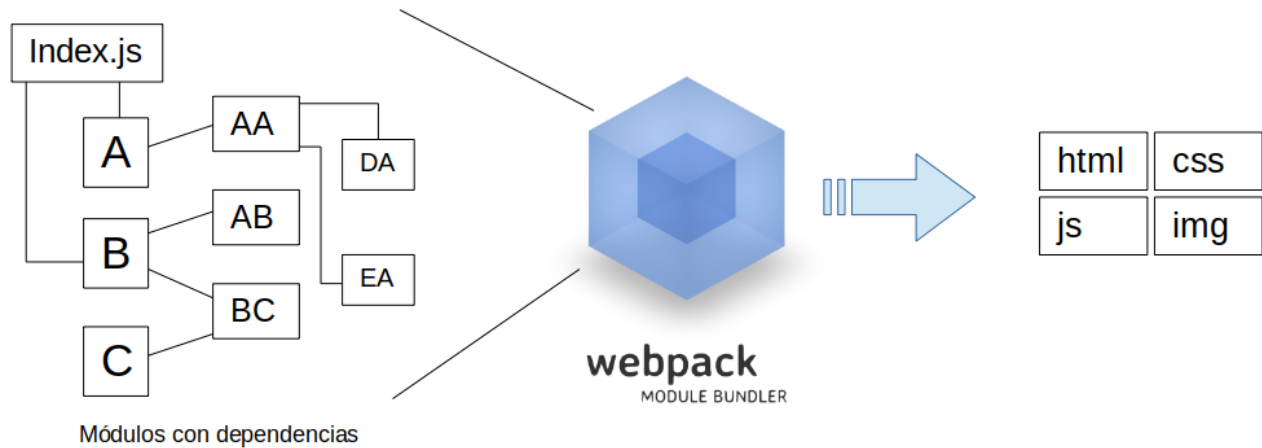


Webpack



WEBPACK

Module Bundler form modern JavaScript applications.

Entry Points: Nuestro modulo principal, de donde se parte a importar los demás módulos. Este es le archivo que leerá webpack para generar el bundle. (Es el archivo principal que llama a los demás archivos, como boostrab, jquery...)

Output: Configuración sobre el archivo resultante. Donde estará este archivo, Como se llamará...

Webpack trabaja con dos cosas principalmente.

Loaders: Nos ayudará a cargar todo tipo de formato de archivos como imágenes en (jpg, png, gif), fuentes personalizadas o de íconos y hasta “dialectos” como coffescript, typescript, stylus, sass, jsx...

Plugins: Nos ayudarán a extender las características de webpack, como para comprimir archivos usando Uglify o a dividir nuestros módulos en chuks mas pequeños para que nuestra aplicación cargue mas rápido.

Instalación.

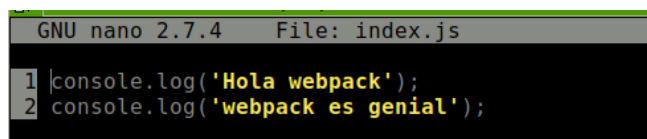
```
$ npm init
```

```
$ npm install webpack --save-dev
```

--save → para que se guarde en el package.json, -dev → para que se guarde como dependencia de desarrollo

```
$ npm list webpack → para ver la versión local de webpack.
```

Ahora vamos a crear nuestro “entry point” para lo cual creamos el archivo index.js como ejemplo.




```
GNU nano 2.7.4 File: index.js
1 |console.log('Hola webpack');
2 |console.log('webpack es genial');
```

Luego dentro de nuestro package.json debemos agregar una nueva tarea como vemos a continuación:

```

{
  "name": "curso-webpack",
  "version": "1.0.0",
  "description": "curso de platzy",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1",
    "build": "webpack index.js bundle.js"
  },
  "keywords": [
    "webpack"
  ],
  "author": "Byhako",
  "license": "MIT",
  "devDependencies": {
    "webpack": "^3.10.0"
  }
}

```



Donde la primera palabra <build> es el nombre que le asignamos a esta tarea, y lo segundo es la tarea que deseamos que se ejecute la cual tiene dos argumentos:

```

webpack <entry> <output>
webpack index.js bundle.js

```

Luego ejecutamos esta tarea como: `$ npm run build`

```

--> webpack(master)$ npm run build

> curso-webpack@1.0.0 build /home/ruben/github/Platzy/webpack
> webpack index.js bundle.js

Hash: 31ef4c7deae8609e48a1
Version: webpack 3.10.0
Time: 492ms
   Asset      Size  Chunks             Chunk Names
bundle.js  2.5 kB      0  [emitted]  main
   [0] ./index.js 28 bytes {0} [built]
--> webpack(master)$ |

```

y con esto tenemos nuestro primer bundle.

[ENTRY POINTS Y EL OUTPUT](#)

Creamos un archivo llamado webpack.config.js en el que estaran nuestros **entry** points y el **output**.

```

1 module.exports = {
2
3   entry: './index.js',
4   output: {
5     filename: './bundle.js'
6   }
7 }

```

Ahora debemos configurar una nueva tarea en el package.json como:

```
6  "scripts": {
7    "test": "echo \\\"Error: no test specified\\\" && e$
8    "build": "webpack index.js bundle.js",
9    "build:local": "webpack"
10 |
11 },
```

Vemos que en la ultima linea llamamos a webpack, y este por defecto busca el archivo llamado webpack.config.js

Luego podemos correr: `$ npm run build:local`

y tenemos nuestro bundle construido.

=====

Si tenemos el archivo webpack.config.js en otra carpeta, digamos en una que se llama external, podemos llamarlo escribiendo su ruta dentro del package.json como sigue:

`"build:external": "webpack --config ./external/webpack.config.js"`

```
6  "scripts": {
7    "test": "echo \\\"Error: no test specified\\\" && exit 1",
8    "build": "webpack index.js bundle.js",
9    "build:local": "webpack",
10   "build:external": "webpack --config ./external/webpack.config.js"
11 },
```

El flag → `--config` es para indicarle que debe buscar en el path que tiene a continuación.

Ahora para que el bundle.js se construya dentro de la carpeta external, y para que lea el index.js debemos modificar el archivo webpack.config.js:

```
1  const path = require('path');
2
3  module.exports = {
4
5    entry: path.resolve(__dirname, 'index.js'),
6    output: {
7      path: path.resolve(__dirname, 'dist'),
8      filename: 'bundle.js'
9    }
10 }
```

Vemos que el archivo bundle.js se va a crear dentro de una carpeta llamada dist (distribución) que estará dentro de external. Ahora al ejecutar el comando `$ npm run build:external` construimos el bundle, y tenemos un arbol de archivos como:

```
external/  
├─ dist  
│   └─ bundle.js  
├─ index.js  
└─ webpack.config.js  
  
1 directory, 3 files
```

=====

LOADERS

Los loaders nos permiten soportar otros tipos de archivos dentro de nuestro archivo js. Estos se cargan dentro del archivo webpack.config.js, en una nueva key que se llama “**module**” y dentro de module, otro key que se llama “**rules**”, y rules recibe un array que es una lista de los loaders que vamos a utilizar. Si por ejemplo, deseamos cargar archivos css desde index.js, necesitamos instalar dos loaders, pues estos no vienen precargados cuando instalamos webpack, entonces los instalamos:

```
$ npm install style-loader css-loader --save-dev
```

y el webpack queda como:

```
1  const path = require('path');  
2  
3  module.exports = {  
4  
5    entry: path.resolve(__dirname, 'index.js'),  
6    output: {  
7      path: path.resolve(__dirname, 'dist'),  
8      filename: 'bundle.js'  
9    },  
10   module: {  
11     rules: [  
12       // Aquí van los loaders  
13       {  
14         // test: que tipo de archivo quiero reconocer, (expresión regular)  
15         // use: que loader se va a encargar del archivo  
16         test: /\.css$/,  
17         use: ['style-loader', 'css-loader'],  
18       }  
19     ]  
20   }  
21 }
```

Es muy importante el orden, primero style-loader, y luego css-loader.

Luego debemos añadir una nueva tarea al package.json:

```
"build:css": "webpack --config ./css-style-loader/webpack.config.js",
```

Supongamos que vamos a cargar un archivo css llamado estilos.css, entonces lo podemos hacer desde el index.js como:

```
1 import './estilos.css';
2
3 console.log('Hola webpack');
4 console.log('webpack es genial');
```

y listo, podemos construir el bundel con `$ npm run build:css`

=====

PLUGINS

Cuando añadíamos css usando loader, estos estilos eran añadidos en una etiqueta style dentro del head del html. Para que todos los css queden en un archivo independiente podemos usar el plugin “extract-text-webpack-plugin” entonces lo instalamos:

```
$ npm install extract-text-webpack-plugin --save-dev
```

Ahora, en el archivo webpack.config.js, después de la key **module**, añadimos una nueva key llamada **plugins**, que será un array de plugins. Dentro de la key **module**, debemos cambiar la manera en la que está declarado la key **use**. Veamos como queda el webpack.config.js

```
1 const path = require('path');
2 const extractTextPlugin = require('extract-text-webpack-plugin');
3
4 module.exports = {
5
6   entry: path.resolve(__dirname, 'index.js'),
7   output: {
8     path: path.resolve(__dirname, 'dist'),
9     filename: 'bundle.js'
10  },
11  module: {
12    rules: [
13      // Aquí van los loaders
14      {
15        // test: que tipo de archivo quiero reconocer, (expresión regular)
16        // use: que loader se va a encargar del archivo
17        test: /\.css$/,
18        use: extractTextPlugin.extract({
19          fallback: "style-loader",
20          use: "css-loader"
21        }),
22      }
23    ]
24  },
25  plugins: [
26    // aquí van los plugins
27    new extractTextPlugin("css/styles.css") // donde se guardan archivos extraídos
28  ]
29 }
```

También debemos añadir una nueva tarea en el package.json.

```
"build:extract": "webpack --config ./plugin-extract-text/webpack.config.js",
```

y listo, podemos construir el bundel con `$ npm run build:css`

En este caso debemos incluir el archivo css generado en la cabecera de nuestro html. Para nuestro ejemplo seria → `<link rel="stylesheet" href="dist/css/styles.css">`
Si tenemos múltiples archivos css, podemos incluirlos todos en la cabecera del html, y en nuestro webpack.config.js, los llamamos como:

```
new extractTextPlugin("css/[name].css") // donde se guardan archivos
```

donde la expresión `[name].css` hace que se carguen todos los archivos .css que se halla incluido en el html.

=====

MULTIPLES ENTRY POINTS

Solo necesitamos cambiar la forma en la que en el webpack.config.js esta la key `entry`, veamos:

```
entry: {
  home: path.resolve(__dirname, './src/js/index.js'),
  precios: path.resolve(__dirname, './src/js/precios.js'),
  contacto: path.resolve(__dirname, './src/js/contacto.js')
},
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'js/[name].js'
},
```

=====

WEBPACK DEVSERVER

Para ver los cambios de nuestro proyecto sin tener que estar compilando con `npm run` cada vez que hay un cambio.

En el archivo package.json, añadimos una nueva tarea:

```
"build:server": "webpack --config ./devserver/webpack.config.js --watch"
```

vemos la ultima etiqueta "--watch" la cual indica que al ejecutar el comando `$ npm run build:server` el sistema permanecerá escuchando nuevos cambios en los archivos dentro de la carpeta devserver, y los actualizara en en navegador automáticamente.

```
--> webpack(master)$ npm run build:server

> curso-webpack@1.0.0 build:server /home/ruben/github/Platzy/webpa
ck
> webpack --config ./devserver/webpack.config.js --watch

Webpack is watching the files...
Hash: 0ba88fff18e75d53559b
Version: webpack 3.10.0
Time: 530ms

  Asset      Size  Chunks             Chunk Names
bundle.js  19.7 kB          0  [emitted]  main
   [0]  ./devserver/index.js  120 bytes {0} [built]
   [1]  ./devserver/estilos.css  1.08 kB {0} [built]
   [2]  ./node_modules/css-loader!./devserver/estilos.css  313 bytes
   {0} [built]
+ 3 hidden modules
```

Aun necesitamos recargar la pagina en el navegador para ver los cambios. Para que esto no sea necesario primero instalamos `$ npm install webpack-dev-server --save-dev`

Otra forma de poner las etiquetas --save-dev es colocar -D, y hace lo mismo.

Ahora debemos cambiar la tarea en el package.json, y definirla como:

```
"build:server": "webpack-dev-server --config ./devserver/webpack.config.js"
```

Luego de esto, al ejecutar el comando `$ npm run build:server` tenemos:

```
Project is running at http://localhost:8080/
webpack output is served from /
Hash: 8d0a67e57ee54069546f
Version: webpack 3.10.0
Time: 3025ms

  Asset      Size  Chunks             Chunk Names
bundle.js  341 kB          0  [emitted]  [big]  main
   [2] multi (webpack)-dev-server/client?http://localhost:
```

Donde vemos que el sistema esta escuchando en el puerto 8080.

En este caso, tenemos definido el archivo bundle.js como nuestro output, por tanto cada vez que halla un cambio, y se recompila, este archivo va a cambiar. En nuestro index.html entonces debemos cambiar la linea

```
<script src="dist/bundle.js"></script>
```

por


```
<script src="http://localhost:8080/bundle.js"></script>
```

Ahora cada vez que realicemos cambios en nuestros entry points, estos se verán reflejados automáticamente en el navegador.

El puerto y otras cosas del server se pueden configurar con una nueva key llamada **devServer** en el archivo `webpack.config.js`. Para el puerto por ejemplo:

```
10 devServer:{
11   port: 8000,
12 },
```

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

EcmaScript

Para tener soporte en todos los navegadores usando código javascript moderno usamos Babel.

<https://babeljs.io/>

Para usarlo en el webpack necesitamos un loader. En el caso de babel podemos encontrarlo en:

<https://github.com/babel/babel-loader>

Install

webpack 3.x | babel-loader 8.x | babel 7.x

```
npm install babel-loader@8.0.0-beta.0 @babel/core @babel/preset-env webpack
```

webpack 3.x | babel-loader 7.x | babel 6.x

```
npm install babel-loader babel-core babel-preset-env webpack
```

Para este ejemplo, creamos una nueva carpeta llamada babel-loader.

En nuestro package.json creamos una nueva tarea para babel:

```
"build:babel": "webpack --config ./babel-loader/webpack.config.js"
```

Ahora vamos a instalar las dependencias necesarias y modificar el webpack.config.js.

Ejecutamos:

```
$ npm install babel-loader babel-core babel-preset-es2015 babel-preset-es2016 -D
```

la parte de babel-preset-es2015, es con la que indico a que versión de EcmaScript quiero darle soporte.

Una vez hecho esto, vamos a configurar los loaders en el webpack.config.js. Como ahora vamos a cargar archivos js, solo debemos añadir dentro de la key **rules** un ítem mas, el cual es:

```
{
  test: /\.js$/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ['es2015']
    }
  },
},
```

En este caso, con la opción presets, estamos dando soporte para archivos escritos en Ecma2015, pero podemos soportar los que necesitemos.

=====

[REACT](#)

Para este ejemplo, creamos una carpeta llamada react.

Necesitamos ahora dar soporte a jsx, lo cual puede ser configurado como un presets de babel.

<https://babeljs.io/docs/plugins/preset-react/>

Corremos:

```
$ npm install --save-dev babel-cli babel-preset-react
```

```
$ npm install react react-dom --save
```

react no es dependencia de desarrollo, es core del proyecto, por eso no lleva el flag -dev

Ahora, agregamos una tarea al package.json.

```
"build:react": "webpack --config ./react/webpack.config.js"
```

En cuanto al webpack.config.js, basta con añadir react en el presets de js:

```
// webpack.config.js
{
  test: /\.js$/,
  use: {
    loader: 'babel-loader',
    options: {
      presets: ['es2015', 'react']
    }
  },
},
```

Y listo, ya podemos comenzar con nuestra aplicación en react.

Imágenes

<https://github.com/webpack-contrib/url-loader>

```
test: /\. (jpg|png|gif)$/ ,
use: {
  loader: 'url-loader',
  options: {
    limit: 100000, // máximo peso de la imagen en bytes
  }
}
```

Necesitamos cargar este loader. `npm install --save-dev url-loader`

Fuentes.

www.fontsquirrel.com

una vez seleccionada la fuente deseada vamos a la pestaña de webfont kit, seleccionamos todos los formatos, y lo descargamos.

En el webpack, le damos soporte a las fuentes tipo woff, eot, ttf, y svg

```
test: /\. (jpg|png|gif|woff|eot|ttf|svg)$/ ,
use: {
  loader: 'url-loader',
  options: {
    limit: 100000, // máximo peso de la imagen en bytes
  }
}
```

Debemos ahora copiar el archivo css de las fuentes en nuestros estilos.css

Las fuentes las copiamos en una carpeta llamada fonts.

Podríamos tener un error si las fuentes que tenemos pesan mas de lo que estipulamos en el limite.

Archivos JSON

<https://github.com/webpack-contrib/json-loader>

```
{
  test: /\.json$/,
  use: 'json-loader'
},
```

Vídeos.

<https://github.com/webpack-contrib/file-loader>

```
{
  test: /\. (webm|mp4)$/ ,
  use: {
    loader: 'file-loader',
    options: {
      limit: 300000, // maximo peso del video en bytes
      name: 'videos/[name].[hash].[ext]'
    }
  }
},
```

Los vídeos que no entran en el margen de tamaño máximo permitido sera ubicados en la carpeta llamada videos/...

También debemos configurar el output, añadiendo cual es el directorio publico desde el que se busaran todos los demás archivos.

```
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'bundle.js',
  publicPath: './dist/'
}
```

SASS

<https://github.com/webpack-contrib/sass-loader>

```
{
  test: /\.scss$/, // sass
  use: extractTextPlugin.extract({
    use: ["css-loader", "sass-loader"]
  }),
}
```

Stylus

<https://github.com/shama/stylus-loader>

```

{
  test: /\.styl$/,
  use: ExtractTextPlugin.extract({
    // ['style-loader', 'css-loader']
    // fallback: 'style-loader',
    use: [
      "css-loader",
      {
        loader: 'stylus-loader',
        options: {
          use: [
            require('nib'),
            require('rupture')
          ],
          import: [
            '~nib/lib/nib/index.styl',
            '~rupture/rupture/index.styl'
          ]
        }
      }
    ]
  }),
}

```

Less

<https://github.com/webpack-contrib/less-loader>

```

{
  test: /\.less$/,
  use: ExtractTextPlugin.extract({
    use: ["css-loader", {
      loader: 'less-loader',
      options: {
        noIeCompat: true,
      }
    }]
  }),
},

```

PostCss

<https://github.com/postcss/postcss-loader>

<https://www.npmjs.com/package/postcss-cssnext>

<https://github.com/MoOx/postcss-cssnext>

```
{
  test: /\.css$/,
  use: ExtractTextPlugin.extract({
    use: [
      {
        loader: 'css-loader',
        options: {
          modules: true,
          importLoaders: 1
        }
      },
      'postcss-loader'
    ]
  })
},
```

Prevenir duplicación

Cuando tenemos multiples entryPoints,

```
entry: {
  home: path.resolve(__dirname, 'src/js/index.js'),
  contact: path.resolve(__dirname, 'src/js/contact.js')
},
```

si varios están asociados a componentes de react o angular, por ejemplo, multiplicaremos el código de react/angular por cada entryPoint ya que en ambos estamos importando los mismos paquetes en algunos casos, como por ejemplo react, o react-dom. Para evitar esto, en nuestro webpack importamos webpack:

```
const webpack = require('webpack')
```

luego debemos añadir un plugin:

```
plugins: [
  // aquí van los plugins
  new ExtractTextPlugin("css/[name].css"),
  new webpack.optimize.CommonsChunkPlugin({
    name: 'common'
  })
]
```

este plugin lo que hace es capturar todo el código común y empaquetarlo en un nuevo archivo, que en este caso hemos llamado 'common', así solo lo tendremos una vez.

Cuando tenemos múltiples entryPoints debemos ponerle nombres dinámicos a las salidas.

```
output: {
  path: path.resolve(__dirname, 'dist'),
  filename: '[name].js'
```

El nombre dinámico esta en la ultima linea.

Ahora en el html debemos cargar el modulo principal de cada entryPoint, y los archivos comunes.

```
<section id="container"></section>
<script src="dist/common.js"></script>
<script src="dist/home.js"></script>
<script src="dist/contac.js"></script>
```

Es importante que el primer archivo sea el de los archivos comunes.

<https://gist.github.com/sokra/1522d586b8e5c0f5072d7565c2bee693>

<https://webpack.js.org/plugins/dll-plugin/>

Entorno de producción

Una de las cosas mas importantes aquí, es la minificación de los archivos. En el caso de los archivos css, lo hacemos activando la opción minimize:

```
{
  test: /\.css$/,
  use: ExtractTextPlugin.extract({
    use: [
      {
        loader: 'css-loader',
        options: {
          minimize: true,
        }
      }
    ]
  })
},
```

Tambien es buena idea usar el plugin `const ExtractTextPlugin = require('extract-text-webpack-plugin');` con este se extrae el css como un archivo que sera cargado en tiempo de ejecución. Este plugin se coloca en la etiqueta plugins como:

```
new ExtractTextPlugin("css/[name].[hash].css")
```


En el caso de los archivos js, se hace en la configuración del package.json:

```
"build:prod": "webpack -p --env.NODE_ENV=production"
```

con la opción -p, estamos haciendo esto. Lo siguiente es una variable de entorno con el valor producción asignado, que le pasamos a nuestro webpack para añadir algunos plugins solo en caso de estar activada.

Otra buena practica es eliminar la carpeta dist, que es el output, cada vez que compilemos el código para producción, para asegurarnos que siempre tenemos únicamente los archivos de producción mas recientes. Hay un plugin que nos ayuda con esto,

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

Este lo usamos solo cuando la variable de entorno llegue con el valor 'producción'.

```
if (env.NODE_ENV === 'production') {  
  plugins.push(  
    new CleanWebpackPlugin(['dist'], {root: __dirname})  
  )  
}
```

Recuerda que es importante tener un publicPath en nuestro output, que es donde el navegador a a leer nuestros archivos.

```
publicPath: path.resolve(__dirname, 'dist')+"/",
```

Podemos ver el progreso de la compilación del webpack usando las etiquetas:

```
npm run build:prod -- --progress --watch
```

* * * * * FIN * * * * *