметно для себя создаешь новую проблему в другом, стили перезатирают друг друга или используется одинаковые классы, которые были описаны выше. Мы будем писать react компоненты, так почему бы нам сразу и не использовать CSS для компонентов, а не глобально? Будем использовать [CSS модули](#)! Нам понадобится [post-css](#) и его [плагины](#). Для начала нам будут интересны [autoprefixer](#) и [precss](#) для ускорения разработки, устанавливаем:

```
npm i --save-dev css-loader style-loader postcss-loader autoprefixer precss
```

Делаем изменения в конфигах

[webpack.base.config.js](#)

[webpack.development.config.js](#)

[webpack.production.config.js](#)

В базовый конфиг добавляем плагин, для остальных добавляем loaders, отличаются только настройки. Тут будет интересна `localIdentName` опция, она позволяет задавать имена CSS классам, для production версии будем использовать хэш из 10

символов, для девелоп — названия классов + хэш из 5 символов. Это очень удобно, т.к при дебаге ты всегда знаешь какой класс тебе нужно поправить. Для примера давайте добавим Menu компонент:

Структура проекта

menu/index.js

Обратите внимание, как используется css модуль. Это локальные стили, т.е для другого меню, мы так же можем использовать класс .menu с другими стилями, и они не пересекутся.

menu/style.css

app.js

Но мы так же можем использовать и «глобальные» стили, например, для html и body. Достаточно подключить их app.js.

routes.js

Добавим немного вложенности, теперь у нас есть App контейнер со вложенной Home страницей.

Настройки webpack-а также позволят использовать горячую перезагрузку для стилей, просто попробуйте сделать некоторые изменения в стилях.

На этом первая часть закончилась. Я не претендую на самый-самый правильный вариант, но я прошёл этот путь и почерпнуть несколько интересных вещей, пока писал эту статью, надеюсь она окажется чем-то полезным и другим.

Ссылка на то, что получилось: github.com/AlexeyRyashencev/react-hot-mobx-es6

Проголосовать:



+19



Поделиться:



Сохранить:



Комментарии (18)

Похожие публикации

Честный realtime на React и Redux, как основа автоаукциона

jumanji2277 • 15 августа 2017 в 16:49

30

Введение в JavaScript итераторы на ES6

ПЕРЕВОД

2

React на ES6+

ПЕРЕВОД

olegshilov • 9 июля 2015 в 16:30

15

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета)

BarakAdama • вчера в 10:52

69

Почему следует игнорировать истории основателей успешных стартапов

ПЕРЕВОД

m1rko • вчера в 10:44

20

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

24

Java и Project Reactor

zealot_and_frenzy • вчера в 10:56

10

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle

erogov • вчера в 12:46

6

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке

NAGru • вчера в 10:10

31

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

alizar • вчера в 15:32

49

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

Pochtoycom • вчера в 13:06

85

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

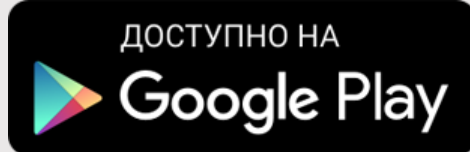
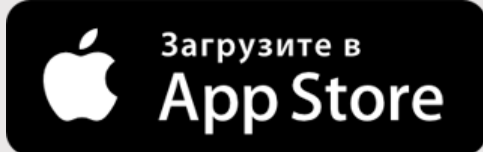
marks • вчера в 14:19

139

Дела шпионские (часть 1)

TashaFridrih • вчера в 13:16

16



Полная версия

2006 – 2018 © TM