**WEBPACK**

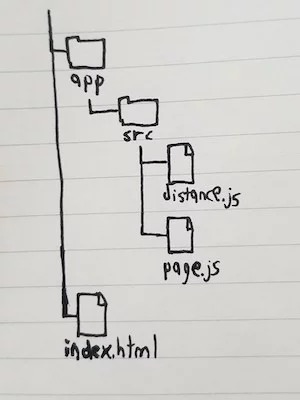
scr origin: <https://itnext.io/webpack-from-0-to-automated-testing-4634844d5c3c>

src translate: <https://tproger.ru/translations/webpack-from-zero-to-hero/>

Если не установлен yarn, то установщики здесь: <https://yarnpkg.com/ru/docs/install#windows-stable>

git init + создание удаленного репозитория (Это по желанию).

Создаем каталоги, общая структура каталогов такова:



Для работы с Webpack необходимо его установить с помощью npm или yarn:

yarn add --dev webpack webpack-cli

Теперь у нас подключён Webpack и доступна его командная строка.

Добавляем JS:

Основная структура приложения будет выглядеть так:

* У вас есть файл app/src/distance.js, экспортирующий функцию, которая запускает формулу вычисления расстояния (на самом деле нужно использовать [ортодому](https://ru.wikipedia.org/wiki/%D0%9E%D1%80%D1%82%D0%BE%D0%B4%D1%80%D0%BE%D0%BC%D0%B8%D1%8F)) и функцию, которая сообщает, какая точка из массива точек ближе всего находится.

// Функция distance() принимает 2 значения, представленных

// числами x и y, и возвращает расстояние между ними

//

// [TODO] Используйте ортодому

function distance(p2, p1) {

let yDist = p2.y - p1.y;

let xDist = p2.x - p1.x;

return Math.sqrt(Math.pow(yDist, 2) + Math.pow(xDist, 2));

}

// sortByDistance принимает ваше местоположение и массив точек

// и возвращает отсортированный массив точек

function sortByDistance(myPt, points) {

return points.sort(

(pt1, pt2) => distance(pt1, myPt) - distance(pt2, myPt));

}

module.exports = {distance, sortByDistance};

* Также у вас есть файл app/src/page.js, который использует код из distance.js, чтобы вывести ближайший магазин из списка, а затем отобразить его на странице.

import {sortByDistance} from './distance';

let stores = [

{name: "Cambridge Naturals", x: -71.1189, y: 42.3895},

{name: "Sarah's Market", x: -71.1311, y: 42.3823},

{name: "Whole Foods Fresh Pond", x: -71.1420, y: 42.3904},

];

let here = {name: "You are here", x: -71.1470, y: 42.3834};

let nearest = sortByDistance(here, stores)[0];

document.getElementById("nearest-store").innerHTML = nearest.name;

* И, наконец, у вас есть страница index.html.

<!DOCTYPE html>

<html>

<head>

<title>Ближайший магазин чая из гибискуса</title>

</head>

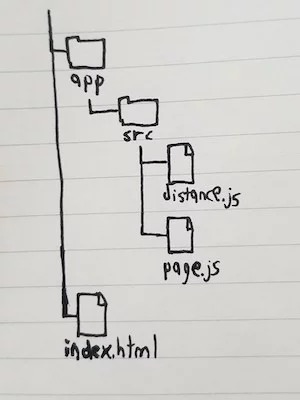
<body>

<p>Nearest store is <span id="nearest-store"></span></p>

<script src="dist/main.js"></script>

</body>

</html>



Отлично, теперь все зависимости JavaScript связаны. Используем Webpack для создания приложения. Выполним следующую команду:

**npx webpack app/src/page.js**

Это работает так: с помощью команды $ npx webpack app/src/page.js вы указываете, что отправной точкой (в терминологии Webpack — точкой входа в ваш код JavaScript) является page.js. Webpack читает файл page.js и видит в нём строку import {sortByDistance} from ./distance. Теперь он знает, что distance.js является зависимостью к page.js. И из всех зависимостей в вашем коде Webpack строит граф и использует его для построения пакетного JavaScript-файла dist/main.js

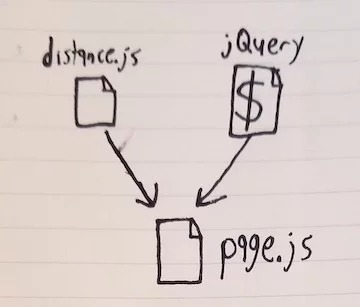
Установка jQuery:

**yarn add --dev jquery**

в page.js импортируем jquery:

**import $ from "jQuery";**

Теперь граф зависимостей выглядит так:



Для того чтобы нам постоянно не писать длинные команды по типу:

npx webpack app/src/page.js и т.д. в файл package.json необходимо добавить вот такие строки:

"scripts": {

"build": "webpack app/src/page.js"

}

Теперь, чтобы запустить сборку пакета, можно просто выполнить

**yarn build**

вместо $ npx webpack app/src/page.js.

**Настройка Webpack с помощью файла webpack.config.js**

Задать расположение точек входа и выхода программы можно, настроив файл конфигурации. Поместите следующий код в файл с именем webpack.config.js:

module.exports = {

entry: \_\_dirname + "/app/src/page.js",

output: {

path: \_\_dirname + "/dist/",

}

}

Теперь можно выполнить

**npx webpack**

или сделать ту же сборку (yarn build), что и раньше, без указания точки входа в программу в аргументах командной строки, т. к. теперь всё это есть в webpack.config.js. Также это значит, что нужно обновить скрипт файла package.json следующим образом:

"build": "webpack"

Файл конфигурации предназначен не только для настройки расположения входных и выходных файлов. Также можно настроить что именно Webpack делает, когда встречает файлы разных типов, используя специальные загрузчики, которые на самом деле являются JavaScript-программами, преобразующими ваш код.

**Загрузчики**

**Babel**

**yarn add --dev babel-core babel-loader@7.1.5 babel-preset-env**

Затем в файле webpack.config.js добавим следующий код в module.exports:

module: {

        rules: [

            {

                test: /\.js$/,

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

                use: [

                    {

                        loader: 'babel-loader',

                        options: {

                            presets: ['env']

                        }

                    }

                ]

            }

        ]

    }

**Добавление тестовых сценариев в сборку**

Добавим несколько тестовых сценариев в файл distance.js. Для этого будем использовать Mocha и Chai. Выполните следующую команду:

**yarn add --dev mocha chai**

Затем создайте новый каталог app/test и новый файл app/test/distance.test.js, содержащий следующий фрагмент:

describe("distance", function() {

it("calculates distance with the good ol' Pythagorean Theorem", function() {

let origin = {x: 0.0, y: 0.0};

let point = {x: 3.0, y: 4.0};

expect(distance(point, origin)).to.equal(5);

});

});

describe("sortByDistance", function() {

it("sortsByDistance", function() {

let places = [

{name: "Far away", x: 100, y: 50},

{name: "Nearby", x: 20, y: 10},

];

let origin = {name: "Origin", x: 0, y: 0};

let sorted = sortByDistance(origin, places);

expect(sorted[0].name).to.equal("Nearby");

expect(sorted[1].name).to.equal("Far away");

});

});

Так как слово import, Node в данный момент не поддерживает, поэтому чтобы distance.test.js увидел функции distance и sortByDistance их необходимо подключать при помощи require():

const chai = require('chai'); // require chai

const expect = chai.expect;

const distancefunc = require('../src/distance');

const distance = distancefunc.distance;

const sortByDistance = distancefunc.sortByDistance;

В свою очередь distance и sortByDistance должны быть экспортированны из своего файла.

module.exports = {distance, sortByDistance};

Далее допишем webpack.config.js:

let glob = require("glob");

let entry = \_\_dirname + "/app/src/page.js";

let outputPath = \_\_dirname + "/dist/";

if (process.env.TESTBUILD) {

 entry = glob.sync(\_\_dirname + "/app/test/\*\*/\*.test.js");

 outputPath = \_\_dirname + "/test-dist/";

}

module.exports = {

    entry: entry,

    output: {

        path: outputPath

    },

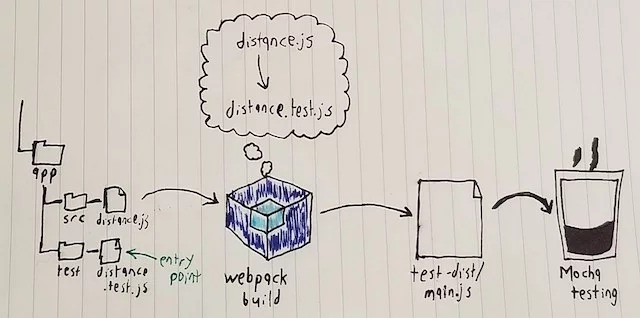
. . . . . .  
. . . . . .

. . . . . .

Список тестовых инструкций:

1. Webpack строит граф зависимостей, начинающийся с тестовых файлов, а не с файлов приложения.
2. Webpack создаёт файл JavaScript, содержащий весь тестовый код и его зависимости без ключевого слова import.
3. Выполняются тесты, запуская Mocha для этого JavaScript-файла.

Всё это будет выглядеть следующим образом:



Если выполнить **TESTBUILD=true npx webpack**, то можно увидеть каталог test-dist.

А если запустить **npx mocha test-dist/main.js**, то можно увидеть, как выполняются наши тесты.

Объединим эти вызовы в один, для этого создадим alias в package.json чтобы не писать длинные команды в консоль:

"test": "TESTBUILD=true npx webpack && mocha test-dist/main.js && npx rm -rf test-dist"

на **WIN** нужен бубен:  
необходимо установить process-env.  
yarn add -D cross-env  
Затем в разделе scripts немного изменил вызов команд:  
"test": "cross-env TESTBUILD=true npx webpack && mocha test-dist/main.js && npx rm -rf test-dist"  
источник: <https://medium.com/@hatzlgerhard/great-article-super-explained-863612b630fb>

теперь мы можно просто набрать:

**yarn test**

**Маппинг исходных файлов тестового кода**

Чтобы Mocha мог использовать source map при запуске тестов, необходимо установить ещё один пакет:

**yarn add --dev source-map-support**

Затем обновим webpack.config.js:

let glob = require("glob");

let entry = \_\_dirname + "/app/src/page.js";

let outputPath = \_\_dirname + "/dist/";

let devtool = "";

if (process.env.TESTBUILD) {

    entry = glob.sync(\_\_dirname + "/app/test/\*\*/\*.test.js");

    outputPath = \_\_dirname + "/test-dist/";

    devtool = "source-map";

}

Затем в объект module.exports добавим строку:   
devtool: devtool

Затем нужно обновить скрипт Mocha в разделе scripts.test файла package.json:

test": "cross-env TESTBUILD=true npx webpack && mocha test-dist/main.js --require source-map-support/register && rmdir /s/q test-dist"

Теперь, если вы выполните $ yarn test и получите ошибку, вы увидите, в какой строке она находится.

**Webpack 4: практические рекомендации по настройке**

src translate: <https://tproger.ru/translations/configure-webpack4/>

src origin:

 WARNING in configuration

The ‘mode’ option has not been set, webpack will fallback to ‘production’ for this value. Set ‘mode’ option to ‘development’ or ‘production’ to enable defaults for each environment.

You can also set it to ‘none’ to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/

Чтобы убрать это предупреждение, необходимо отредактировать скрипт, чтобы в нем появился флаг режима (mode flag):

"dev": "webpack --mode development",

В Webpack есть два режима: production и development.

добавим еще один флаг в конфиг:

"build": "webpack --mode production",

В данном примере я буду использовать build-скрипт, поскольку он уже “из коробки” обеспечивает множеством инструментов оптимизации и с этого момента можно свободно их использовать. **Основное различие между скриптами** build **и** dev **в том**, как они выводят файлы. Build создан для продакшен-кода, dev — для разработки: *он поддерживает «горячую» замену модулей, webpack-dev-server* и многие другие вещи, которые помогают разработчикам.

Вы легко можете переопределять значения по умолчанию в скриптах yarn, просто используя флаги:

  "scripts": {

    "dev": "webpack --mode development ./src/index.js --output ./dist/main.js",

    "build": "webpack --mode production ./src/index.js --output ./dist/main.js"

 },

Это перепишет опции по умолчанию без каких-либо других настроек.

*Флаг наблюдения* **--watch** для включения режима просмотра. Он будет следить за изменениями вашего файла и компилировать всякий раз при его изменении:

один из вариантов

"scripts": {

    "dev": "webpack --mode development ./src/index.js --output ./dist/main.js **--watch**",

    "build": "webpack --mode production ./src/index.js --output ./dist/main.js **--watch**"

 },

флаг входа. Работает так же, как и выход, но перезаписывает точку входа.

**Импорт HTML и CSS**

Создаем style.css в нашей папке ./src

Затем ставим:

**yarn add -D extract-text-webpack-plugin style-loader css-loader**

**ExtractTextPlugin** необходимо использовать для компиляции нашего CSS.

Далее включаем в точку входа (т.е. в page.js) наши стили: **import "./style.css";**

Затем в webpack.config.js создайте новое правило для файлов css:

            {

                test: /\.css$/,

                use: ExtractTextPlugin.extract(

                    {

                        fallback: 'style-loader',

                        use: ['css-loader']

                    }

                )

            }

        ]

    },

    plugins: [

        new ExtractTextPlugin( { filename: 'style.css' } )

    ]

};

Нам необходимо использовать ExtractTextPlugin, потому что Webpack по умолчанию понимает только формат .js. ExtractTextPlugin получает ваш .css и извлекает его в отдельный css-файл в вашем каталоге ./dist.

Если же у вас модуль ExtractTextPlugin не работает, вы можете переключиться на **MiniCssExtractPlugin**.

У Webpack, начиная с 4 версии, есть проблемы с этим плагином, поэтому вы можете столкнуться с определенной ошибкой.

Чтобы ее исправить, запустите:

**yarn add -D extract-text-webpack-plugin@next**

Теперь если запуcтить **yarn dev** , style.css будет выводиться в папку ./dist

**Поддержка SCSS**

Очень часто разрабатываются веб-сайты с SASS и PostCSS. Они весьма полезны.

Поэтому мы сперва включим поддержку SASS.

**yarn add -D node-sass sass-loader**

Затем вместо ./src/style.css создаем .src\scss\main.scss

После этого дописываем загрузчик sass-loader в массив правил rules из webpack.config.js и меняем шаблон в test:

{

                test: **/\.scss$/**,

                use: ExtractTextPlugin.extract(

                    {

                        fallback: 'style-loader',

                        use: ['css-loader', **'sass-loader'**]

                    }

                )

            }

**Шаблон HTML**

Ставим html-плагин:

**yarn add -D html-webpack-plugin**

Добавляем html-webpack-plugin в webpack.config.js

const HtmlWebpackPlugin = require('html-webpack-plugin');

. . . .

. . . .

plugins: [

        . . . . .

        new HtmlWebpackPlugin({

            inject: false,

            hash: true,

            template: './app/src/index.html',

            filename: 'index.html'

        })

    ]

Создадим html-шаблон файла index.html в ./src

**yarn dev**

**Кэширование и хеширование**

Хеширование необходимо, чтобы браузер «научился» запрашивать только измененные файлы. Одним из самых популярных способов решения проблем кеширования является добавление хеш-номера в активные (находящиеся в разработке) файлы (asset), такие как style.css и script.js.

Хеширование необходимо, чтобы браузер «научился» запрашивать только измененные файлы.

Webpack 4 имеет встроенные функции, реализованные через

chunkhash: <https://webpack.js.org/guides/caching/>

далее коррект. webpack.config.js:

const path = require('path');

let outputPath = path.resolve(\_\_dirname, 'dist');

module.exports = {

    entry: {

        main: entry

    },

    output: {

        path: outputPath,

        filename: '[name].[chunkhash].js'

    },

В файл ./app/src/index.html добавьте:

<!DOCTYPE html>

<html>

 <head>

   <title>Ближайший магазин чая из гибискуса</title>

   <link rel="stylesheet" **href="<%=htmlWebpackPlugin.files.chunks.main.css %>"**>

 </head>

 <body>

  <div>

      <p>Nearest store is <span id="nearest-store"></span></p>

      <div class="component">

        <h2 class="component\_\_title">Component title</h2>

        <div class="component\_\_text">component text</div>

      </div>

  </div>

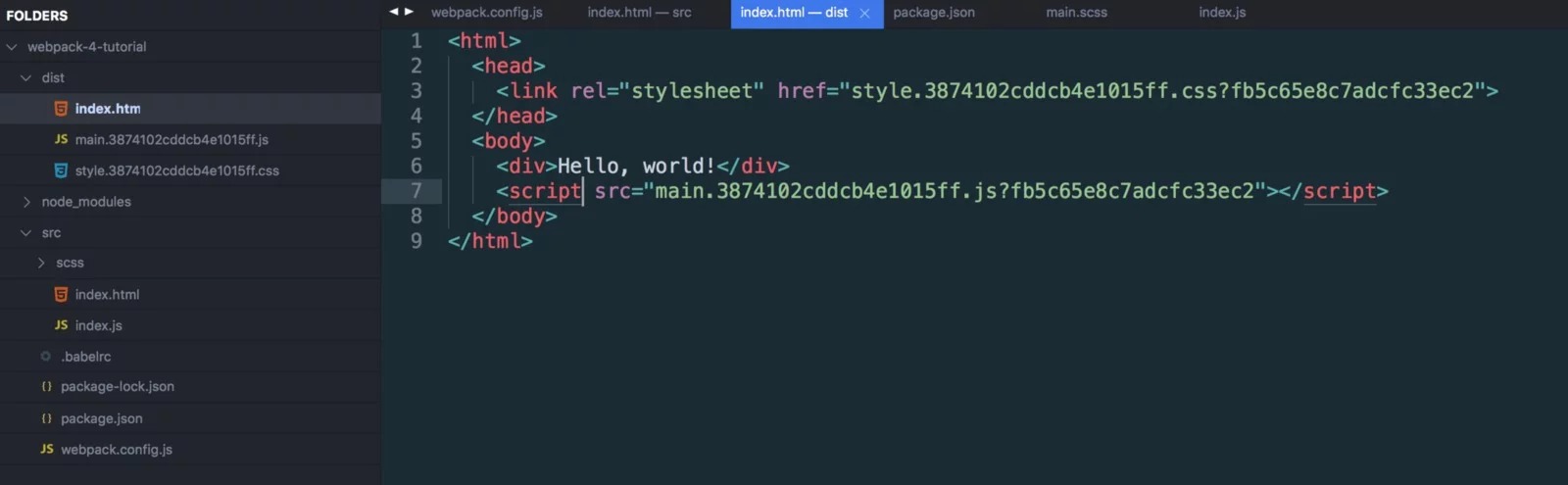
   <script src="**<%= htmlWebpackPlugin.files.chunks.main.entry %>"**></script>

 </body>

</html>

С помощью такого синтаксиса ваш шаблон «научится» использовать хешированные файлы.

Теперь в нашей папке ./dist есть файл index.html



Далее, если мы ничего не изменим в файлах .js и .css и запустим:

**yarn dev**

То увидим, что, независимо от количества запусков, числа в хешах должны быть идентичны в обоих файлах.

**Проблемы с хешированием и их решения**

Хотя у нас есть рабочий алгоритм, он пока не идеален. Что, если мы добавим новые строки кода page.js-файл вот так:

console.log('test');

Если вы снова запустите dev-скрипт, увидите, что в обоих файлах обновлен хеш и в main.css тоже, хотя в нем мы ничего не меняли.

А если поменять стили app\src\scss\main.scss, то новый хэш не будет сгенерирован в dist/

Эта известная проблема, и этот вопрос даже разобран на [StackOveflow](https://stackoverflow.com/questions/44491064/updating-chunkhash-in-both-css-and-js-file-in-webpack).

Как это исправить?

Как минимум есть 2 решения.

*решение 1:*

Ставим

**yarn add -D webpack-md5-hash**

Затем переписываем webpack.config.js

const WebpackMd5Hash = require('webpack-md5-hash');

plugins: [

**new ExtractTextPlugin( {**

**filename: 'main.[hash].css',**

**disable: false,**

**allChunks: true**

**} ),**

        new HtmlWebpackPlugin( {

            inject: false,

            hash: true,

            template: './app/src/index.html',

            filename: 'index.html'

        } ),

**new WebpackMd5Hash()**

    ]

};

Далее протестируем js-файлы. Теперь оба файла меняют хеш.

*решение 2:*

mini-css-extract-plugin более 5 млн. скачиваний

Все еще могут остаться некоторые конфликты, поэтому теперь попробуем подключить плагин mini-css-extract-plugin: <https://www.npmjs.com/package/mini-css-extract-plugin>.

**yarn add -D mini-css-extract-plugin**

Плагин Mini CSS предназначен для замены плагина extract-text и в дальнейшем для обеспечения лучшей совместимости.

Затем реструктурируем свой webpack.config.js файл, чтобы скомпилировать style.css с mini-css-extract-plugin,

удаляем 'extract-text-webpack-plugin'

const ExtractTextPlugin = require('extract-text-webpack-plugin');

и вместо него подключаем 'mini-css-extract-plugin'

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

меняем правило для обработки scss из ./src

{

                test: /\.scss$/,

                use: ['style-loader', MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader']

            }

удаляем старый плагин ExtractTextPlugin и подключаем новый MiniCssExtractPlugin:

 plugins: [

        // new ExtractTextPlugin( {

        //     filename: 'main.[hash].css',

        //     disable: false,

        //     allChunks: true

        // } ),

**new MiniCssExtractPlugin( {**

**filename: 'main.[contenthash].css'**

**} ),**

        new HtmlWebpackPlugin( {

            inject: false,

            hash: true,

            template: './app/src/index.html',

            filename: 'index.html'

        } ),

        new WebpackMd5Hash()

    ]

Теперь, когда мы редактируем main.scss из ./src, генерируется новый хеш для main.css в /dist. Также, когда мы редактируем scss, меняются только хеши css, и, когда я редактирую ./src/script.js, меняются только хеши script.js!

**Интегрирование PostCSS**

Чтобы выходной css-файл был безупречным, мы можем добавить наверх PostCSS: <https://www.npmjs.com/package/postcss>

PostCSS предоставляет вам autoprefixer, cssnano и другие приятные и удобные штуковины. Установим postcss-loader и autoprefixer

**yarn add -D postcss-loader autoprefixer**

Затем необходимо создать отдельный конфиг postcss.config.js юв корне?

module.exports = {

    plugins: [

        require('autoprefixer')

    ]

};

Далее в webpack.config.js, в обработчик scss добавляем, 'postcss-loader

{

                test: /\.scss$/,

                use: ['style-loader', MiniCssExtractPlugin.loader, 'css-loader', **'postcss-loader'**, 'sass-loader']

            }

**Пожалуйста, обратите внимание на порядок плагинов, которые мы используем для .scss**

use: [ 'style-loader', MiniCssExtractPlugin.loader, 'css-loader', 'postcss-loader', 'sass-loader']

Загрузчик инициализирует плагины, начиная с самого последнего.

Автор советует  [PostCSS.parts](https://www.postcss.parts/)  для изучения плагинов, доступных для PostCSS, например:

* [utilities](https://github.com/ismamz/postcss-utilities);
* [cssnano](https://github.com/ben-eb/cssnano);
* [style-lint](https://github.com/stylelint/stylelint).
* [css-mqpacker](https://github.com/hail2u/node-css-mqpacker)

Подключим некоторые из них, а также еще подключим cleancss (для удаления комментариев из скомпилированного css):

....................................

**Очищаем ./dist**

**yarn add -D clean-webpack-plugin**

Добавляем в webpack.config.js

const { CleanWebpackPlugin } = require('clean-webpack-plugin');

plugins: [

        new CleanWebpackPlugin(),

. . . .

все.