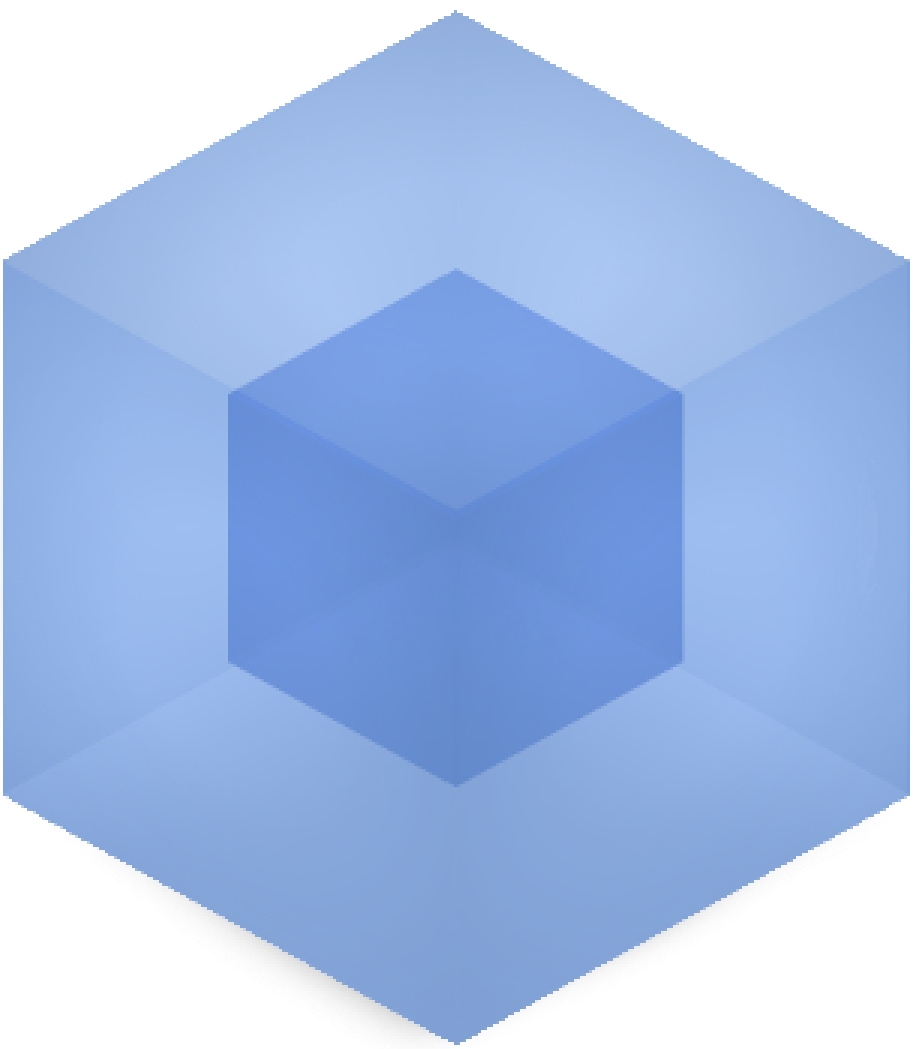
РАЗРАБОТКА ВЕБ-САЙТОВ*, NODE.JS*, JAVASCRIPT*, HTML*

Webpack для Single Page App

mrsum 31 августа 2015 в 14:29 59,1k

Привет!



Отгремели фанфары, прошел звон в ушах от истязаний «евангелистов», модников в сфере фронтенд разработки. Кто-то ушел на sprockets, кто-то пустился во все тяжкие и стал писать свои велосипеды или расширять функционал gulp или grunt. Но статей по поводу популярных инструментов автоматизации процесса сборки – стало существенно меньше и это факт! Пора бы заполнить освободившееся пространство чем-то существенно иным.

Уверен многие слышали о webpack. Кто-то подумал «он слишком много на себя берет» и не стал дочитывать даже вводную статью на оффициальном сайте проекта. Некоторые решили попробовать, но столкнувшись с небольшим количеством примеров настройки — отверг инструмент решив подождать пару лет. Но в целом, разговоров «вокруг» ходит масса. В этой статье — хочу развенчать некоторые сомнения. Может быть, кто-то заинтересуется и перейдет на «светлую сторону». Вообщем желающие под кат.

Честно сказать, активно работать с этим мощным инструментом, я начал в начале этого года. Сначала конечно же “wow” эффект. Который достаточно быстро сменился болью от жутко неудобной документации. Но пересилив этот этап – честно забыл о многих типичных проблемах, был поражен скоростью и удобством. Не буду утомлять, разберемся с…

Механика

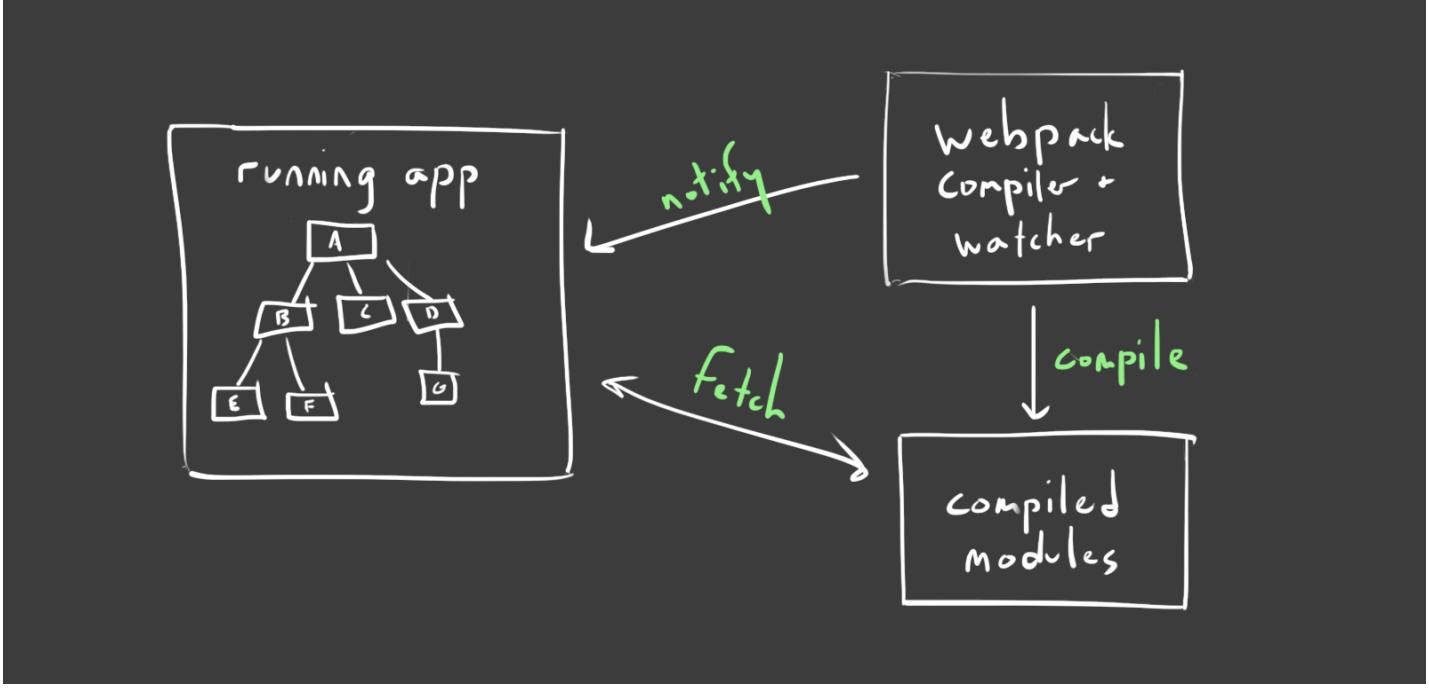
Логика работы очень простая, но эффективная
Webpack принимает один или несколько файлов, так называемых (entry points) используя пути из конфигурационного файла, далее загружает его.

Сборщик – во время обработки, встречая знакомую ему функцию **require()** – оборачивает содержимое вызываемого файла в функцию. Которая возвращает его контекст.
При этом не нужно заботиться о «гонке загрузки». Все что вы потребуете, будет доставлено в нужной последовательности!

Важно отметить, что во время разработки, когда запущен **webpack-dev-server** – промежуточные файлы **(chunks)** попадают в оперативную память RAM. Браузер же, получает их по протоколу **webpack://** прямо из контейнера.
Так же **dev-server** – поднимает простейший веб сервер на порт 8080, к которому можно достучаться по адресу ***localhost:8080***

Этот подход управления содержимым, как Вы понимаете ускоряет время промежуточной сборки, на значимое количество секунд. Что в рамках рабочего дня – экономит уйму Вашего времени.

Что касается окончательной сборки, то webpack используя тот же конфиг выполняет лишь этап, когда файлы раскладываются по папкам в файловой системе, готовые работать на Ваше благо.



source: jlongster.com/Backend-Apps-with-Webpack--Part-III

Подготовка

Тут все достаточно тривиально. Все что нам нужно это node.js и npm. Далее просто следуйте простым командам:

```
$ mkdir app
$ cd $_
$ npm init
& npm i webpack webpack-dev-server --save-dev # важно поставить
именно в dev dependencies
```

Как бы это не звучало, но больше половины Вы уже сделали.
Давайте перейдем к конфигурации проекта.
Для этого нужно определить что именно Вы хотите видеть и какие инструменты использовать. Условимся на этом:

- Common JS
- Stylus

- Jade
- Autoprefixer

Остальное поставим по необходимости.

Настройка

По умолчанию, webpack принимает конфигурационный файл выполненный в формате JSON. Находиться он должен в директории проекта и называться **webpack.config.js**

Для более удобной работы с запуском задач, а их будет две:
- **Сборка готового проекта**
- **Режим разработки**

Воспользуемся ***script*** секцией package.json файла, добавив следующие строчки:

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

После этого, находясь в директории проекта, Вам будут доступны следующие команды:

```
$ npm run dev # режим разработки http://localhost:8080
$ npm run build # сборка готовых ассетов и файлов в ./dist
```

Создадим файлы конфигурации терминальной магией:

```
$ touch webpack.config.js # корневой файл конфигурации
обрабатываемый webpack
$ mkdir config # директория с конфигурациями
$ cd $_ # перейдем
$ touch global.js # общие настройки инструментов, загрузчиков и
плагинов
$ mkdir env && cd $_
$ touch development.js # файл с 'тонкой' настройкой webpack для
режима разработки
$ touch production.js # аналогичный файл для конечной сборки
```

Перейдем к настройке окружения, для этого откроем файл **./webpack.config.js** и заполним следующим содержимым:

```
'use strict';

var _ = require('lodash');
var _configs = {
  global: require(__dirname + '/config/global'),
  production: require(__dirname + '/config/env/production'),
  development: require(__dirname + '/config/env/development')
};

var _load = function(environment) {
  // Проверяем окружение
  if (!environment) throw 'Can\'t find local environment variable
via process.env.NODE_ENV';
  if (!_configs[environment]) throw 'Can\'t find environments see
_config object';

  // load config file by environment
  return _configs && _.merge(
    _configs[environment](__dirname),
    _configs['global'](__dirname)
  );
```

```
};

/**
 * Export WebPack config
 * @type {[type]}
 */
module.exports = _load(process.env.NODE_ENV);
```

Как вы заметили lodash, исправим его отсутствие выполнением следующей команды:

```
$ npm i lodash --save-dev
```

Немного схитрив, мы сможем используя метод ***merge*** – библитеки lodash, 'склеить' нужную нам исходную конфигурацию, используя для этого 2 файла. В качестве аргумента функции принимая process.env.

./config/global.js

Файл содержит ~~ненормативную лексику~~ почти всю логику работы сборщика, следовательно к его содержимому нужно отнестись более ответственно:

```
'use strict'

var path        = require('path');
var webpack     = require('webpack');
var Manifest    = require('manifest-revision-webpack-plugin');
var TextPlugin  = require('extract-text-webpack-plugin');
var HtmlPlugin    = require('html-webpack-plugin');

module.exports = function(_path) {
```

```
  //define local variables
  var dependencies  = Object.keys(require(_path +
'/package').dependencies);
  var rootAssetPath = _path + 'app';

  return {
    // точки входа
    entry: {
      application: _path + '/app/app.js',
      vendors: dependencies
    },

    // то, что получим на выходе
    output: {
      path: path.join(_path, 'dist'),
      filename: path.join('assets', 'js', '[name].bundle.
[chunkhash].js'),
      chunkFilename: '[id].bundle.[chunkhash].js',
      publicPath: '/'
    },

    // Небольшие настройки связанные с тем, где искать сторонние
библиотеки
    resolve: {
      extensions: ['', '.js'],
      modulesDirectories: ['node_modules'],
      // Алиасы - очень важный инструмент для определения области
видимости ex. require('_modules/test/index')
      alias: {
        _svg:         path.join(_path, 'app', 'assets', 'svg'),
        _data:        path.join(_path, 'app', 'data'),
        _fonts:       path.join(_path, 'app', 'assets', 'fonts'),
        _modules:     path.join(_path, 'app', 'modules'),
        _images:      path.join(_path, 'app', 'assets', 'images'),
        _stylesheets: path.join(_path, 'app', 'assets',
'stylesheets'),
        _templates:   path.join(_path, 'app', 'assets', 'templates')
      }
    },
    // Настройка загрузчиков, они выполняют роль обработчика
исходного файла в конечный
    module: {
      loaders: [
        { test: /\.jade$/, loader: 'jade-loader' },
        { test: /\.
(css|ttf|eot|woff|woff2|png|ico|jpg|jpeg|gif|svg)$/i, loaders:
['file?context=' + rootAssetPath +
```

```
'&name=assets/static/[ext]/[name].[hash].[ext]'] },
        { test: /\.styl$/, loader: TextPlugin.extract('style-
loader', 'css-loader!autoprefixer-loader?browsers=last 5
version!stylus-loader') }
      ]
    },

    // загружаем плагины
    plugins: [
      new webpack.optimize.CommonsChunkPlugin('vendors',
'assets/js/vendors.[hash].js'),
      new TextPlugin('assets/css/[name].[hash].css'),
      new Manifest(path.join(_path + '/config', 'manifest.json'), {
        rootAssetPath: rootAssetPath
      }),
      // Этот файл будет являться "корневым" index.html
      new HtmlPlugin({
        title: 'Test APP',
        chunks: ['application', 'vendors'],
        filename: 'index.html',
        template: path.join(_path, 'app', 'index.html')
      })
    ]
  }
};
```

Аттеншен господа!

Появились новые зависимости – которые нужно доставить в проект следующей командой:

```
$ npm i path manifest-revision-webpack-plugin extract-text-webpack-
plugin html-webpack-plugin --save-dev
```

Разработка

Методом проб и ошибок, оптимальный для меня конфиг режима разработки – стал выглядеть следующим образом:

```
'use strict';

/**
 * Development config
 */
module.exports = function(_path) {

  return {
    context: _path,
    debug: true,
    devtool: 'eval',
    devServer: {
      contentBase: './dist',
      info: true,
      hot: false,
      inline: true
    }
  }
};
```

Окончательная сборка

А вот настройка конечное сборки все еще в стадии «изменений». Хотя работает на 5+

```
'use strict';

/**
 * Production config
 */
module.exports = function(_path) {
  return {
    context: _path,
    debug: false,
    devtool: 'cheap-source-map',
    output: {
      publicPath: '/'
    }
  }
}
```

index.html

Этот шаблон, мы положим в папку **./app/index.html** – именно он будет отдавать правильные пути до хешированной статики, после конечной сборки.

```
<!DOCTYPE html>
<html{% if(o.htmlWebpackPlugin.files.manifest) { %} manifest="{%=
o.htmlWebpackPlugin.files.manifest %}"{% } %}>
  <head>
    <meta charset="UTF-8">
    <title>{%=o.htmlWebpackPlugin.options.title || 'Webpack App'%}
</title>
    {% for (var css in o.htmlWebpackPlugin.files.css) { %}
      <link href="{%=o.htmlWebpackPlugin.files.css[css] %}"
rel="stylesheet">
    {% } %}
  </head>
  <body>
    {% for (var chunk in o.htmlWebpackPlugin.files.chunks) { %}
    <script src="{%=o.htmlWebpackPlugin.files.chunks[chunk].entry
%}"></script>
    {% } %}
  </body>
</html>
```

В заключении

Хотелось бы поблагодарить разработчиков webpack за столь мощный инструмент.
Скорость его работы действительно поражает, даже на больших объемах исходных файлов.

Ах, да. Для того чтобы в своем проекте использовать например **underscore** достаточно установить его привычной npm командой

```
$ npm i underscore --save
```

После выполнения, библиотека попадет в dependencies – следовательно webpack, поместит его содержимое в файл vendors.[hash].js при этом захеширует название файла полученное от md5 размера исходников + время последнего их изменения.

Для чего это надо, объяснять не буду.
На этом все, пробуйте, пишите комменты.
Спасибо.

Ссылки

Тут можно посмотреть код приведенный в статье
А тут ознакомиться с документацией по проекту
Ну и тут статья, которая поможет разложить все еще раз.

Комментарии (61)

Похожие публикации

Node.js. Паттерны проектирования и разработки

5

ПЕРЕВОД

ph_piter • 26 февраля 2016 в 13:15

Тонкости Javascript/Node.js. Увеличиваем производительность в десятки раз

36

ИЗ ПЕСОЧНИЦЫ

peacecoder85 • 24 февраля 2016 в 13:46

Node.js в бою (создание кластера)

17

ПЕРЕВОД

AndreyNagih • 9 ноября 2015 в 08:50

Популярное за сутки

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

69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов

20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке 31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s 49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить 85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

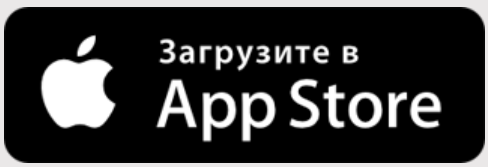
marks • вчера в 14:19

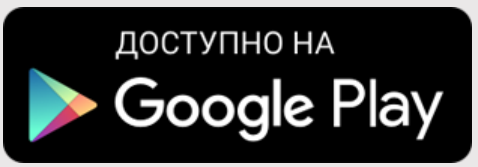
Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16

Мобильное приложение





Полная версия