Módulos en Javascript

Módulos en Javascript

Módulos en Javascript

¿Por qué necesitamos módulos?

- Scope (encapsulación)
- Tamaño de los archivos
- Fragilidad
- Legibilidad
- Experiencia del desarrollador

Módulos en Javascript

Sistemas de módulos:

- Funciones auto-ejecutables - IIFE
- AMD - Asynchronous module definition
- Common JS
- ES Modules (ES2015 modules) <———— Nuevo estándar ECMA

Módulos en Javascript

Funciones auto-ejecutables

```
(function () {
    var message = 'I'm a immediately invoked function expression';
    console.log(message);
 }
)();
```

Contras:

- No imports
- No es asíncrono
- Dependencias difíciles de manejar y puede haber dependencias circulares

Módulos en Javascript

Common JS

```
module.exports.sum = function(a, b) {
   return a + b;
};
—-
var sum = require('./sum.js');
sum(3,7);
```

Contras:

- No funciona directamente en la web. Hay que transpilarlo.
- No es asíncrono

Módulos en Javascript

AMD - Asyncronous module definition

```
define(['./sumModule'] , function (sumFunction) {
  return function () {
    var add3to2 = sumFunction(3,2);
    // …
  };
});
```

Contras:

- Sintaxis es compleja
- Las librerías son requeridas siempre (problemas de tamaño de build)

Módulos en Javascript

ES Modules (ES 2015, ES6)

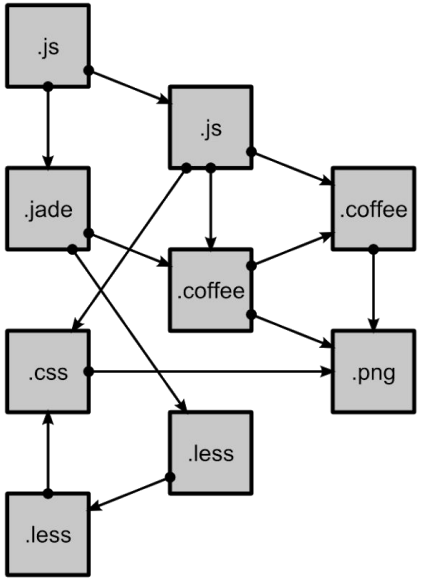
```
const sum = (a, b) => a + b;
—-
import { sum } from ‘./sum’;
sum(3,7)
```

Contras:

- No se puede utilizar en todos los navegadores

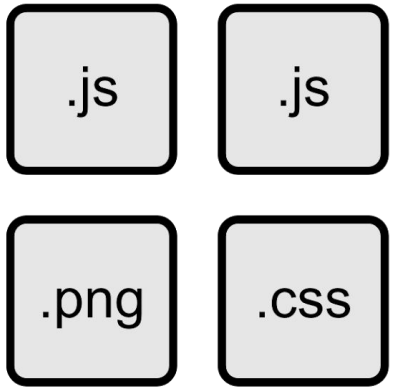
Los módulos procesados son más rápidos

Introducción



modules
with dependencies

webpack
MODULE BUNDLER



static
assets

KEEP CODING

Introducción

Webpack es un empaquetador de módulos (module bundler).

Es una herramienta que a partir de varios archivos de código fuente (módulos) nos permite generar un único archivo JavaScript y otro CSS y los assets.

- Código más rápido y compacto: Menos peticiones HTTP y más ligeras

- Código mucho más mantenible

Introducción

Webpack es el estándar: Se utiliza para todos los grandes frameworks actuales: React, Vue y Angular.

Es la manera más eficiente de desplegar Javascript hoy en día.

Incluso se puede utilizar para empaquetar de archivos fuera de la web: Electron, …

Introducción

Webpack es module bundler + task runner (Grunt, Gulp, …)

Un task runner nos permite utilizar ciertas tareas que nos ayudarán a procesar nuestro código:

- Minificación
- Compilación (ES6)
- Code splitting

…

Introducción

Webpack es module bundler + task runner + herramienta de desarrollo

- Dev server: Servidor local de código optimizado
- Hot module replacement
...

Introducción

- Funciona a través de plugins: Gran ecosistema de módulos de procesamiento

- Genera un grafo de dependencias entre módulos para este procesamiento

Instalación

Definir nuestro package.json

```
npm init -y
```

Instalando webpack

```
npm install --save-dev webpack webpack-cli
```

Ejecución

Lo damos de alta como script npm

```
// en el package.json

"scripts": {
    "webpack": "webpack",
}
```

Ejecutamos con pm

npm run webpack

Modo observer

```
// en el package.json

"scripts": {
    “dev": “webpack --watch“,
}
```

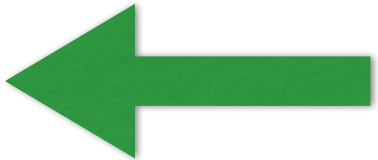
Definiciones

Entry point

Un **entry point** indica el módulo por el cual Webpack recogerá los archivos (source files) y empezará a construir el grafo de dependencias

Al procesar este entry point buscará las dependencias (módulos y librerías) de este módulo inicial

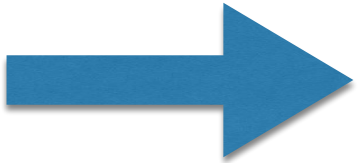
Por defecto: src/index.js



Output

La propiedad **output** le dice a webpack donde dejar los archivos generados en el proceso de construcción.

Por defecto: dist/main.js



Loaders

Webpack solo puede procesar archivos javascript en un principio.

Los loaders le permiten procesar otro tipo de archivos (.css, .scss, .html, .jpeg, .ejs), transformándolos en módulos que pueden incluirse en el grafo de dependencias de la aplicación.

Nos ayudan a saber qué hacer con estos tipos de archivos y procesarlos correctamente

Plugins

Los plugins son módulos de procesamiento que tienen una función concreta a la hora de transformar los archivos:
- Optimización de paquetes, definición de variables de entorno, …

El 80% del código de Webpack son plugins y hay una gran comunidad creándolos, manteniéndolos y adaptándolos a las nuevas versiones

Configuración

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: path.join(__dirname, 'src', 'index'),
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

Entendiendo la configuración

1) Webpack coge el **entry** point

2) Lee el contenido y, cada **import** realizado, procesa el contenido…

3) Lo añade al archivo **output** de salida.

Placeholders

Placeholders que se pueden utilizar en el output:

[name]: Nombre del archivo, por defecto main
[folder]: Carpeta del archivo
[hash]: Hash generado automáticamente por build
[chunkhash]: Hash generado automáticamente por cada entry file
…

Dev server

Olvídate de recargar el navegador con cada cambio de código.

Instalamos webpack-dev-server

```
npm install --save-dev webpack-dev-server
```

Modificamos package.json

```
"scripts": {
   ...
   "start": "webpack-dev-server"
},
```

Modificamos webpack.config.js

```
devServer: {
  open: true, // abre el navegador por defecto
  overlay: true,  // muestra errores en pantalla
  port: 3000 // puerto de escucha
}
```

Empaquetando HTML

Empaquetando HTML

Para automatizar el empaquetado de HTML, debemos utilizar el plugin **html-webpack-plugin**.

Instalamos html-webpack-plugin

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

Modificamos webpack.config.js

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

[…]

plugins: [
  new HtmlWebpackPlugin({
     template: “src/index.html”
  })
]
```

Minificando código HTML

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

[…]

plugins: [
  new HtmlWebpackPlugin({
    template: “src/index.html”,
    minify: {
      collapseWhitespace: true
    }
  })
]
```

https://www.npmjs.com/package/html-minifier

Usando ES6 con Babel

Usando ES6 con Babel

Para poder usar ES6, necesitamos utilizar el loader **babel-loader y babel.**

KEEP CODING

Instalamos dependencias

```
npm install --save-dev babel-loader babel-core babel-preset-env
```

Crear archivo .babelrc

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

Modificamos webpack.config.js

```
module: {
  rules: [
    {
      test: /\.js$/,
      use: ‘babel-loader’,
      exclude: path.join(__dirname, 'node_modules')
    }
  ]
}
```

Empaquetando CSS

Empaquetando CSS

Para automatizar el empaquetado de CSS, debemos utilizar dos loaders:

- css-loader: para soportar la carga de CSS en el entry point
- style-loader: para cargar automáticamente el CSS en el HTML

Instalamos style-loader y css-loader

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

Modificamos webpack.config.js

```
module: {
  rules: [
    {
      test: /\.css$/,
      use: [ 'style-loader', 'css-loader' ]
     }
  ]
},
```

Hot Module Replacement

Evitar que se recargue el navegador para actualizar CSS

Activar HMR en devServer en webpack.config.js

```
devServer: {
    open: true,
    overlay: true,
    port: 3000,
    hot: true,
    contentBase: path.join(__dirname, 'src'),
    watchContentBase: true
},
```

Modificamos webpack.config.js

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

plugins: [
    ...
    new webpack.HotModuleReplacementPlugin(),
]
```

Empaquetando SASS

Empaquetando SASS

Para automatizar el empaquetado de SASS, debemos utilizar el sass-loader y tener instalado node-sass (compilador de SASS).

Instalamos sass-loader y node-sass

```
npm install --save-dev sass-loader node-sass
```

Modificamos webpack.config.js

```
module: {
  rules: [
    {
      test: /\.scss$/,
      use: [ 'style-loader', 'css-loader', 'sass-loader' ]
    }
  ]
},
```

Desarrollo vs producción

Separamos la configuración de dev y production

```
const commonConfig = {
    entry: …,
    …
};

const devConfig = {
  devServer: {
    overlay: true, …
  }, …
};
```

```
const productionConfig = {};
```

De este modo podremos añadir la configuración dependiendo de nuestro entorno

Utilizamos webpack merge para unirla

```
merge = require("webpack-merge")

merge ({ a: [1], b: 5, c: 20 },  { a: [2], b: 10, d: 421 })

>> { a: [ 1, 2 ], b: 10, c: 20, d: 421 }
```

```
module.exports = mode => (
  mode === 'development'
    ? merge(commonConfig, devConfig)
    :  merge(commonConfig, productionConfig)
```

Modificamos package.json

```
"scripts": {
    "start" : "webpack-dev-server mode=development"
    "prod": "webpack mode=production"
},
```

Cargando assets

Usando assets

Para que Webpack recoja los assets, necesitamos utilizar el loader **file-loader** y hacer referencia a dichos assets desde javascript o en HTML.

En JavaScript:

```
import pdf from ‘./assets/catalog.pdf’;
```

En HTML:

```
<a href=“./assets/catalog.pdf” >
    Download catalogue
</a>
```

Instalamos dependencias

```
npm install --save-dev file-loader
```

Modificamos webpack.config.js

```
module: {
  rules: [
    {
      test: /assets\/.*/,
      use: 'file-loader?name=[name].[ext]'
    }
  ]
}
```

Optimizando imágenes

Optimizando imágenes

Para optimizar el tamaño de las imágenes, necesitamos utilizar el loader **image-webpack-loader**.

Instalamos dependencias

```
npm install --save-dev image-webpack-loader
```

Modificamos webpack.config.js

```
use:
  { loader: 'file-loader',
    options: {
      name: '[path][name].[hash].[ext]'
    }
  },
  {
    loader: 'image-webpack-loader'
  }
]
}
```

Generando el CSS aparte

Generando el CSS aparte

Para generar el CSS a parte, necesitamos utilizar un plugin **mini-css-extract-plugin.**

Instalamos mini-css-extract-plugin

```
npm install --save-dev mini-css-extract-plugin
```

Modificamos webpack.config.js

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

module: {
  rules: [
    {
      test: /\.scss$/,
      use: [
        MiniCssExtractPlugin.loader,
        'css-loader',
        'sass-loader'
      ]
    }
  ]
}
```

Modificamos webpack.config.js

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



plugins: [
    ...
    new MiniCssExtractPlugin({ filename: ‘style.css' })
]
```

Minimizando CSS

Minimizando CSS

Para minimizar el CSS en produccion, necesitamos utilizar un plugin **optimize-css-assets-webpack-plugin.**

```
const OptimizeCSSAssetsPlugin = require(“optimize-css-assets-webpack-plugin");

module.exports = {
 optimization: {
    minimizer: [
      new OptimizeCSSAssetsPlugin({})
    ]
  }, …
```

Utilizando alias

Utilizando alias

Para utilizar rutas absolutas Webpack nos permite definir unos alias que podemos utilizar en nuestros imports.

```
module.exports = {
  resolve: {
    alias: {
      assets: path.resolve(__dirname, 'assets')
    }
  } ...
```

Utilizando sourcemaps

Utilizando sourcemaps

Los source maps son archivos que indexan nuestro Js y CSS y nos permitirán acceder al código base una vez minimizado

JS:

```
module.exports = {
  devtool: ‘source-map'
…
new UglifyJsPlugin({
  cache: true,
  parallel: true,
  sourceMap: true // set to true if you want JS source maps
}),
```

Utilizando sourcemaps

Los source maps son archivos que indexan nuestro Js y CSS y nos permitirán acceder al código base una vez minimizado

CSS:

```
...
new OptimizeCSSAssetsPlugin(
  {
    cssProcessorOptions: { map: { inline: false, annotation: true } }
  }
)
```

Generando varios HTML

Generando varios HTML

Nos vale con añadir otra entrada de **HtmlWebPackPlugin**.

```
module: {
  plugins: [
    new HtmlWebpackPlugin({
      filename: 'contact.html', // archivo de destino
      template: path.join(__dirname, 'src', ‘contact.html'),
      minify: { collapseWhitespace: true }
    }),
  ]
}
```

Usando plantillas EJS (de verdad)

Usando plantillas EJS

Para poder utilizar plantillas EJS de verdad, debemos instalar **ejs**, **html-loader** y **ejs-html-loader**.

Instalamos dependencias

npm install --save-dev ejs html-loader ejs-html-loader

Modificamos webpack.config.js

```
module: {
  rules: [
    {
      test: /\.(html|ejs)$/,
      use: ['html-loader', 'ejs-html-loader']
    }
  ]
}
```