



NODE COMO HERRAMIENTA DE DESARROLLO FRONT

© 2017, ACTIBYTI PROJECT SLU, Barcelona



GOBIERNO
DE ESPAÑA

MINISTERIO
DE ENERGÍA, TURISMO
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE
EMPRENDIMIENTO Y
EMPLEO JUVENIL
garantía juvenil



Agenda Digital para España



UNIÓN EUROPEA

Fondo Social Europeo
“El FSE invierte en tu futuro”

ÍNDICE DE CONTENIDOS

1. Node como herramienta de desarrollo front
2. Grunt
3. Gulp
4. Webpack

1

NODE COMO HERRAMIENTA DE DESARROLLO FRONT

Node.JS

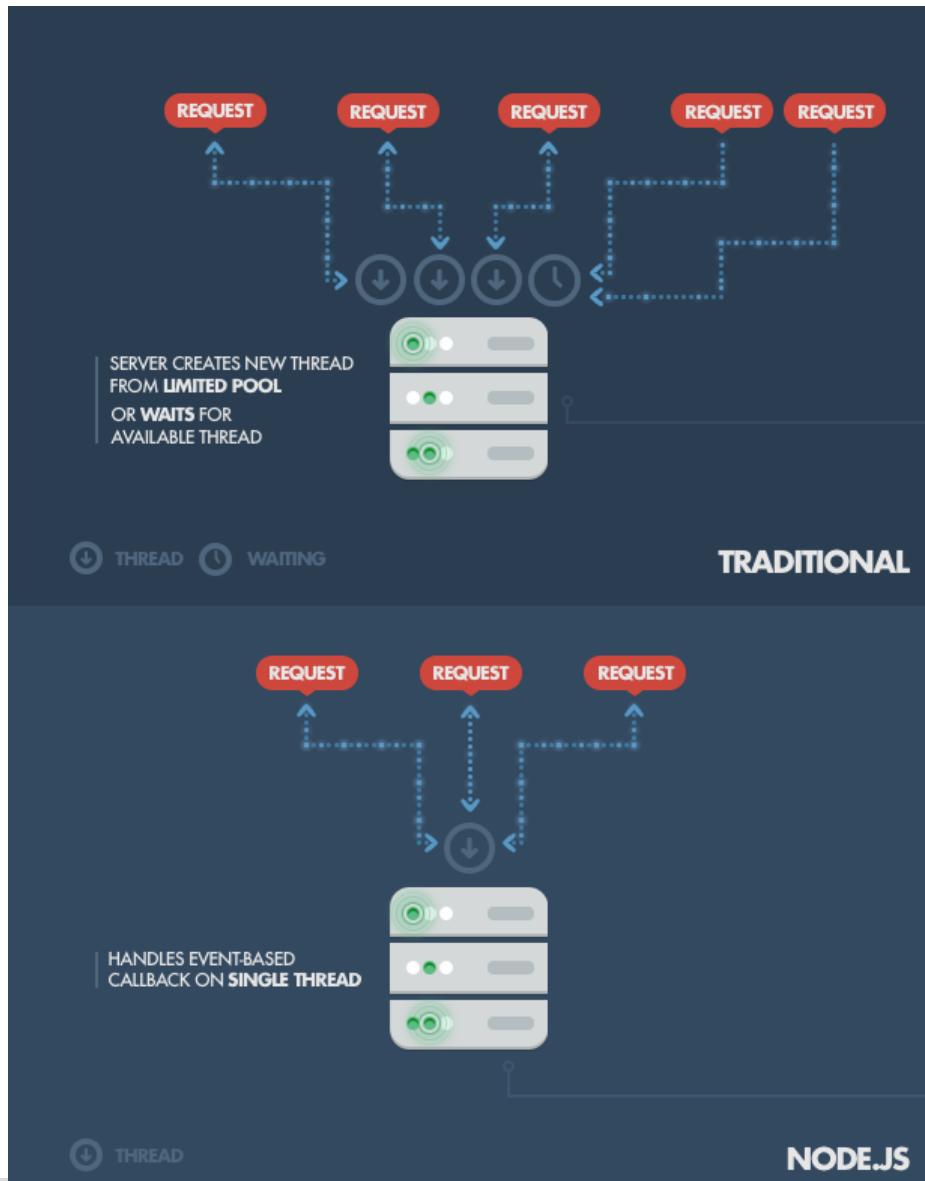


- Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.
 - <https://nodejs.org/en/>
- Node.js brilla en aplicaciones web de tiempo real empleando la tecnología push a través de Websockets, donde tanto el cliente como el servidor pueden iniciar la comunicación, lo que les permite intercambiar datos libremente.
- Esto está en contraste con el paradigma de respuesta web típica, donde el cliente siempre inicia la comunicación.
- Además, todo se basa en el Open Web Stack (HTML, CSS y JS) que se ejecuta en el puerto estándar 80.

Cómo funciona

- La idea principal de Node.js:
 - uso no-bloqueante, event-driven I/O, permanecer ligero y eficiente en la superficie del uso intensivo de datos en tiempo real de las aplicaciones que se ejecutan en dispositivos distribuidos.
- Node REALMENTE destaca en la construcción rápida y escalable de aplicaciones de red, debido a que es capaz de manejar un gran número de conexiones simultáneas con alto rendimiento, lo que equivale a una alta escalabilidad.
- Frente a las tradicionales técnicas de servicio web donde cada conexión (solicitud) genera un nuevo subprocesso, retomando la RAM del sistema y finalmente a tope a la cantidad de RAM disponible, Node.js opera en un solo subprocesso, no utiliza el bloqueo de llamadas de E/S, lo que le permite admitir decenas de miles de conexiones simultáneas (celebrada en el caso de loop).

Tradicional vs Node



➤ © Tomislav Capan

NPM: El Node Package Manager

- Node provee un conjunto de módulos que pueden apoyar y facilitar el desarrollo de componentes y aplicaciones front. Estos se gestionan mediante la herramienta NPM
- La idea de los módulos NPM es muy similar a la de Ruby Gemas: un conjunto de componentes reutilizables disponibles públicamente a través de una fácil instalación a través de un repositorio en línea, con la versión y la dependencia de gestión.
- Una lista completa de los paquetes de módulos puede encontrarse en el sitio web de NPM <Https://npmjs.org/> o acceder utilizando la herramienta de la CLI de NPM que automáticamente se instala con Node.js.
- En la siguiente dirección se puede ver una lista de módulos útiles
 - <https://www.codementor.io/ashish1dev/list-of-useful-nodejs-modules-du107mcv3>



Pongámoslo en práctica – Instalando Node.js

- Descargar e instalar la versión 6+ de Node:
<https://nodejs.org/en/>

- Comprobar que está correctamente instalado abriendo una ventana de consola y escribiendo: node -v

package.json

- En este archivo, que debe estar en la raíz del proyecto, va a quedar reflejada la configuración del **proyecto Node**:
 - Nombre del proyecto.
 - Autor.
 - Versión.
 - Dependencias.
 - Scripts.
 - Repositorio Git.
 - Motor de Node.
 - ...
- En la documentación de node hay una guía exhaustiva de su uso:
 - <https://docs.npmjs.com/getting-started/using-a-package.json>
- Podemos crear automáticamente un package.json para nuestro proyecto con la orden (con el flag –yes para no interactivo):

npm init

npm init --yes

Especificar dependencias

- Para especificar los paquetes de los que depende un proyecto, se deben listar los paquetes que desea utilizar en el archivo **package.json**.
- Hay dos tipos de paquetes que puede listar:
 - **Dependencies**: estos paquetes son requeridos por la aplicación en producción
 - **DevDependencies**: estos paquetes solo son necesarios para el desarrollo y pruebas
- Las dependencias se pueden editar manualmente o mediante los flags **-save** y **-save-dev** de **npm install**, para añadir una dependencia en Dependencies y DevDependencies respectivamente.

```
npm install <package_name> --save
```

```
npm install <package_name> --save-dev
```

Instalar paquetes

- Hay dos maneras de instalar paquetes npm: local o globalmente.
 - En ambos casos se usa la orden **npm install**

```
npm install <package_name>
```

- La instalación del paquete depende de la forma en que desea utilizar el paquete.
- Si se desea depender del paquete en el módulo que se está construyendo usando, por ejemplo **require** de Node.js, entonces se instalará localmente. Este es el comportamiento predeterminado de **npm install**.
- Si se desea usar el módulo como una herramienta de línea de comandos, por ejemplo el **CLI de grunt**, entonces es mejor instalarlo globalmente. Esto se logra añadiendo el parámetro **-g**

```
npm install -g <package_name>
```

Versión instalada

- Si no hay un archivo **package.json** en el directorio local, se instalará la versión más reciente del paquete.
- Si hay un archivo **package.json**, se instalará la última versión que cumpla la **regla semver** (<https://docs.npmjs.com/getting-started/semantic-versioning>) declarada en package.json para ese paquete (si existe).
- Una vez que el paquete está en **node_modules**, puede ser usado mediante **require**.



Ejemplo

- Creamos un archivo index.js que usa lodash

```
// index.js
var lodash = require('lodash');

var output = lodash.without([1, 2, 3], 1);
console.log(output);
```

- Si ejecutamos el archivo con la orden node index.js nos indicará que no existe, por tanto hay que instalarlo

```
npm install lodash
```



Otros ejemplos prácticos

```
# Grunt  
sudo npm install grunt-cli -g
```

```
# CoffeeScript  
sudo npm install coffee-script -g
```

```
# Stylus  
sudo npm install stylus -g
```

```
# Grunt and Grunt plugins  
npm install grunt --save-dev  
npm install grunt-contrib-coffee --save-dev  
npm install grunt-contrib-stylus --save-dev  
npm install grunt-contrib-watch --save-dev
```



Pongámoslo en práctica – Instalando Lite-server



Instala Lite-Server

- Abre una ventana de comando
- Instala lite-server:

```
npm install -g lite-server
```

Proyecto de prueba



- Crea un directorio con el nombre “prueba1”
- Crea un proyecto node: npm init -y
- Crea una página index.html con un texto “Hola Mundo”, en el proyecto
- Modifica el proyecto (package.json) para que en “start” lance lite-server

```
"scripts": {  
    "start": "lite-server"  
},
```

Iniciar y parar el servidor



- Lanza el proyecto con:
 - En la consola escribe: **npm start**
 - Se iniciará el servidor en la dirección <http://localhost:3000/>
 - Revisa la página en el navegador
- Modifica el proyecto
 - Modifica index.html para que diga “Hola Mundo!!”
 - Observa como se recarga el navegador con la nueva versión
- Para el proyecto con:
 - Ve a la consola y pulsa *CTRL+C* → Acepta la finalización pulsando “S”
→ El servidor parará
 - Verifica el navegador

2

Grunt

Grunt

- Durante nuestros flujos de trabajo de desarrollo web, hay muchas tareas que tenemos que repetir.
- Se trata de tareas como minificar archivos JavaScript y CSS, pruebas de unidades, linting de archivos para comprobar errores, compilación de archivos de preprocesador CSS (LESS, SASS) y mucho más.
- Grunt es un corredor de la tarea. Esto significa que las tareas repetitivas que tratamos en nuestro flujo de trabajo diario se automatizan.
- Será necesario instalar el cli de Grunt



```
npm install -g grunt-cli
```

Componentes

- Una instalación típica de **GruntJS** tiene muchos ficheros, pero debemos centrarnos en 2: **package.json** y **Gruntfile.js**
 - **package.json**: Nos permite guardar datos de nuestro proyecto para luego usarlos en el Gruntfile. Es típico guardar lasUrls a nuestras carpetas contenedoras de ficheros Coffee o Sass/Less.
 - **Gruntfile.js**: Puede tener extensión .js o .coffee. Es el fichero principal y en el que configuraremos nuestras tareas y abrir nuestros plugins.
- Este es un ejemplo de package.json:

```
{  
  "cssFolder": "path/to/sass/files",  
  "jsFolder": {  
    "folder1": "folder1"  
  }  
}
```

Visión del conjunto

- Esta será la estructura de archivos de nuestras muestras.

dist // Mantendrá todo fuera de los archivos finales (archivos minified)

 css

 js

src // Tendrá todos nuestros archivos originales

 css

 style.css

 pretty.less

 js

 magic.js

Gruntfile.js // Nuestra configuración Grunt

package.json //Nuestra configuración de paquetes npm (cómo extraer paquetes)

- Tenemos una carpeta **src** y una carpeta **dist**. Estaremos generando el código fuente en la carpeta **src** y Grunt minificará esos archivos y los guardará en las carpetas **dist**.
- Los archivos en **dist** son los usaremos para producción.

Paquetes Grunt que necesitamos

- Cuando usamos npm, definimos los paquetes que necesitamos en un archivo **package.json**. Vamos a ese archivo y agregamos los paquetes que necesitamos.

```
{  
  "name": "grunt-getting-started",  
  "version": "0.1.0",  
  "devDependencies": {  
    "grunt": "~0.4.4",  
    "grunt-contrib-jshint": "latest",  
    "jshint-stylish": "latest",  
    "grunt-contrib-uglify": "latest",  
    "grunt-contrib-less": "latest",  
    "grunt-contrib-cssmin": "latest",  
    "grunt-contrib-watch": "latest"  
  }  
}
```

- Se han definido el nombre de nuestro proyecto, la versión, y las dependencias de desarrollo.

Paquetes Grunt que necesitamos

- En la siguiente tabla se muestra la función de cada paquete grunt-contrib-* * * *

Pluggin	Descripción
contrib-jshint	Validar archivos utilizando jshint
contrib-uglify	Minifica archivos JS utilizando UglifyJS
contrib-watch	Ejecutar tareas siempre que se cambien los archivos vistos
contrib-clean	Limpiar archivos y carpetas
contrib-copy	Copiar archivos y carpetas
contrib-concat	Combinar archivos en un solo archivo
contrib-cssmin	Comprimir archivos CSS
contrib-less	Compila archivos LESS a CSS

- La lista completa de paquetes está disponible en el repositorio de plugins de Grunt (<https://gruntjs.com/plugins>).

Instalación de los paquetes

- Ahora que tenemos los paquetes que necesitamos definidos, vamos a instalarlos.
- Con nuestro archivo package.json listo, entramos en la línea de comandos y escribimos:

\$ npm install
- Npm descargará los paquetes en una carpeta node_modules. Una vez finalizada la descarga, estarán listos para usarlos en nuestro proyecto.

Gruntfile

- Para definir nuestra configuración para Grunt, utilizaremos nuestro archivo **Gruntfile.js**.
- En nuestro Gruntfile.js, vamos a añadir los elementos básicos que necesitaremos (ver siguiente diapositiva).

Gruntfile.js

```
// Gruntfile.js
// Nuestra función de envoltura (requerida por grunt y sus complementos)
// Toda la configuración entra module.exports = function(grunt) {
    // CONFIGURAR GRUNT
    grunt.initConfig({
        // Obtener la información de configuración de package.json
        // De esta manera podemos usar cosas como nombre y versión (pkg.name)
        pkg: grunt.file.readJSON('package.json'),
        // Toda nuestra configuración irá aquí
    });

    //Carga de pluggins GRUNT
    // Sólo podemos cargarlos si están en nuestro paquete.json
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-less');
    grunt.loadNpmTasks('grunt-contrib-cssmin');
    grunt.loadNpmTasks('grunt-contrib-watch');
};

dentro de esta función
```

Configuración de paquetes

- Con la configuración básica de Grunt lista, vamos a configurar uno de los paquetes.
- Comencemos con el paquete de JSHint para validar nuestros archivos JavaScript y decirnos si hay algún error.
- Esta es la forma para que Grunt sabe qué archivos queremos que hagan lint, minify o cualquier otra acción.

```
// Gruntfile.js
// ....
grunt.initConfig({
    // Configurar jshint para validar archivos js
    jshint: {
        // Utiliza jshint-stylish para que nuestros errores aparezcan y se lean bien
        options: { reporter: require('jshint-stylish') },
        build: ['Gruntfile.js', 'src/**/*.{js}']
    }
});
```

Configuración de paquetes

El formato básico para configurar nuestros paquetes será el siguiente:

- Invocar el **paquete** (ej. *jshint*)
- Establecer **opciones** si es necesario.
 - Las opciones se encuentran definidas normalmente en los documentos para cada paquete específico
- Crear un atributo **build** y pasar archivos, directorios o cualquier otro elemento que deseemos que se procese.

Ejecución

- Para ejecutar grunt simplemente hay que llamar a grunt e indicar la configuración/tarea a ser ejecutada

```
$ grunt <config>
```



Ejecutando tareas comunes con Grunt

Linting de código

- Vamos a probar la configuración anterior generando un código de prueba.

```
// Gruntfile.js
grunt.initConfig({
    // Configurar jshint para validar archivos js
    jshint: {
        // Utiliza jshint-stylish para que nuestros errores aparezcan y se lean bien
        options: { reporter: require('jshint-stylish') },
        build: ['Gruntfile.js', 'src/**/*.{js}']
    }
});
```

- Creamos el archivo a **src/js/magic.js**

```
var hello = 'look im grunting!'
var awesome = 'yes it is awesome!'
```

- Ejecutamos grunt

```
$ grunt jshint
```

Minificación archivos JavaScript



- Vamos a añadir el paquete uglify, configurarlo y decirle qué archivos usar y crear.

```
grunt.initConfig({  
  pkg: grunt.file.readJSON('package.json'),  
  ...  
  // Configure uglify para minificar archivos js  
  uglify: {  
    options: { banner: '/*\n <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> \n*/\n' },  
    build: { files: { 'dist/js/magic.min.js': 'src/js/magic.js' } }  
  }  
});
```

- Aquí estamos configurando una opción llamada banner. Esto agregará un comentario al principio de nuestro archivo minificado.
- En nuestra compilación, estamos definiendo el archivo que queremos crear (**dist/js/magic.min.js**) a partir del src (**src/js/magic.js**).

Minificación archivos JavaScript



- Para probar usaremos un archivo realmente grande y minificarlo: **jQuery**.
- Copia todo el archivo jQuery y pégalo en el archivo **src/js/magic.js**.
- Para ejecutar, en la consola escribe:

```
$ grunt uglify
```

- Como resultado hemos reducido un archivo original de 360kb a 97.1kb

Compilación LESS a CSS



- Aunque usaremos LESS para este ejemplo, también puede usar el paquete *grunt-contrib-compass* para hacer lo mismo para SASS.
- Esta es nuestra configuración para compilar archivos LESS.
- Al igual que nuestro ejemplo JS minifying, utilizaremos nuestra configuración para definir los archivos de origen y salida.

```
// Gruntfile.js
grunt.initConfig({
  ...
  // Compilar less en hojas de estilo css
  less: { build: { files: { 'dist/css/pretty.css': 'src/css/pretty.less' } } }
});
});
```

- Para este ejemplo, no vamos a establecer ninguna opción. Basta con compilar nuestro src/css/pretty.less a un archivo dist/css/pretty.css.



Compilación LESS a CSS

- Añadimos algunos less en nuestro archivo de origen.

```
/* src/css/pretty.less */  
@red : #CC594A;  
@yellow : #B8CC24;  
@blue : #8BC5FF;  
@purple : #6F3596;  
body {  
background:@red;  
color:@yellow; }  
button { background:@blue; }  
div { background:@purple; }
```

- Ahora hemos definido algunas variables LESS y las hemos aplicado a nuestros elementos.
- Ejecutar la tarea:

\$ grunt less



Compilación LESS a CSS

- Se habrá creado el archivo **dist/css/pretty.css**.

```
/* dist/css/pretty.css */
body {
background: #cc594a;
color: #b8cc24; }
button {
background: #8bc5ff; }
div {
background: #6f3596; }
```

Ejecución de varias tareas a la vez

- Con Grunt, puede crear tareas que ejecuten múltiples tareas al mismo tiempo. Por ejemplo, si queremos lanzar todas nuestras tareas anteriores simplemente con una llamada.
- Cuando ejecutas grunt desde la línea de comandos, Grunt buscará una tarea llamada **default**. En ella podemos referenciar otras tareas.

```
// Gruntfile.js
...
// Creacion de tarea por defecto
grunt.registerTask('default', ['jshint', 'uglify', 'cssmin', 'less']);
```

- Ejecutamos grunt:

```
$ grunt
```
- Y todas las tareas en nuestra tarea default se ejecutarán.

Diferentes tareas para diferentes entornos

- Si queremos desarrollar en un **entorno de desarrollo** y luego cuando vamos a la **producción**, queremos que se ejecuten tareas diferentes.
- En Grunt podemos definir múltiples tareas dentro de cada configuración. Indicando el entorno
 - **dev**: entorno de desarrollo
 - **production**: entorno de producción

Diferentes tareas para diferentes entornos

- En la siguiente configuración de grunt se plantean tareas distintas para desarrollo y producción

```
// Gruntfile.js
grunt.initConfig({
    // Obtener la información de configuración de package.json
    // De esta manera podemos usar cosas como nombre y versión (pkg.name)
    pkg: grunt.file.readJSON('package.json'),
    ...
    // Configure uglify para minificar archivos js
    uglify: { options: { banner: '\n <%= pkg.name %> <%=
        grunt.template.today("yyyy-mm-dd") %> \n\n' } },
    dev: { files: { 'dist/js/magic.min.js': ['src/js/magic.js', 'src/js/magic2.js'] } },
    production: { files: { 'dist/js/magic.min.js': 'src/**/*js' } } }
});
```

Diferentes tareas para diferentes entornos

- › Ahora podemos indicar diferentes las tareas según el entorno de salida.

```
// Gruntfile.js
// Creación de tareas
// Esta tarea predeterminada pasará por toda la configuración (dev y producción) en cada tarea
grunt.registerTask('default', ['jshint', 'uglify', 'cssmin', 'less']);
// Esta tarea sólo ejecutará la configuración de dev
grunt.registerTask('dev', ['jshint:dev', 'uglify:dev', 'cssmin:dev', 'less:dev']);
// Sólo ejecuta la configuración de producción
grunt.registerTask('production', ['jshint:production', 'uglify:production', 'cssmin:production',
'less:production']);
```

- › Llamar al desarrollo ejecutando:

\$ grunt dev

- › Llamar a producción ejecutando:

\$ grunt production

3

Gulp

Gulp

- Como desarrolladores a menudo necesitamos ver las herramientas que usamos y decidir si estamos utilizando la herramienta adecuada para el trabajo.
- Gulp es un sistema de generación de flujos, mediante el uso de secuencias de nodo, la manipulación de archivos se realiza en memoria y un archivo no se escribe hasta que se lo indique.
- Al igual que Grunt, Gulp es un corredor de tareas javascript.
- Gulp prefiere sin embargo el código sobre la configuración. Siendo que sus tareas están escritas en código, Gulp se siente más como un framework de construcción, dándole las herramientas para crear tareas que se adapten a sus necesidades específicas.



Instalando Gulp

Instalación de Gulp



- Gulp es fácil de instalar y ejecutar usando npm
- Para instalarlo globalmente.

```
$ npm install --global gulp
```

- Después de esto, necesitaremos gulp como un devDependencies en cualquiera de los proyectos en los que se quiera usar.
 - Es necesario terne configurado tu package.json.
- Finalmente necesitaremos el archivo de configuración **gulpfile.js** en la raíz del proyecto. Este contiene las tareas a ejecutar.
 - Como paso intermedio, agregaremos el complemento **gulp utilites** para que tengamos una tarea ejecutable.

```
$ npm install --save-dev gulp-util
```

Instalación de Gulp



- En el archivo **gulpfile.js** añadiremos un simple log que registra que gulp se está ejecutando.

```
// grab our gulp packages
var gulp = require('gulp'), gutil = require('gulp-util');
// create a default task and just log a message
gulp.task('default', function() {
    return gutil.log('Gulp is running!')
});
```

- Ejecutamos con la orden **gulp** en la línea de comandos:

```
>gulp
[12:32:08] Using gulpfile ~/Projects/gulp-scotch-io/gulpfile.js
[12:32:08] Starting 'default'...
[12:32:08] Gulp is running!
[12:32:08] Finished 'default' after 1 ms
```

Estructura de directorios

- La composicion básica de la estructura de directorios en Gulp es muy simple:
 - **source** es la carpeta donde haremos nuestro trabajo con el código fuente.
 - **assets** será creada cuando llamemos a gulp y será el directorio de distribución.
 - El archivo **bundle.js** se creará por gulp cuando minifique y combine todos nuestros archivos JS.
 - **assets/style.css** será creado por gulp cuando procesemos y combinamos nuestros archivos SASS en source/scss.

```
public/
|   assets/
|   |   stylesheets/
|   |   |   style.css
|   |   javascript/
|   |   |   vendor/
|   |   |   |   jquery.min.js
|   |   |   bundle.js
source/
|   javascript/
|   |   courage.js
|   |   wisdom.js
|   |   power.js
|   scss/
|   |   styles.scss
|   |   grid.scss
gulpfile.js
packages.json
```



Ejemplo de configuración

Jshint para guardar



- Nuestra primera tarea será lint en nuestro javascript (comprobar los errores) usando jshint
- Configuraremos gulp para ejecutar esta tarea cada vez que guarde un archivo javascript.
- Para comenzar necesitaremos el paquete **gulp-jshint**. También necesitaremos un **reporter** para jshint para formatear la salida.
-
- Vamos a instalar esos componentes con npm.

```
$ npm install --save-dev gulp-jshint jshint-stylish
```

Configuración de tareas

- Agregamos la tarea lint a nuestro gulpfile.

```
/* File: gulpfile.js */
// grab our packages
var gulp  = require('gulp'),
    jshint = require('gulp-jshint');

// define the default task and add the watch task to it
gulp.task('default', ['watch']);

// configure the jshint task
gulp.task('jshint', function() {
  return gulp.src('source/javascript/**/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('jshint-stylish'));
});

// configure which files to watch and what tasks to use on file changes
gulp.task('watch', function() {
  gulp.watch('source/javascript/**/*.js', ['jshint']);
});
```



Configuración de tareas



- Hemos reescrito nuestra tarea **default** para que la tarea **watch** sea una dependencia. Esto hará que la ejecución **gulp** ejecutará dicha tarea.
- La nueva tarea **jshint** procesa todos los archivos .js que existen en *source/javascript* o en cualquiera de sus subdirectorios.
- Estos archivos se pasan al **plugin gulp-jshint**, que luego pasa al **reporter** para darnos los resultados jshint.
- Podemos ejecutar la tarea con: `$ gulp jshint`
- Ahora si se detecta un cambio en cualquiera de nuestros archivos javascript, se ejecutará la tarea jshint.



Compilando Sass

Compilación de Sass con libsass



- Sass sirve como una forma de extender CSS dando soporte para variables, reglas anidadas, mixins, importaciones en línea, y más.
- Para la compilación sass usaremos **gulp-sass**
 - NOTA:
 - gulp-sass utiliza node-sass que a su vez utiliza libsass. En Windows necesitará instalar python 2.7.x y Visual Studio Express 2013 para compilar libsass. Mac y Linux utilizarán cualquier gcc disponible.
 - Una alternativa es usar *gulp-ruby-sass*, que utiliza rubí y la gema sass en su lugar.



Archivo de configuración

```
/* file: gulpfile.js */
var gulp = require('gulp'), jshint = require('gulp-jshint'), sass = require('gulp-sass');

/* tarea Jshint estaría aquí */

/* tarea SASS*/
gulp.task('build-css', function() {
    return gulp.src('source/scss/**/*.{scss}')
        .pipe(sass())
        .pipe(gulp.dest('public/assets/stylesheets'));
});

/* Tarea watch actualizada para incluir sass */
gulp.task('watch', function() {
    gulp.watch('source/javascript/**/*.{js}', ['jshint']);
    gulp.watch('source/scss/**/*.{scss}', ['build-css']);
});
```

Compilación de Sass con libsass



- También podemos agregar **sourcemaps** usando **gulp-sourcemaps**.
- Sourcemaps permite mapear archivos procesados, minificado u otros archivos modificados, a las fuentes originales.

```
/* file: gulpfile.js */
var gulp      = require('gulp'),
    jshint    = require('gulp-jshint'),
    sass      = require('gulp-sass'),
    sourcemaps = require('gulp-sourcemaps');

gulp.task('build-css', function() {
  return gulp.src('source/scss/**/*.{scss}')
    .pipe(sourcemaps.init()) // Process the original sources
    .pipe(sass())
    .pipe(sourcemaps.write()) // Add the map to modified source.
    .pipe(gulp.dest('public/assets/stylesheets'));
});
```

Javascript concat y minify



- Al trabajar con un montón de javascript, por lo general llega un momento de unirlo todo.
- El plugin de propósito general **gulp-concat** permite lograr esto fácilmente.
- También podemos ir un paso más allá y ejecutarlo a través de **uglify** y obtener un tamaño de archivo mucho menor.

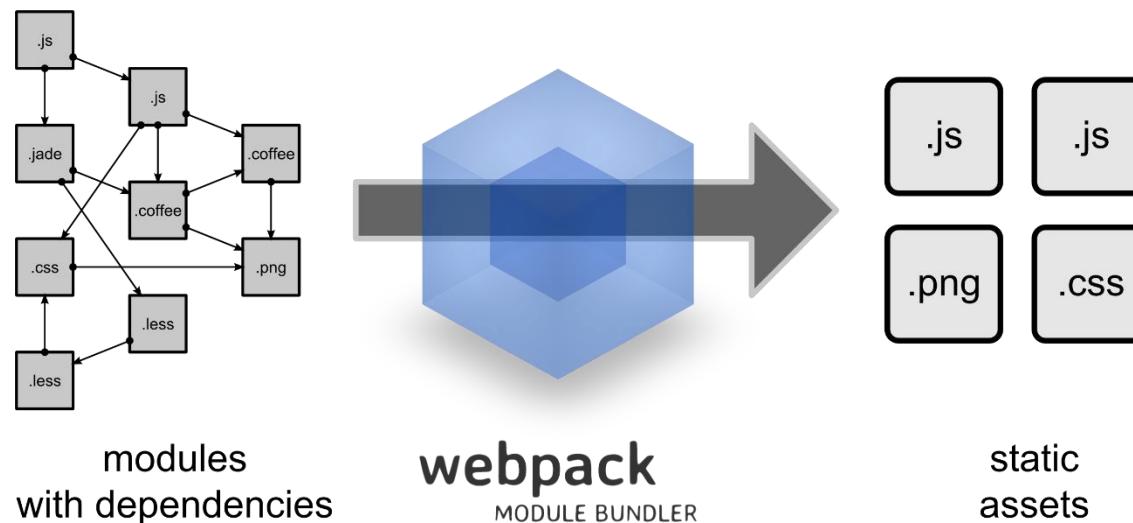
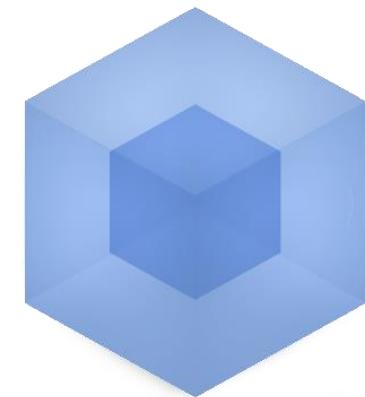
```
gulp.task('build-js', function() {
  return gulp.src('source/javascript/**/*.js')
    .pipe(sourcemaps.init())
    .pipe(concat('bundle.js'))
    //only uglify if gulp is ran with '--type production'
    .pipe(gulpif(util.env.type === 'production', uglify(), util.noop()))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('public/assets/javascript'));
});
```

3

Webpack

Webpack

- Webpack es la última y mejor en las herramientas de desarrollo front-end.
- Es un empaquetador de módulos que funciona muy bien con los workflows front-end más modernos como Babel, ReactJS, CommonJS, entre otros.
- <https://webpack.github.io/>





Instalando y usando Webpack



Proyecto ejemplo

- Crea un directorio de proyecto con dos archivos
 - app.js

```
document.write('welcome to my app');  
console.log('app loaded');
```

- index.html

```
<html>  
  <body>  
    <script src="bundle.js"></script>  
  </body>  
</html>
```

Instala webpack



- Usaremos npm para instalar webpack de manera global

```
npm install webpack -g
```

- Esto se hará solo una vez



Empaqueta

- Abre la consola y ejecuta

```
webpack ./app.js bundle.js
```

Configuración

- Un archivo de configuración en Webpack es básicamente un módulo **common.js**.
- En él se pone toda su configuración, loaders y otra información específica relacionada con su compilación.
- En su directorio raíz, creamos un archivo llamado **webpack.config.js** y agreguamos el código siguiente:

```
module.exports = {  
  entry: "./app.js",  
  output: {  
    filename: "bundle.js"  
  }  
}
```

- **entry** - nombre del archivo de nivel superior o conjunto de archivos que queremos incluir en nuestra compilación, puede ser un solo archivo o una matriz de archivos.
- **output** - un objeto que contiene la configuración de salida. En nuestra compilación, sólo especificamos el nombre de archivo (bundle.js) que queremos que Webpack construya.

- Para ejecutar la configuración, escribimos en consola: **webpack**

Watchmode

- Con watchmode, Webpack observará los archivos y cuando uno de ellos cambie, inmediatamente reiniciará la compilación y recreará su archivo de salida.
- Hay 2 maneras de habilitar watchmode
 - Desde la línea de comando `webpack --watch`
 - Añadir una clave **watch** con valor true
 - Luego ejecutar webpack

```
module.exports = {  
  entry: "./app.js",  
  output: {  
    filename: "bundle.js"  
  },  
  watch: true  
}
```

`webpack`

Webpack Dev Server

- Webpack tiene un servidor web llamado webpack-dev-server.
- Desde la línea de comandos, instalamos webpack-dev-server globalmente:

```
Npm instalar webpack-dev-server -g
```

- Para inicializar el servidor:

```
webpack-dev-server
```

- El servidor sirve en el puerto 8080

```
http://localhost:8080/webpack-dev-server/
```

- Otra característica, es que se ejecuta en modo watch, es decir si hay cambios en la aplicación, refrescará el navegador automáticamente

Compilando múltiples archivos

- Será necesario invocar un módulo desde otro, mediante **require**
`require('./logger');`
- Otra manera es añadir en el archivo de configuración los archivos extra en la clave **entry**

```
....  
entry: ['./global.js', './app.js'],  
....
```




Compilando varios archivos



Añade nuevos archivos y requerirlos

- Crea el archivo **logger.js** con el siguiente código

```
console.log('logger.js is now loaded...');
```

- Invoca el módulo desde app.js

```
require('./logger');
```

- Ejecuta el server



Añade un nuevo archivo sin requerir

- Crea el archivo **global.js** con el siguiente código

```
console.log('global.js is now loaded...');
```

- Modifica las entradas del archivo de configuración

```
module.exports = {
  entry: ["./global.js", "./app.js"],
  output: {
    filename: "bundle.js"
  }
}
```

- Ejecuta el server

Loaders y preloaders de Webpack

- Los loaders permiten preprocesar archivos según se requiera **[requiere()]** o se "carguen".
- Son el equivalente a las "**tareas**" de otras herramientas de compilación, y proporcionan una manera potente de gestionar los pasos de construcción del frontend.
- Pueden transformar archivos de un lenguaje (ej. CoffeeScript) a JavaScript o imágenes en línea a URL de datos.
- Incluso permiten hacer **require()** de archivos css desde el JavaScript



Instalando el loader de babel

Instalación de loaders



- A continuación, instalaremos y utilizaremos dos cargadores: Babel y JSHint.
- Antes de comenzar a usar estos cargadores, instalaremos algunos módulos npm.
 - Primero, usar npm init (responda sí a todas las preguntas) para crear un archivo package.json.
 - A continuación, en el terminal, escribe:

```
npm install babel-core babel-loader jshint jshint-loader node-libs-browser babel-preset-es2015 babel-preset-react webpack --save-dev
```

Instalación de loaders



- **babel-core** es el paquete babel npm
- **babel-loader** es el cargador de módulos babel para Webpack
- **jshint** es una herramienta que ayuda a detectar errores y posibles problemas en su código JavaScript
- **jshint-loader** es el módulo jshint loader para Webpack
- **Node-libs-browser** es una dependencia de igual a Webpack. Proporciona ciertas bibliotecas de nodos para el uso del navegador.
- **babel-preset-es2015** es un preset babel para todos los plugins es2015.
- **babel-preset-react** es un preset de babel para todos los plugins de React.
- – **save-dev** guarda losmódulos como dependencias de desarrollo

Usar Babel con Webpack



- Una vez instalado babel podemos generar código ES6
- Modifica el archivo logger.js

```
let checkName= (firstName, lastName) => {  
  if(firstName !== 'nader' || lastName !== 'dabit') {  
    console.log('You are not Nader Dabit');  
  } else {  
    console.log('You are Nader Dabit');  
  }  
}  
  
checkName('nader', 'jackson');
```



Añade los loaders a la config de Kebpack

- Abre **webpack.config.js** y agregua el loader babel.
- Para añadir un cargador en Webpack, primero debe crear una clave denominada **module**, será un objeto.
- En el objeto de módulo, crea una clave denominada **loaders**, que es una matriz.
- En la matriz, se crea un objeto por cada loader. En el ejemplo hemos añadido el loader **“babel-loader”**

```
module.exports = {  
  entry: ["./global.js" , "./app.js"],  
  output: {  
    filename: "bundle.js"  
  },  
  module: {  
    loaders: [  
      {  
        test: /\.es6$/,  
        exclude: /node_modules/,  
        loader: 'babel-loader',  
        query: {  
          presets: ['react', 'es2015']  
        }  
      }  
    ]  
  },  
  resolve: {  
    extensions: [", '.js', '.es6']  
  },  
}
```



- **test** - una expresión regular que prueba qué tipo de archivos se ejecutan a través de este cargador. Como puede ver, hemos agregado un regex para probar todos los archivos con una extensión es6.
- **exclude** - los archivos que el cargador debe excluir/ignorar. Hemos añadido la carpeta *node_modules*.
- **loader** - el nombre del cargador que vamos a utilizar (**babel-loader**).
- **query**: puede pasar opciones al cargador escribiéndolas como una cadena de consulta o utilizando la propiedad de consulta como hemos hecho anteriormente.
- **cacheDirectory** - Predeterminado false. Cuando se establece, el directorio dado se utilizará para almacenar en caché los resultados del cargador. Futuras compilaciones de webpack intentarán leer desde el caché para evitar la necesidad de ejecutar el potencialmente costoso proceso de recompilación de Babel en cada ejecución
- **presets** - nos permite usar los preajustes `react` y `es2015` que se instalaron anteriormente



Ejecutar el proyecto

- Cierra y relanza el server de webpack

Preloaders

- Como se puede suponer, los **preloaders** se ejecutan antes de que los cargadores lo hagan.
- Por ejemplo si queremos usar JSHint como preloader, añadiremos la siguiente configuración a webpack.config.js.
- La ejecución se realiza de la misma manera que los loaders

```
module.exports = {  
  entry: ["./global.js" , "./app.js"],  
  output: {  
    filename: "bundle.js"  
  },  
  module: {  
    preLoaders: [  
      {  
        test: /\.js$/,  
        exclude: /node_modules/,  
        loader: 'jshint-loader'  
      }  
    ],  
    loaders: [  
      {  
        test: /\.es6$/,  
        exclude: /node_modules/,  
        loader: 'babel-loader',  
        query: {  
          cacheDirectory: true,  
          presets: ['react', 'es2015']  
        }  
      }  
    ]  
  },  
  resolve: {  
    extensions: ['', '.js', '.es6']  
  }  
}
```

Crea un script de inicio

- Esto es en realidad una característica de npm y de la funcionalidad que puede utilizarse en su archivo package.json.
- Se puede usar para facilitar su proceso de inicio del empaquetamiento.
- En package.json, hay una clave llamada **scripts**. Elimina el contenido de la clave de secuencias de comandos y reemplácelo con lo siguiente:

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

- Ahora cuando necesites ejecutar webpack-dev-server, se puede ejecutar lo siguiente en su lugar:

```
npm start
```

Builds de desarrollo y producción

- Para ejecutar un paquete de producción, dejar que webpack minimice su código, ejecute Webpack con un flag **-p**:

```
webpack -p
```

- Esto producirá un paquete minificado.
- Para trabajar con diferentes archivos de configuración para producción vs desarrollo es necesario instalar **strip-loader**

```
npm install strip-loader --save-dev
```

- Strip-loader es un loader para quitar funciones arbitrarias del código de producción

Builds de desarrollo y producción

- Para la configuración de producción podemos crear un archivo específico **webpack-production.config.js**

```
var WebpackStripLoader = require('strip-loader');
var devConfig = require('./webpack.config.js');
var stripLoader = {
  test: [/\.\js$/, /\.es6$/],
  exclude: /node_modules/,
  loader: WebpackStripLoader.loader('console.log')
}
devConfig.module.loaders.push(stripLoader);
module.exports = devConfig;
```

- **WebpackStripLoader** - necesitamos el módulo npm del cargador de tiras
- **devConfig** - necesitamos el archivo de configuración del paquete web original
- **var stripLoader** - creamos un nuevo objeto y pasamos las claves de prueba y exclusión como se ha visto antes.

- › Cabe destacar:

```
loader: WebpackStripLoader.loader ('console.log')
```
- › Aquí pedimos a strip-loader que elimine cualquier sentencia console.log de nuestra compilación.
- › **devConfig.module.loaders.push** (stripLoader); - empujamos el nuevo objeto en nuestra matriz de cargadores de nuestra configuración original.
- › **module.exports = devConfig;** - exportamos nuestro nuevo objeto de configuración

Ejecutar

- Acceder al terminal y escribir:

```
webpack --config webpack-production.config.js -p
```

- Esto ejecuta webpack con el indicador de configuración, permitiéndonos especificar un archivo de configuración personalizado.
- **-p** minimiza el código para producción.
- Abre bundle.js. Si realiza una búsqueda, se verá que no hay instrucciones *console.log* en nuestro paquete.



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es



GOBIERNO
DE ESPAÑA

MINISTERIO
DE ENERGÍA, TURISMO
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE
EMPRENDIMIENTO Y
EMPLEO JUVENIL
garantía juvenil



Agenda Digital para España



UNIÓN EUROPEA

Fondo Social Europeo
"El FSE invierte en tu futuro"