Bloque 4

Programación con JavaScript

Node Package Manager. Webpack.

Programación con JavaScript
Cefire 2017/2018
Autores: Alejandro Amat Reina
Arturo Bernal Mayordomo

Índice

Introducción

Podemos decir que NPM (Node Package Manager) es una herramienta que facilita reutilizar código de otros programadores en nuestros proyectos. Este código está distribuido en paquetes (**packages**) o módulos. Podemos encontrar algunas de las librerías más populares, frameworks y herramientas como jQuery, Angular, Express, JSHint, ESLint, Sass, Browserify, Bootstrap, Cordova, Ionic, entre otras.

Además, NPM nos proporciona un completo sistema de automatización de tareas basado en scripts. Para la mayoría de los casos, NPM es suficiente, siendo innecesario el uso de otras herramientas como Gulp, Grunt o Bower, aunque muchos desarrolladores las combinan en sus proyectos, pero con la desventaja de que necesitas aprender cómo funcionan 2, 3 o 4 herramientas diferentes.

Instalación de NPM

Para instalar NPM, vamos a necesitar instalar primero Node.js (NPM es parte de Node.js). Node permite la ejecución de aplicaciones JavaScript en el servidor usando el motor v8 (Chrome), permitiendo la creación de aplicaciones tanto de cliente como de servidor con un único lenguaje → JavaScript.

Podemos consultar las instrucciones de cómo instalarlo en Windows y Mac aquí. Si usamos Linux, hay varias distribuciones que incluyen Node en sus repositorios. Si queremos instalar la última versión disponible (añadiendo un repositorio externo), podemos seguir las instrucciones que hay aquí. La versión recomendable para instalar es la última 8.x, porque es la actual versión estable, con soporte desde octubre de 2017 hasta diciembre de 2019.

Una vez que tengamos Node instalado, podemos comprobar si lo tenemos activado desde la línea de comando ejecutando:

```
arturo@arturo-lenovo:~$ node -v
v6.13.1
arturo@arturo-lenovo:~$ npm -v
5.8.0
```

Además, podemos actualizar NPM a la última versión disponible utilizando el comando **npm install npm -g** (en Linux ejecútalo como root o con sudo).

Paquetes NPM

Un paquete (librería, framework, tool, …), es un directorio que contiene varios archivos, incluyendo un archivo **package.json**, que contiene información sobre el autor, versión, otros paquetes de los que a su vez depende este paquete, etc. En esta sección vamos a ver cómo instalar, actualizar, eliminar, etc, paquetes en nuestro proyecto (o de forma global en nuestro sistema).

Podemos consultar la ayuda de **npm** mediante los comandos:

- **npm -h** → Muestra la ayuda rápida y una lista con los comandos típicos de npm
- **npm comando -h** → Muestra una ayuda rápida con información específica sobre el comando npm (npm install -h).
- **npm help comando** → Se abre una página en el navegador o el man de Linux, mostrando ayuda sobre un determinado comando
- **npm help-search palabras** → Nos devuelve una lista de ayuda de aquellos que contiene la palabra(s) buscada(s) (separadas por espacios)

Creando nuestro package.json

Hay dos tipos de proyectos que podemos crear, una página o aplicación web que ejecutarán los usuarios, o una librería/herramienta para ser incluida en proyectos de otros desarrolladores. Ambos tendrán dependencias de otros paquetes y también necesitan ser probadas, *minificadas*, etc..

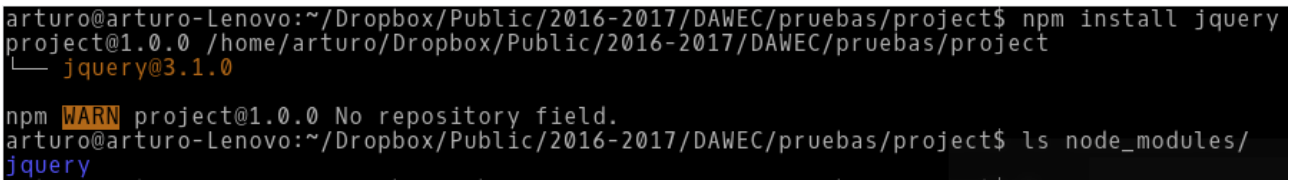
Para crear un archivo **package.json** en nuestro proyecto, simplemente nos moveremos al directorio principal de nuestro proyecto (no hace falta que esté vacío) y ejecutamos **npm init**. Nos preguntará algunas cosas sobre nuestro proyecto y creará el archivo **package.json** basado en eso. Podemos dejar muchos de los valores por defecto o vacíos presionando enter. Justo antes de crear el archivo, nos mostrará el contenido que tendrá.

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "A project created to learn NPM",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Arturo",
  "license": "ISC"
}
```

Instalando paquetes

El comando para instalar un paquete en nuestro proyecto es **npm install nombre-paquete**. Si ejecutamos este comando por primera vez, veremos que creará un directorio llamado **node_modules** dentro del directorio de proyecto. Este directorio contendrá todos los paquetes instalados mediante npm (incluyendo a su vez

las dependencias de dichos paquetes). Vamos a probar con jQuery.



Como vemos, instalará jQuery (la última versión) dentro del directorio node_modules (el archivo que enlazaremos en nuestro proyecto estará en jquery/dist/).

Incluyendo las dependencias en package.json

Para incluir una dependencia en nuestro archivo package.json, usamos la opción **--save** o **-S** junto al comando de instalación (**i** → alias de *install*). Nota: desde la versión 5 de NPM se puede omitir esta opción ya que lo hace por defecto.

npm i jquery -S

```
{
    "name": "project",
    "version": "1.0.0",
    "description": "Proyecto creado para aprender NPM",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Arturo",
    "license": "ISC",
    "dependencies": {
        "jquery": "^3.1.0"
    }
}
```

Esto nos asegura que NPM instalará o actualizará a la última versión “3.x.x” de la librería jQuery en nuestro proyecto. Por defecto, será considerado como una **dependencia de producción** (nuestra aplicación la necesita para ejecutarse). Hay otros paquetes que no se necesitan al ejecutar la aplicación, sino por otros motivos, como pruebas (ejemplo: karma), o minificar/ofuscar código (ejemplo: uglify). Estas se conocen como **dependencias de desarrollo** y normalmente no están incluidas (se pueden desinstalar) cuando nuestra aplicación está en producción.

Para incluir un paquete en nuestra lista de dependencias de desarrollo usaremos la opción –**save-dev** o **-D**. Por ejemplo:

npm i uglify --save-dev

```
{
    "name": "project",
    "version": "1.0.0",
    "description": "Proyecto creado para aprender NPM",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Arturo",
```

```
    "license": "ISC",
    "dependencies": {
        "jquery": "^3.1.0"
    },
    "devDependencies": {
        "uglify": "^0.1.5"
    }
}
```

Instalando las dependencias existentes del proyecto

Si descargas un proyecto existente con un archivo package.json (o clonamos un repositorio GIT), normalmente no tendrá las dependencias instaladas (ocupan mucho). En resumen, el directorio **node_modules** se añade a **.gitignore** para ser ignorado por GIT. Para volver a instalar las dependencias tanto para producción como para desarrollo que aparecen en package.json ejecutamos **npm install** (o **npm i**).

Podemos instalar solo las dependencias de producción o solo las dependencias de desarrollo usando **npm install --only=prod**, o **npm install --only=dev**.

Instalando paquetes globales

Hay herramientas de JavaScript como **handlebars, gulp, grunt, etc**. que pueden ser instaladas como globales en el sistema operativo, y de esta forma, usarlas en cualquier directorio o proyecto.

Para instalar una dependencia de forma global, lo que haremos será incluir en el comando de la instalación la opción **-g** al final del comando. Hay que tener en cuenta que debemos ser administradores (root o sudo en Linux). Por ejemplo:

npm i handlebars -g → Instala el comando handlebars de forma global

```
arturo@arturo-sobremesa:~$ handlebars -v
4.0.5
```

Listando los paquetes instalados

Para poder ver los paquetes instalados en el directorio npm_modules escribiremos **npm list**. Nos mostrará mediante una vista de árbol, los paquetes que hay instalados y las dependencias que tiene cada uno de ellos. Para ver sólo los paquetes que hemos instalado nosotros (sin las dependencias de estos paquetes), ejecutamos **npm list --depth=0**.

```
arturo@arturo-sobremesa:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm list --depth=0
project@1.0.0 /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
├── jquery@3.1.0
└── uglify@0.1.5
```

Si queremos mostrar un listado de los paquetes globales del sistema, necesitamos añadir la opción **--global=true** al comando.

```
arturo@arturo-sobremesa:~$ npm list --global=true --depth 0
/usr/lib
├── docker@0.2.14
├── handlebars@4.0.5
└── npm@3.10.6
```

Otra opción bastante útil es incluir **--dev=true** (lista sólo los paquetes en

desarrollo), **--prod=true** (solo los paquetes en producción), o **--long=true** (muestra una descripción de cada uno).

```
arturo@arturo-sobremesa:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm list --long=true --
depth=0
project@1.0.0
| /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
| A project created to learn NPM
├── jquery@3.1.0
|   JavaScript library for DOM operations
|   git+https://github.com/jquery/jquery.git
|   https://jquery.com
└── uglify@0.1.5
    A simple tool to uglify javascript & css files
    git://github.com/nanjingboy/uglify.git
    https://github.com/nanjingboy/uglify
```

Instalando una versión específica de un paquete

Por defecto, cuando instalamos un paquete, se instalará la última versión estable. Pero a veces, nuestro proyecto necesita incluir una versión anterior (por ejemplo, si queremos soportar IExplorer 8, necesitamos la versión de jQuery 1.x).

Las versiones de los paquetes normalmente tienen tres números **X.Y.Z**. La mayoría de paquetes siguen las reglas **SemVer** (Semantic Versioning) para aplicar la numeración. Son estas:

- Cuando la **Z** es incrementada, significa que un bug o problema ha sido solucionado, pero no se añade ninguna funcionalidad nueva.

- Cuando la **Y** es incrementada, significa que nuevas funcionalidades han sido añadidas, pero que esto no afecta al código de las versiones previas (siempre que se mantenga la **X**). Por ejemplo, una aplicación hecha con AngularJS 1.2 debería seguir funcionando con AngularJS 1.5 (pero al revés no tiene por qué).

- El número **X** se incrementará sólo si han habido suficientes cambios de forma que no se garantiza que las versiones anteriores (con la **X** menor) funcionen o sean compatibles con la nueva. Por tanto debemos tener cuidado cuando hagamos una migración y vigilar los cambios que se han producido en la librería o framework. Por ejemplo, la versión jQuery 2.x no soporta IE8 ni versiones anteriores, sin embargo la versión 1.x sí.

Si queremos instalar una versión específica de una librería en lugar de la última versión, lo que debemos hacer es escribir paquete@x.y.z en lugar de sólo el nombre.

npm i jquery@"1.12.3" → Instalará esta versión de jQuery que es todavía compatible con IE8. Si añadimos la opción **--save**, nos la guardará en el archivo **package.json** y **nunca** será actualizada a menos que cambiemos nosotros la versión.

También podemos especificar que queremos instalar una versión previa a x.y.z con: **pakage@"<x.y.z"**.

Otras posibilidades (ejemplos):

- **"*" ó "x"** → Instala la última versión de un paquete (cuidado con los cambios que pueda tener esa versión y cómo nos puede afectar…).

- **"3" ó "3.x", ó "3.x.x"** → Esto instala la última versión de un paquete siempre que sea 3.x.x (no se actualizará a la versión 4.x.x).

- **"^3.3.5"** → Instalará la última versión 3 (3.x.x) de un paquete, pero como mínimo deberá ser la versión 3.3.5 (el carácter **^** significa que sólo el primer número, 3, debe respetarse). Este es el comportamiento por defecto de NPM

- **"3.3" ó "3.3.x"** → Instalará la última versión 3.3.x de un paquete (nunca se actualizará a 3.4.x o posterior).

- **"~3.3.5"** → Instalará la última versión de 3.3 (no actualizará a la 3.4 o superior) pero como mínimo deberá ser la versión 3.3.5 (el carácter ~ significa que los 2 primeros números, 3.3, deben respetarse). Equivale a **"3.3.x".**

Actualización de paquetes

Para actualizar todas las dependencias de un proyecto debemos ejecutar **npm update** (mirará en nuestro archivo package.json para ver a qué versiones se permite actualizar). Para actualizar un paquete en concreto utilizaremos **npm update paquete**. Podemos usar **--prod** o **--dev** para actualizar sólo paquetes en producción o en desarrollo. O **-g** para actualizar un paquete global.

Eliminando paquetes

Para eliminar un paquete de nuestro proyecto teclearemos **npm uninstall package**. Si añadimos **--save**, nos eliminará también sus dependencias del proyecto (si no necesitamos ese paquete nunca más). En lugar de **uninstall** podemos usar **remove**, **rm**, **un**, **r** o **unlink** para hacer lo mismo. Para desinstalar un paquete global usaremos la opción **-g**.

npm r jquery --save → Elimina JQuery (y sus dependencias) del proyecto y también del archivo package.json.

npm prune --production → Elimina las dependencias de desarrollo dejando sólo las dependencias de **producción.**

Automatizando tareas con scripts en NPM

En nuestro archivo **package.json**, además de administrar las dependencias, podemos crear algunos scripts útiles para nuestro proyecto (ejecutar un servidor web, testear nuestra aplicación, minificar, etc.). Estos scripts son comandos y todos tienen un nombre que les identifica. Debemos poner estos scripts dentro de la sección "script" (es un array). La sintaxis del script será: "**nombre-script": "Comando a ejecutar**". Para ejecutar un script usaremos: **npm run nombre-script**.

```
"scripts": {
    "hello": "echo 'hello'"
}
```

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm run hello

> project@1.0.0 hello /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
> echo 'hello'

hello
```

Scripts de inicio

Hay algunos scripts como **start**, **stop** o **test** que se pueden ejecutar sin la necesidad de escribir **run**. El script *start* normalmente se usa para ejecutar un servidor web, y por tanto probar nuestra aplicación. A veces, se usa para ejecutar un compilador de TypeScript o de SASS, por ejemplo.

En nuestro ejemplo, el script *start* abrirá el navegador chrome con una URL donde tendremos nuestro servidor web ejecutándose, de forma que podremos probar nuestra aplicación:

```
"scripts": {
    "start": "google-chrome http://localhost/project"
}
```

Ejecutaríamos este script escribiendo **npm start**.

Script de test

En una aplicación seria, querremos probar nuestro código antes de publicarlo, por tanto deberíamos crear un script (o más) que ejecute los tests.

Vamos a hacer un ejemplo para depurar un archivo JavaScript con **ESLint**. Esta herramienta nos permite comprobar nuestro código (errores del código, buenas prácticas, etc.), pero no realizar test de unidad u otro tipo de tests (mejor usar otras herramientas como mocha, jasmine, karma,…).

Primero vamos a instalar de forma global ESLint ejecutando **npm i eslint -g**. Una vez instalado ejecutamos el comando: **eslint --init** dentro del directorio de proyecto. Nos hará una serie de preguntas sobre la configuración de la herramienta:

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ eslint --init
? How would you like to configure ESLint? Answer questions about your style
? Are you using ECMAScript 6 features? No
? Where will your code run? Browser
? Do you use CommonJS? No
? Do you use JSX? No
? What style of indentation do you use? Tabs
? What quotes do you use for strings? Double
? What line endings do you use? Unix
? Do you require semicolons? Yes
? What format do you want your config file to be in? JSON
Successfully created .eslintrc.json file in /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
```

Se creará un archivo de configuración con las opciones elegidas (**.eslintrc.json** en mi caso, en Linux). Podemos cambiar algunas de las reglas, por ejemplo en lugar de mostrar un error, que muestre un warning (cambiando “error” por “warn”):

```
{
    "env": {
        "browser": true
    },
    "extends": "eslint:recommended",
    "rules": {
        "indent": ["error",  "tab"],
        "linebreak-style": ["error",  "unix" ],
        "quotes": ["warn", "double"],
        "semi": [error", "always"]
    }
}
```

Ahora, vamos a ejecutar el test sobre algún archivo JavaScript:

```
'strict mode';

function printNum() {
        num = 34; // Indentación con tabulador
    return num * 2; // Espacios en lugar de tabulado
}

alert('Hello World!');
printNum() // He olvidado poner el punto y coma ';'
```

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ eslint main.js

/home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project/main.js
  1:1   warning  Strings must use doublequote                        quotes
  4:2   error    'num' is not defined                                no-undef
  5:5   error    Expected indentation of 1 tab character but found 0 indent
  5:12  error    'num' is not defined                                no-undef
  8:7   warning  Strings must use doublequote                        quotes
  9:11  error    Missing semicolon                                   semi

✖ 6 problems (4 errors, 2 warnings)
```

Ahora podemos poner el comando en nuestro **package.json**, para que cuando ejecutemos **npm test** (o npm tst), ejecute eslint sobre ese fichero:

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "eslint main.js"
}
```

Tarea previa o posterior al script (hooks)

Algunas tareas requieren realizar algunos pasos previos como por ejemplo minificar nuestro código JavaScript o CSS antes de ejecutar la aplicación (start script). Para ejecutar más de un comando podemos enlazarlos usando && (and). Estos comandos se ejecutarán en orden (excepto si uno de ellos falla).

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "echo 'We are going to make some tests' && eslint main.js && echo
'Test successful!'",
}
```

Sin embargo, es más limpio si usamos los *hooks* de NPM (prefijos) **pre** y **post**. Cuando creas un script que se ejecute junto a otro y lo quieras ejecutar antes usarás (pre), o después (post) seguido del nombre del script principal. En este ejemplo, crearemos un script **pretest** y un **posttest**.

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "eslint main.js",
    "pretest": "echo 'We are going to make some tests'",
    "posttest": "echo 'Test successful!'"
}
```

Ahora, cuando ejecutemos test, se ejecutará primero **pretest**, después **test**, y finalmente **posttest** en este orden. Si un script falla, el siguiente no será ejecutado.

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm test -s
We are going to make some tests

/home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project/main.js
  1:1  warning  Strings must use doublequote  quotes

✖ 1 problem (0 errors, 1 warning)

Test successful!
```

Ejemplo: Minificar con uglify

Uglify es una herramienta que minifica (comprime) archivos JavaScript, haciéndolos más ligeros, lo que hace que nuestras páginas carguen más rápido, y más complejos de entender o leer. En nuestro proyecto, uglify se instalaría como una dependencia de desarrollo (**npm i uglify-js -D**).

Ahora, sólo necesitamos añadir una línea como esta en la parte de los script, lo cual comprimirá y unificará todos los archivos JS del directorio **js**/ en un único archivo: **bundle.js** (que incluiremos en nuestro HTML):

```
"build": "uglifyjs  -mc -o bundle.js js/*.js"
```

Si ejecutamos **npm run build**, nos generará **bundle.js** con el código JS minificado dentro:

```
"strict mode";function printNum(){var r=34;return 2*r}alert("Hello
World!"),printNum();
```

Ejemplo: Ejecutar un script cuando algunos archivos cambian

Hay una herramienta que puede utilizarse cuando alguno de los archivos que tenemos en un directorio cambia. Esta herramienta se llama **watch**, y podemos añadirla a nuestras dependencias del proyecto ejecutando **npm i watch --save-dev**.

Para usar esta herramienta, crearemos un script que ejecute: **watch 'comando' directorio**. Cuando se produzca un cambio en un archivo dentro del directorio especificado, el comando será ejecutado de forma automática.

```
"build": "uglifyjs  -mc -o bundle.js js/*.js",
"build:watch": "watch 'npm run build' ./js" → Cambio detectado en js/
```

Ahora, si ejecutamos **npm run build:watch**, esta herramienta estará de forma continua vigilando y cuando un archivo del directorio **js** cambie, se ejecutará el script **build**. Para pararlo, pulsaremos **Ctrl+c** en la consola.

Ejemplo: Lanzando un servidor web y vigilando cambios de forma concurrente

Si ejecutamos un servidor web usando NPM (lite-server por ejemplo), no podremos ejecutar ninguna otra tarea hasta que esta acabe (cerrada con Ctrl+c por ejemplo). Hay un paquete en NPM que nos permite ejecutar más de una tarea de forma concurrente y se llama concurrently.

Primero instalaremos estos dos dependencias en modo desarrollo (**npm i lite-server -D**) (**npm i concurrently -D**). Creamos una tarea que ejecutará a su vez dos tareas a la vez, pasándole los comandos como parámetro (entre comillas).

```
"start": "concurrently \"npm run build:watch\" \"npm run serve\"",
"serve": "lite-server",
"build": "uglifyjs  -mc -o bundle.js js/*.js",
"build:watch": "watch 'npm run build' ./js"
```

```
"devDependencies": {
  "concurrently": "^2.2.0",
  "lite-server": "^2.2.2",
  "uglify-js": "^2.7.10",
  "watch": "^0.19.2"
}
```

lite-server carga por defecto el archivo con nombre **index.html** como punto de partida.

```
$: npm start

> project@1.0.0 start /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
> concurrently 'npm run build:watch' 'npm run serve'

[0]
[0] > project@1.0.0 build:watch /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
[0] > watch 'npm run build' ./js
[0]
[1]
[1] > project@1.0.0 serve /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
[1] > lite-server
```

Webpack

Introducción

Otra forma de solucionar el problema de tener que incluir todos los archivos en el html, además de obtener otras muchas ventajas, es utilizar el empaquetado de módulos (module bundler). La idea es coger los assets de un sitio web (JavaScript, CSS y HTML), y transformarlos en un formato más conveniente para consumirlo a través de un navegador. Haciendo esto correctamente nos evitaremos muchos dolores de cabeza durante nuestro desarrollo web.

Webpack no es la herramienta más fácil de aprender debido a su enfoque basado en la configuración, pero es increíblemente potente.

¿Qué es Webpack?

Los navegadores Web están diseñados para consumir HTML, CSS y JavaScript. A medida que un proyecto crece, el seguimiento y la configuración de todos estos archivos se hace demasiado complejo para administrarlo sin ayuda. Webpack fue diseñado para ayudar a resolver este problema.

Así pues, Webpack es un **empaquetador de módulos (module bundler)**, es decir, nos permite generar un achivo único con todos aquellos módulos que necesita nuestra aplicación para funcionar. Un módulo será un fichero javascript, por lo tanto, nos permitirá **meter todos nuestros archivos javascript en un** único archivo, llamémoslo *bundle*.

Obviamente, la primera ventaja que podemos deducir de esta definición es que, al igual que ocurre cuando utilizamos un sistema de módulos como los vistos en el punto anterior, ya no vamos a necesitar enlazar todos los ficheros javascript que utilizamos en nuestra aplicación en el código html, bastará con empaquetar todos estos ficheros en un bundle y enlazar éste con nuestro html. Pero Webpack tiene muchas otras ventajas que lo convierten en una herramienta indispensable para todo desarrollador front-end. Entre otras cosas, destaca que:

- Puede generar sólo aquellos fragmentos de JS que realmente necesita cada página (haciendo más rápida su carga).

- Dispone de herramientas (loaders) que permiten importar y empaquetar también otros recursos (CSS, templates, …), así como ficheros codificados en otros lenguajes (ES6 con Babel, TypeScript, SaSS, etc).

- Sus *plug-ins* permiten hacer otras tareas importantes, como por ejemplo minificar, ofuscar, comprimir el código...

Hay otras herramientas en el mercado que realizan tareas similares a las que nos ofrece Webpack, por ejemplo browserify o requirejs, pero Webpack va un paso más allá gracias, sobre todo, a su sistema de plugins. También tenemos automatizadores de tareas (tasks runners) como Grunt o Gulp, pero estas nos obligan a escribir el flujo

de trabajo de forma manual. Llevar ese problema a un paquete, como hace Webpack, es un avance importante.

Conceptos

Antes de entrar de lleno en el uso de la herramienta, estudiaremos cuatro conceptos que son fundamentales para entender el funcionamiento de la misma:

1. Punto de entrada (Entry): Webpack crea un grafo con todas las dependencias de la aplicación. El punto de partida de este grafo se conoce como punto de entrada. El punto de entrada le dice a Webpack dónde comenzar a analizar dependencias para saber qué hay que empaquetar. Podemos pensar que el punto de entrada de nuestra aplicación será el primer archivo que se cargará, a partir del cuál se irán añadiendo al bundle todos los recursos de los que dependan el punto de entrada y sus dependencias.

2. Punto de salida (Output): Una vez que hemos empaquetado todos los archivos (assets) de nuestra aplicación, tenemos que decirle a Webpack dónde almacenaremos este paquete (en qué archivo lo guardaremos), es decir, cuál será el punto de salida. Este fichero será el único que tendremos que enlazar con nuestro html.

3. Cargadores (loaders): El objetivo es que la responsabilidad de gestionar todos los assets del proyecto sea de Webpack y no del navegador. Webpack trata cada fichero (.css, .html, .scss, .jpg, etc.) como un módulo. Sin embargo, **Webpack sólo entiende javascript**. Por este motivo, los cargadores transforman estos archivos en módulos a medida que se añaden al gráfico de dependencias de Webpack.

4. Plugins: Dado que los cargadores sólo ejecutan transformaciones tomando como base el archivo, los plugins se suelen utilizar (aunque no se limitan a esto) para realizar acciones y funcionalidades personalizadas en "compilaciones" o "fragmentos" de los módulos empaquetados. Ejemplos de tareas que se pueden realizar utilizando plugins son: minificación de código javascript, minificación de código css, compresión de código, etc.

Instalar Webpack

Podemos instalar Webpack de forma local para un proyecto determinado o de forma global para todos los proyectos. Lo más recomendable es instalarlo de forma local a un proyecto, ya que, de esta forma, podremos tener distintas versiones de Webpack para distintos proyectos.

Instalar Webpack localmente

Veamos como instalar Webpack localmente utilizando npm:

- Para instalar la última versión estable:

```
npm install webpack –save-dev
```

- Para instalar la versión indicada con @<version>:

```
npm install webpack@<version> --save-dev
```

Para ejecutar la instalación local de Webpack podemos acceder a la versión ejecutable a través de *node_modules/.bin/webpack*

Si utilizamos scripts npm en nuestro proyecto, npm intentará buscar webpack en los módulos locales, por lo que esta técnica de instalación resulta útil en estos casos.

```
"scripts": {
  "build": "webpack --config mywebpack.config.js"
}
```

Este es el uso más habitual de Webpack, lo veremos más claramente con un ejemplo un poco más adelante.

Instalar Webpack globalmente

Para instalar Webpack de forma global utilizando npm, haremos lo siguiente:

```
npm install --global webpack
```

Como ya hemos comentado anteriormente, este método de instalación no es el más recomendable, ya que obligaría a que todos nuestros proyectos utilizaran la misma versión de Webpack.

Un ejemplo sencillo

Comenzaremos con un ejemplo muy sencillo que nos permitirá ver las posibilidades que nos ofrece Webpack. Para implementar el ejemplo, seguiremos los siguientes pasos:

1. Creamos el directorio donde guardaremos nuestra aplicación y entramos en él:
   - `mkdir curso-webpack`
   - `cd curso-webpack`
2. Inicializamos npm:
   - `npm init -y`
3. Instalamos Webpack localmente:
   - `npm install --save-dev webpack`
4. Comprobamos que se ha instalado correctamente:
   - `./node_modules/.bin/webpack -v`
5. Creamos un directorio src que será donde guardaremos los módulos javascript de nuestra aplicación:
   - `mkdir src`
   - `cd src`
6. Creamos el fichero app.js que será el punto de entrada de nuestra aplicación:

```
function component () {
    var element = document.createElement('div');

    /* para la ejecución de la siguiente línea es necesario disponer de la
librería lodash */

    element.innerHTML = _.join(['Hello','webpack'], ' ');
    return element;
}
document.body.appendChild(component());
```

7. Creamos el documento index.html en el directorio raíz de nuestro proyecto (curso-webpack):

```
<html>
<head>
    <title>webpack 2 demo</title>
    <script src="https://unpkg.com/lodash@4.16.6"></script>
</head>
<body>
<script src="src/app.js"></script>
</body>
</html>
```

Si abrimos el documento index.html con el navegador, veremos que aparece el mensaje "`Hello webpack`" en el cuerpo de la página.

Como se puede ver, en este ejemplo hay dependencias implicitas entre las dos etiquetas `<script>` del fichero index.html. En concreto, el fichero app.js depende de la librería lodash, que debe ser incluida en la página para que la ejecución sea correcta. Decimos que la dependencia es implicita porque el fichero app.js no declara una dependencia de la librería lodash, simplemente asume que la variable global "_" existe.

Los proyectos que funcionan de esta forma tienen una serie de problemas:

- Si una dependencia no se incluye en el html o no se incluye en el orden adecuado, la aplicación no funcionará correctamente.

- Si incluimos una dependencia que no se usa, el navegador estará descargando código javascript que no es necesario.

Para solucionar estos problemas utilizaremos npm para instalar las librerías lodash y webpack, después crearemos el bundle con las dependencias de nuestra aplicación. Seguiremos los siguientes pasos:

1. Instalamos lodash con npm:

   npm install --save lodash

2. Importamos el módulo en nuestro fichero app.js:

   ```
   import _ from 'lodash';
   ```

3. Modificamos el fichero index.html para que utilice el fichero bundle.js que vamos a generar en lugar de incorporar los dos ficheros javascript anteriores.

Quedará así:

```
<html>
<head>
    <title>webpack 2 demo</title>
</head>
<body>
<script src="dist/app.bundle.js"></script>
</body>
</html>
```

En este caso, app.js está declarando la dependencia de la librería de forma explicita. Al indicar qué dependencias necesita un módulo, Webpack puede utilizar esta información para crear un gráfico de dependencias. A continuación, utiliza el gráfico para generar un paquete optimizado en el que los scripts se ejecutarán en el orden correcto. Además, las dependencias que no se utilicen no se incluirán en el paquete.

4. Por último, ejecutaremos Webpack con `app.js` como punto de entrada y `app.bundle.js` como punto de salida:

   ```
   ./node_modules/.bin/webpack src/app.js dist/app.bundle.js
   ```

   Este comando creará el directorio dist que contendrá el fichero app.bundle.js en el que se incluirá todo el código necesario para el correcto funcionamiento de la aplicación.

Si volvemos a cargar el fichero index.html en el navegador, comprobaremos que sigue funcionando exactamente igual que antes de realizar los cambios.

Una cosa a destacar de este ejemplo es que estamos utilizando la sentencia import de ES2015 (no soportada todavía en la mayoría de navegadores) sin necesidad de transpilar el código para su correcto funcionamiento. Esto es así porque Webpack reemplaza estas instrucciones por otras compatibles con ES5.

En cualquier caso, Webpack sólo sustituye las sentencias import y export, por lo que, si en nuestro código estamos utilizando otras características de ES2015, debemos asegurarnos de usar un transpilador de código, como por ejemplo el que hemos visto en el punto anterior “Babel” (más adelante veremos como podemos hacer esto).

El archivo de configuración de Webpack

Para una configuración más compleja, podemos usar un archivo de configuración en el que le indiquemos a Webpack cómo debe realizar el proceso de empaquetado. El archivo de configuración de Webpack por defecto se llama `webpack.config.js`.

Podemos crear un archivo de configuración que genere el mismo bundle que el comando utilizado en el ejemplo anterior de la siguiente forma:

```
var path = require('path');
module.exports = {
  context: path.resolve(__dirname, './src'),
    entry: {
              app: './app.js',
  },
```

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

Nota: La variable __dirname hace referencia al directorio en el que se encuentra ubicado el fichero de configuración, que normalmente será la raíz del proyecto.

Este fichero puede ser ejecutado por Webpack con el siguiente comando:

./node_modules/.bin/webpack --config webpack.config.js

En cualquier caso, como estamos utilizando el nombre por defecto para el archivo de configuración, nos podemos ahorrar el parámetro `--config`. El siguiente comando funcionará exactamente igual que el anterior:

```
./node_modules/.bin/webpack
```

El archivo de configuración es un archivo JavaScript que exporta un objeto. Después, Webpack se encargará de procesar este objeto para realizar el empaquetado del bundle.

Debido a que el fichero de configuración es un módulo estándar de Node.js CommonJS, podemos hacer cosas como:

- Importar otros archivos a través de require.
- Utilizar expresiones de control de flujo de JavaScript, por ejemplo, el operador condicional (?:).
- Utilizar constantes o variables para valores utilizados con frecuencia.
- Crear funciones para generar una parte de la configuración...

Usando Webpack con npm

Como se puede observar, no es demasiado cómodo utilizar Webpack con la interfaz de comandos. Para mejorar esto ya hemos comentado anteriormente que podemos crear un pequeño atajo utilizando npm. Introduciremos el siguiente script en el fichero package.json:

```
"scripts": {
  "build": "webpack --config mywebpack.config.js"
}
```

De esta forma, podremos generar el bundle ejecutando el siguiente comando:

```
npm run build
```

npm crea un entorno temporal para ejecutar los scripts en el que incluye todos los directorios bin de las dependencias de desarrollo, por lo tanto, tendremos acceso a Webpack.

Podemos pasar parámetros a Webpack añadiendo dos guiones al comando npm run build, por ejemplo:

```
npm run build -- --watch
```

El comando anterior activa el watcher de Webpack, más adelante veremos para qué sirve esto.

Funcionamiento de Webpack

Webpack básicamente hace lo siguiente:

1. Comenzando en la carpeta indicada en context, busca los ficheros indicados en entry y lee su contenido.

2. Cada dependencia en forma de import (ES6) o require (Node) que encuentra al analizar el código, se guarda para la compilación final. A continuación, busca esas dependencias y las dependencias de esas dependencias, hasta que llega al final del "árbol", uniendo sólo aquello que necesita.

3. A partir de ahí, Webpack empaqueta todo en el directorio output.path, usando el nombre indicado en output.filename como nombre de archivo ([name] se reemplaza con la clave indicada en entry, en el caso del ejemplo anterior 'app').

Modo watch y servidor de desarrollo integrado

Tener que ejecutar un comando manualmente para recompilar nuestro bundle cada vez que hagamos un cambio en nuestro Javascript antes de poder probarlo puede llegar a resultar un tanto tedioso. Para mejorar este proceso disponemos de dos posibilidades: ejecutar Webpack en modo watch o utilizar el servidor de desarrollo integrado con Webpack.

Modo watch

Si ejecutamos Webpack en modo watch se quedará en ejecución a la espera de que hagamos cambios en nuestros módulos. Cada vez que se detecte un cambio se realizará la compilación y actualización de los bundles implicados.

Para ejecutar Webpack en modo watch sólo tenemos que lanzarlo pasando el parámetro --watch. El comando será el siguiente:

```
./node_modules/.bin/webpack --watch
```

Por supuesto, podemos utilizar el script de npm para lanzar Webpack en modo watch. Ya comentamos anteriormente que para pasar un parámetro a un script npm sólo teníamos que colocar delante dos guiones, por lo tanto, lanzaremos el script con el siguiente comando para poner Webpack en modo watch:

```
npm run build -- --watch
```

Como ya hemos comentado, este comando dejará Webpack a la espera de que se realice algún cambio en uno de nuestro ficheros para recompilar el bundle.

Servidor de desarrollo integrado

Para simplificar el desarrollo de nuestra aplicación en la parte Front-End, Webpack incorpora un servidor web de desarrollo que nos va a permitir probar nuestras aplicaciones de forma muy sencilla.

Este servidor de desarrollo nos va a proporcionar (entre otras cosas) lo que se conoce como recarga de la página en vivo (live reloading), es decir, si tenemos nuestra aplicación web cargada en el navegador y hacemos un cambio en nuestro código javascript, Webpack recompilara el código, lo volverá a empaquetar y recargará la página sin necesidad de que nosotros tengamos que realizar acción alguna, ni siquiera recargar la página en el navegador.

Instalación del servidor

Para poder utilizar este servidor web, lo primero que tenemos que hacer es instalarlo utilizando npm. Ejecutaremos el siguiente comando:

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

Esto instalará un servidor que escuchará las conexiones que se realicen a localhost en el puerto 8080 (el puerto de escucha es configurable, pero nosotros utilizaremos el puerto por defecto).

Configuración del servidor

Para poder utilizar el servidor tenemos que añadir el objeto devServer a nuestro fichero webpack.config.js:

```
var path = require('path');
module.exports = {
    context: path.resolve(__dirname, './src'),
    entry: {
        app: './app.js',
    },
    output: {
        filename: '[name].bundle.js',
        path: path.resolve(__dirname, 'dist'),
    },
    devServer: {  // Configuración del servidor web de desarrollo
        contentBase: path.resolve(__dirname, './'),
        publicPath: '/dist/'
    }
};
```

Como vemos en el ejemplo, el objeto devServer contiene dos propiedades:

- publicPath: Le indica al servidor cuál es la url del directorio en el que tiene que servir los bundles generados por Webpack. Debemos indicar aquí la url que daría acceso al directorio en el que hemos configurado la propiedad output.path (en nuestro caso /dist/). Esto es así porque el servidor genera los bundles en memoria, de modo que tenemos que indicar cómo acceder a ellos.

Nota: para ver el resto de propiedades disponibles en el objeto devServer podemos acceder a la

siguiente url:

https://webpack.js.org/configuration/dev-server/#devserver

Una vez hemos instalado y configurado nuestro servidor, tendremos que arrancarlo en un terminal independiente, ya que la ejecución bloqueará el acceso a la consola.

Para arrancarlo podemos utilizar el siguiente comando:

./node_modules/.bin/webpack-dev-server

Al igual que ocurría con la ejecución de Webpack, esta forma de ejecución es un poco farragosa, así que nos crearemos un script npm para ejecutarlo. En nuestro archivo package.json añadiremos el siguiente script:

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

Ahora, podemos ejecutar el servidor con el comando:

npm run webserver

Tras la ejecución del comando nos aparecerá una información como la de la siguiente imagen:

```
> curso-webpack@1.0.0 server /home/dwes/PhpstormProjects/curso-webpack
> webpack-dev-server

Project is running at http://localhost:8080/
webpack output is served from /dist/
Content not from webpack is served from /home/dwes/PhpstormProjects/curso-webpack
Hash: 03780eabd811a04fc78d
Version: webpack 2.3.2
Time: 3822ms
```

Aquí nos está diciendo (entre otras cosas) que para acceder a nuestro proyecto debemos ir a la url http://localhost:8080, y que los bundles de Webpack se sirven en /dist/.

Si accedemos a esta url, se cargará el fichero index.html. Como este fichero tiene vinculado el fichero dist/app.bundle.js y el servidor web utiliza la ruta dist/ para acceder a los bundles que tiene generados en memoria, visualizaremos la página sin ningún problema.

- contentBase: Le indica al servidor de donde tiene que cargar el contenido estático (en nuestro caso los ficheros html). Le estamos diciendo que debe tomarlo del directorio en el que se encuentra el fichero webpack.config.js (en realidad, este es el valor que toma por defecto, por lo que, en este caso, no sería obligatorio indicar esta propiedad).

Ejecución del servidor

Si con la página cargada, nos vamos a editar el fichero app.js y cambiamos el texto “Hello“ por el texto “Hola”, veremos que al volver al navegador se reflejará en la página el cambio que acabamos de realizar sin necesidad de recargar la página.

Evidentemente, cuando utilizamos el servidor web de desarrollo integrado en Webpack ya no necesitamos utilizar el modo watch, el propio servidor se encargará de actualizar los bundles cuando se detecten cambios en los ficheros del proyecto.

Trabajando con múltiples puntos de entrada

En muchas ocasiones, tenemos aplicaciones que no son SPA (Single Page Application) en el sentido estricto del término, sino que están compuestas por diferentes partes más o menos independientes entre sí. Así, para acceder a cada una de las partes de nuestra aplicación tendremos un archivo html que tendrá vinculado el javascript que necesite para su correcto funcionamiento.

Para cada una de estas partes, tendremos que definir un punto de entrada que generará un bundle, y este será el que vincularemos al archivo html correspondiente.

Podemos especificar cualquier número de puntos de entrada / salida modificando sólo el objeto entry del archivo de configuración de Webpack:

- Si tenemos multiples ficheros que queremos que se empaqueten juntos:

```
var path = require('path');
module.exports = {
    context: path.resolve(__dirname, './src'),
    entry: {
        app: ['./app.js', './fecha.js']
    },
    output: {
        filename: '[name].bundle.js',
        path: path.resolve(__dirname, 'dist')
    }
};
```

  El bundle incluirá los ficheros app.js y fecha.js junto con las dependencias de ambos. El orden en el que aparecerán en el bundle generado coincidirá con el indicado en el array.

- Si tenemos múltiples ficheros que queremos que se empaqueten por separado:

```
var path = require('path');
module.exports = {
    context: path.resolve(__dirname, './src'),
    entry: {
        app: './app.js',
        fecha: './fecha.js'
    },
    output: {
        filename: '[name].bundle.js',
        path: path.resolve(__dirname, 'dist')
```

```
    }
};
```

En este caso, se generarán dos ficheros (app.bundle.js y fecha.bundle.js) que podrán ser vinculados a cualquier html de manera independiente. Cada uno de estos bundles incorporará las dependencias del fichero de entrada correspondiente.

Nota: El problema de tener varios puntos de entrada puede surgir cuando estos ficheros tienen dependencias comunes (muy habitual cuando ambos ficheros utilizan las mismas librerías de terceros), para solucionar este problema podemos utilizar el plugin commonsChunkPlugin que veremos más adelante.

Ejemplo con múltiples ficheros de entrada

Vamos a modificar el ejemplo anterior para tener dos puntos de entrada, para ello seguiremos los siguientes pasos:

1. Crearemos un nuevo documento html llamado fecha.html en el directorio curso-webpack:

```
<html>
<head>
    <title>webpack 2 fecha</title>
</head>
<body>
<script src="dist/fecha.bundle.js"></script>
</body>
</html>
```

Como se puede observar este documento tiene vinculado el fichero javascript dist/fecha.bundle.js que generaremos con Webpack.

2. Crearemos un nuevo fichero javascript llamado component.js en el directorio curso-webpack/src:

```
export default function getComponent(contenido)
{
    var element = document.createElement('div');
    element.innerHTML = contenido;
    return element;
}
```

Este fichero exporta la función getComponent utilizando la directiva export de ES2015. Ya comentamos anteriormente, que esta es la única característica (junto con el import) que entiende Webpack de esta versión del lenguaje sin necesidad de transpilar el código.

3. Modificaremos el fichero curso-webpack/src/app.js de la siguiente forma:

```
import _ from 'lodash';
import getComponent from './component';

var contenido = _.join(['Hello','webpack'], ' ');
document.body.appendChild(getComponent(contenido));
```

Ahora estamos importando la librería externa lodash y la función getComponent del módulo component que hemos creado en el paso anterior. De esta forma, al empaquetar, Webpack detectará que hay una dependencia con el fichero component y lo incluirá en el bundle resultante. Después utilizamos la función getComponent para mostrar en la página el contenido generado mediante el método join de lodash.

4. Crearemos un nuevo módulo javascript llamado fecha.js en el directorio curso-webpack/src:

```
import moment from 'moment';
import getComponent from './component';

var contenido = "Hoy es " + moment().format('DD/MM/YYYY H:mm:ss');
document.body.appendChild(getComponent(contenido));
```

   Este módulo es muy parecido al app.js. Utiliza el método getComponent del módulo component para mostrar contenido en la página, por lo que necesita importarlo. La diferencia es que en lugar de generar el contenido utilizando la librería lodash, genera una cadena que contiene la fecha actual formateada utilizando la librería moment que sirve para trabajar con fechas. Así pues, este módulo también importa la librería moment, por lo que necesitaremos instalarla.

5. Utilizaremos npm para instalar la librería moment en nuestro proyecto:

   npm install --save moment

6. Ya solo nos queda configurar Webpack para que genere dos bundles: app.bundle.js que incluimos en index.html y fecha.bundle.js que incluimos en fecha.html. Para ello utilizaremos el archivo de configuración que hemos visto anteriormente para explicar la generación de múltiples puntos de entrada y salida:

```
var path = require('path');
module.exports = {
    context: path.resolve(__dirname, './src'),
    entry: {
        app: './app.js',
        fecha: './fecha.js'
    },
    output: {
        filename: '[name].bundle.js',
        path: path.resolve(__dirname, 'dist')
    }
};
```

7. Por último, generaremos los bundles con el comando de npm:

   npm run build

Si abrimos el fichero index.html con el navegador veremos que nos muestra el mensaje “Hello webpack”, mientras que si abrimos el fichero fecha.html nos muestra la fecha y hora actuales.

Webpack ha generado dos bundles:

- app.bundle.js: que inluirá el módulo app.js y sus dependencias (component.js y lodash).
- fecha.bundle.js: que inluirá el módulo fecha.js y sus dependencias (component.js y moment).

Con esto hemos conseguido que sólo se introduzcan en el bundle los módulos necesarios.

Módulos ES2015 (export / import)

Vincular con el html todos los archivos JavaScript que necesitábamos tiene varias desventajas:

- Causa confusión cuando cargamos muchos archivos (librerías, archivos propios, …).
- Hace que nuestro HTML se quede sucio.
- Muchos archivos utilizan variables globales que pueden ser usadas en otros archivos, pero también anuladas por otra variable con el mismo nombre.

Webpack soporta la carga de módulos (archivos importados desde otros) a través de diferentes métodos. Los más usados son el sistema CommonJS mediante la instrucción **require** para importar (sistema que usa NodeJS), y el sistema que introdujo ES2015 → **export** / **import**. Para el curso usaremos el último método.

Con la carga de módulos decidimos lo que un archivo (módulo) debe poner a disposición del resto de los módulos. La forma más sencilla de hacerlo es exportar sólo una función, clase, variable, etc. utilizando la sintaxis **export default** (no necesitamos darle un nombre). Luego, desde otros archivos, lo importamos dándole el nombre que queremos.

```
Fichero: src/person.class.js
export default class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}

Fichero: src/main.js
import Person from './person.class';// Lo importamos con el nombre Person
let p = new Person("Peter", 42);
console.log(p.name); // Muestra "Peter"
```

Si queremos exportar indicando el nombre de lo que exportamos, o bien exportar más de una cosa del módulo, omitiremos la palabra clave **default** y le daremos un nombre al elemento que estamos exportando. La importación tiene que hacerse entre corchetes {} utilizando el mismo nombre con el que exportamos el elemento.

```
Fichero: src/person.class.js
export class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
Fichero: src/main.js
import {Person} from './person.class'; // Lo importamos con su nombre
let p = new Person("Peter", 42);
console.log(p.name); // Muestra "Peter"
```

Otra forma de exportar algo es declararlo al final del fichero. Después, podremos importar lo que nos interese desde otros archivos. No tenemos que importar todo lo que se está exportando en un archivo, sólo lo que necesitamos.

```
Fichero: src/person.class.js
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
const ROLES = ["admin", "guest", "user"];
const GUEST_NAME = "Anonymous";
export {Person, ROLES, GUEST_NAME};

Fichero: src/main.js
import {Person, GUEST_NAME} from './person.class';
let p = new Person(GUEST_NAME, 30);
console.log(p.name); // Muestra "Anonymous"
```

También podemos importar todo lo que se está exportando en un archivo usando el asterisco *, pero tendremos que darle un alias usando la palabra clave **as** (como en SQL). Entonces, podemos acceder a todo lo que se exporta utilizando el alias (es como un objeto que contiene todas las exportaciones).

```
import * as persons from './person.class';
let p = new persons.Person(persons.GUEST_NAME, 30);
console.log(p.name); // Prints "Anonymous"
```

En resumen: A partir del archivo (o archivos) que hayamos escogido como punto de entrada en Webpack, importaremos el resto de archivos usando lo que acabamos de ver, nunca en el HTML directamente.

Cargadores (Loaders)

Como ya comentamos al inicio del tema, Webpack únicamente entiende el lenguaje javascript, pero nos permite trabajar con prácticamente cualquier tipo de archivo, siempre y cuando lo convirtamos previamente a JavaScript. Esta tarea la realizaremos con lo que se conoce como loaders.

Un loader puede referirse a un preprocesador como Sass, o un transpiler como Babel. En NPM, usualmente se les llama *-loader como sass-loader, css-loader o babel-loader.

En el archivo de configuración introduciremos los loaders dentro del objeto module. Aquí tendremos un array de rules, donde cada rule hará referencia a un loader. Lo veremos mucho más claro con un ejemplo.

Loader para ES6 con Babel

Para poder utilizar el loader de ES6 vía Babel en nuestro proyecto lo primero que haremos será instalarlo localmente utilizando npm:

npm install --save-dev babel-loader babel-core babel-preset-latest

Una vez instalado tendremos que añadir la regla del loader en el archivo de configuración webpack.config.js:

```
var path = require('path');
module.exports = {
    context: path.resolve(__dirname, './src'),
    entry: {
        app: './app.js',
        fecha: './fecha.js'
    },
    output: {
        filename: '[name].bundle.js',
        path: path.resolve(__dirname, 'dist'),
    },
    module: {
        rules: [
            {
                test: /\.js$/,
                exclude: [/node_modules/],
                use: [{
                    loader: 'babel-loader',
                    options: {presets: ['latest']},
                }],
            },
            // Aquí estarían los loaders para otros tipos de archivo
        ]
    },
    devServer: {  // Configuración del servidor web de desarrollo
        contentBase: path.resolve(__dirname, './'),
        publicPath: '/dist/'
    }
};
```

Como se puede observar en el ejemplo hemos añadido el objeto module y dentro de este el array de rules. En este array hemos añadido un único cargador para ficheros javascript. Las propiedades que tiene el cargador son las siguientes:

- test: indica una expresión regular que nos servirá para discriminar que ficheros se deben de convertir. En este caso, se convertirán todos los ficheros cuyo nombre acabe en .js, es decir, archivos javascript.

- Exclude: Indicamos un array de rutas que queremos que se excluyan, es decir, que el cargador no convierta los archivos existentes en estas rutas. En nuestro caso, hemos excluido la ruta /node_modules/, ya que no nos interesa que los

librerías instaladas con npm se vean afectadas por este cargador.

- Use: En esta propiedad indicaremos un array con los loaders que vamos a pasar y sus correspondientes opciones. En nuestro caso, pasamos el loader babel-loader con la configuración para EcmaScript 2015 y futuras versiones.

Para probar el correcto funcionamiento del loader, basta con introducir alguna de las nuevas características de ES6 en nuestro ejemplo y comprobar que el bundle generado la convierte a ES5 sin problemas. Por ejemplo, podemos ir al fichero src/app.js y cambiar la siguiente línea:

```
var contenido = _.join(['Hello','webpack'], ' ');
```

Por esta otra:

```
let contenido = _.join(['Hello','webpack'], ' ');
```

Después generaremos el bundle con el comando correspondiente (npm run build). Si después abrimos el bundle generado dist/app.bundle.js y buscamos la cadena “'Hello'”, veremos que se ha sustituido el modificador let por var.

Loader para CSS

Otra posibilidad muy interesante que nos ofrece Webpack es integrar el css de nuestra aplicación dentro de nuestro bundle. De esta forma, podremos modularizar nuestros css como más nos interese.

Comenzaremos instalando los loaders necesarios con npm:

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

A continuación, añadiremos el loader a nuestro fichero de configuración:

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

Evidentemente, esto lo añadiremos dentro del array de rules. Simplemente, estamos indicando que los archivos css de nuestro proyecto serán convertidos utilizando los loaders css-loader y style-loader (los loaders se ejecutan en orden inverso al orden en el que aparecen en el array, es decir, primero se pasará el loader css-loader y después style-loader).

Para probar el correcto funcionamiento de estos loaders seguiremos los siguientes pasos:

1. Creamos el directorio css/ dentro del directorio src/.

2. Dentro de src/css/ creamos dos ficheros css con el siguiente contenido:

fichero app.estilos.css

```
body
{
    background-color:red;
}
```

fichero fecha.estilos.css

```
body
{
    background-color:blue;
}
```

3. En el fichero src/app.js añadimos el siguiente import:

```
import styles from './css/app.estilos.css';
```

4. En el fichero src/fecha.js añadimos el siguiente import:

```
import styles from './css/fecha.estilos.css';
```

5. Ejecutamos el comando para generar el bundle (npm run build).

Si ahora cargamos la página index.html veremos que tiene el fondo rojo, mientras que si cargamos la página fecha.html tendrá el fondo azul.

Cargar nuestro css a través de javascript nos va a permitir asociar la funcionalidad de nuestros componentes con los estilos que queremos aplicarles, lo cual puede resultar muy útil.

Plugins

De los cuatro conceptos vistos al principio del tema tan sólo nos queda hablar de los plugins. Los plugins nos servirán para ampliar las capacidades de Webpack. Para usarlos sólo tendremos que añadirlos al fichero de configuración en la propiedad plugins del objeto exportado.

Si queremos utilizar plugins integrados en Webpack debemos de hacer un require del objeto webpack para tener acceso a él:

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

Separar partes comunes commonsChunkPlugin

En muchos casos, cuando tenemos múltiples puntos de entrada, hay partes de la aplicación que se utilizan en más de un punto de entrada. Esto implica que las partes repetidas están en todos los bundles que se generen y dependan de estas, lo que no es óptimo a nivel de rendimiento, ya que no se podrá cachear el código común. Para solucionar este problema podemos utilizar el plugin CommonsChunkPlugin.

Este plugin nos va a permitir extraer todos los módulos comunes de diferentes bundles y agregarlos a un nuevo bundle común.

En nuestro ejemplo, tenemos dos puntos de entrada (app y fecha). Ambos puntos de entrada utilizan la función getComponent del módulo component, por lo tanto

dependen de él. Esto hace que el módulo component esté repetido en app.bundle.js y en fecha.bundle.js (puedes comprobarlo abriendo los dos ficheros en un editor y buscando la función getComponent, observarás que se encuentra repetida en los dos bundles).

Si queremos separar este código común en un bundle a parte, añadiremos la propiedad plugins a nuestro objeto exportado:

```
…
plugins: [
     new webpack.optimize.CommonsChunkPlugin({
         name: 'commons',
         minChunks: 2,
     }),
 ],
…
```

Como se puede observar, la propiedad plugins es un array al que le pasamos los objetos que implementan la funcionalidad que queremos utilizar. Cuando creamos el objeto, le podemos pasar en el constructor las propiedades de configuración que espera recibir. En este caso, le estamos diciendo que el nombre del punto de entrada para este bundle común es commons (por lo tanto el fichero de salida será commons.bundle.js) y en la propiedad minChunks le pasamos el número mínimo de módulos que tienen que depender de un módulo determinado para que este se traslade al bundle común, en nuestro caso, le decimos que los módulos que se repitan dos o más veces se pasarán al bundle commons.bundle.js.

Si ahora volvemos a generar los bundles (con el comando npm run build) veremos que se crea un nuevo bundle (commons.bundle.js). Si abrimos los bundles app.bundle.js y fecha.bundle.js y buscamos la función getComponent, observaremos que esta ya no se encuentra en los mismos, mientras que si la buscamos en el bundle commons.bundle.js sí que la podremos encontrar.

Evidentemente, si cargamos la página index.html o fecha.html en nuestro navegador veremos que se produce un error javascript y no se muestra nada. Esto es porque tenemos que incluir el fichero commons.bundle.js en nuestros html:

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

Introduciremos la línea anterior, tanto en index.html como en fecha.html justo encima del otro script que tienen vinculado.

Separar librerías

Habitualmente, en aplicaciones que utilizan librerías de terceros y tienen múltiples puntos de entrada, todos los puntos de entrada suelen depender de las mismas librerías, por lo tanto, es una buena práctica tener las librerías de nuestra aplicación en un bundle a parte.

Supongamos que en nuestro ejemplo cambiamos los ficheros app.js y fecha.js de la siguiente forma:

fichero app.js

```
import styles from './css/app.estilos.css';
import _ from 'lodash';
import moment from 'moment';
import getComponent from './component';
let contenido = _.join(['Página','principal'], ' ');
document.body.appendChild(getComponent(contenido));
let fecha = "Hoy es " + moment().format('DD/MM/YYYY H:mm:ss');
document.body.appendChild(getComponent(fecha));
```

fichero fecha.js

```
import styles from './css/fecha.estilos.css';
import moment from 'moment';
import _ from 'lodash';
import getComponent from './component';
let contenido = _.join(['Página','fecha'], ' ');
document.body.appendChild(getComponent(contenido));
let fecha = "Hoy es " + moment().format('DD/MM/YYYY H:mm:ss');
document.body.appendChild(getComponent(fecha));
```

Ahora los dos ficheros dependen de las dos librerías (lodash y moment). Si volvemos a generar los bundles (npm run build), veremos que ahora las librerías se han pasado al bundle commons y han desaparecido de los otros dos bundles, claro ahora se repiten dos veces, por lo tanto pasan al bundle común. Esto que puede parecer correcto, tiene un inconveniente, y es que cada vez que modifiquemos uno de nuestros módulos comunes se volverá a generar todo el bundle commons, incluyendo todas las librerías, aunque estas no han cambiado.

Para solucionar este problema añadiremos una nueva entrada en la propiedad plugins de nuestro archivo de configuración:

```
…
new webpack.optimize.CommonsChunkPlugin({
    name: 'vendor',
    minChunks: function (module) {
        return module.context && module.context.indexOf('node_modules') !== -1;
    }
}),
…
```

Ahora le estamos diciendo que todas las librerías que se encuentren en el directorio node_modules (las que hayamos instalado con npm) se empaqueten en el bundle vendor.bundle.js. Podemos comprobarlo abriendo los bundles commons.bundle.js y vendor.bundle.js.

Sólo nos falta vincular el nuevo bundle en el html:

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

Esta línea la añadiremos en index.html y en fecha.html justo antes de el script del bundle commons.bundle.js que vinculamos antes.

Modo producción

Existen algunas diferencias entre construir la versión para desarrollo de nuestra

aplicación y construir la versión para producción. Por ejemplo, cuando construimos la versión para desarrollo nos interesa poder depurar el código javascript, por lo que no nos convendrá tener el javascript minificado y/o comprimido, sin embargo en la versión de producción si que es interesante, ya que esto mejorará rendimiento de la aplicación.

Por este motivo, Webpack dispone de un parámetro **-p** con el que le indicamos que queremos que Webpack genere la versión para producción de nuestra aplicación.

Ejecutar el comando Webpack con el parámetro **-p** es equivalente a ejecutar el siguiente comando:

```
webpack --optimize-minimize --define process.env.NODE_ENV="'production'"
```

Esto realiza las siguientes acciones:

- Minificación de código usando el pluging UglifyJsPlugin.
- Ejecuta el pluging LoadersOptionPlugin que nos permite introducir opciones par configurar los loaders de la aplicación.
- Establece la variable de entorno NodeJS que activa determinados paquetes para compilar de forma diferente.

Estas tareas también se podrían hacer de forma manual, pero el parámetro -p es una forma sencilla de conseguir una versión de producción sin demasiados requisitos personalizados.

Y hasta aquí llega esta pequeña introducción a Webpack, por supuesto, la herramienta nos ofrece muchas más posibilidades que, por motivos obvios de tiempo, no se han podido ver en este tema, pero espero que esta introducción pueda servir de base para que podáis ampliar conocimientos por vuestra cuenta.