Browserify

Tiempo estimado: 20min

Antes de adentrarnos en React, vamos a presentar la utilidad Browserify, utilizada ampliamente en el desarrollo de software React. Groso modo, consiste en una herramienta de empaquetado de archivos.

Para comenzar, presentamos Browserify. A continuación, el concepto de módulos y sus tipos. Después, su instalación. Y finalmente, listamos algunos módulos predefinidos que vienen de fábrica con Browserify.

Al finalizar la lección, el estudiante sabrá:

- Qué es Browserify.
- Qué es un módulo.
- Cómo instalar Browserify.
- Qué herramientas de automatización proporcionan *plugins* de Browserify.
- Cuáles son los principales módulos predefinidos de Browserify.

Introducción

Browserify es un sistema de módulos, similar a NPM de Node. No es más que una aplicación que empaqueta aplicaciones JavaScript en un único archivo. En nuestro caso, la utilizaremos para generar el archivo final de nuestras aplicaciones React. Si lo deseamos, podemos utilizar webpack en vez de Browserify. Nosotros utilizaremos Browserify, por encontrarse su uso más extendido en React así como en otros *frameworks*, con sus más de dos millones de descargas mensuales.

Cuando se usa Browserify en aplicaciones React, la idea es escribirlas como si fuesen aplicaciones Node.

Se encuentra escrito en JavaScript bajo la plataforma Node. Su sitio web oficial es browserify.org. Y se puede utilizar gratuitamente.

Lo que hace Browserify es generar un archivo empaquetado (*bundle file*) que contiene el código de la aplicación y cualquier otro módulo externo que use o del que dependa. Es importante tener claro que el archivo está autocontenido, contiene tanto el código específico del componente como aquello que él utiliza, sin necesidad de acceder a módulos externos. De esta manera, podemos utilizarlo en el navegador importándolo o cargándolo mediante un único elemento HTML <script>.

Browserify se puede utilizar para empaquetar tanto aplicaciones *frontend* como *backend*. En nuestro caso, nos centraremos en aplicaciones o módulos webs a ejecutar en el navegador.

Módulos

Un módulo (*module*) es un conjunto de uno o más archivos que implementan un determinado componente JavaScript, el cual se puede reutilizar en distintas aplicaciones y otros componentes. Todo módulo implementa una o más tareas específicas y relacionadas. Y presentan una API. Son elementos que facilitan la organización del código y su reutilización. Esto ayuda a administrar el código fuente de la aplicación, haciéndolo más fácil y eficiente.

Browserify puede trabajar con módulos implementados mediante el sistema de paquetes de Node. Por lo que si el estudiante sabe desarrollar un módulo en Node, no tendrá problemas para trabajar con Browserify. Es más, muchos módulos publicados en NPM se pueden utilizar tanto en aplicaciones Node como en aplicaciones React, o sea, en el navegador, gracias al uso de Browserify. Así, encontramos Angular, Ember, Handlebars, jQuery y React.

Archivo principal

El archivo principal (*main file*) o punto de entrada (*entry point*), según la jerga de Browserify, es el

archivo en el que comienza la carga y ejecución del módulo. El archivo en el que comenzar la carga de la aplicación o componente. Recordemos, en Node, este archivo se indica mediante la propiedad main del archivo package.json.

Este archivo es de vital importancia para Browserify. Lo que hace es recorrerlo buscando las importaciones que se hacen en él para así comenzar a generar lo que se conoce como grafo de dependencias (*dependency graph*), el conjunto de módulos utilizados por el componente. Trabaja recursivamente. Cada vez que se encuentra con una importación, también la recorre añadiendo al grafo todo módulo utilizado directa o indirectamente. Cuando se queda sin nada que recorrer, o sea, ha recorrido todas las importaciones, entonces tiene el grafo que necesita y pasa a generar el archivo empaquetado con el código específico del componente y el de los módulos importados por él y por sus módulos importados.

Como sabemos, Browserify se puede utilizar tanto para componentes *frontend* como *backend*. Si lo deseamos, podemos hacer que un módulo disponga de archivos de entrada distintos para cada lado. Para ello, se utiliza las propiedades main y browser del archivo package.json. main se utiliza como punto de entrada predeterminado para ambos extremos. Pero si deseamos indicar uno específico para el navegador, lo indicaremos en la propiedad browser. Cuando el componente se ejecute en el navegador, Browserify usará como archivo de entrada el indicado en la propiedad browser; mientras que si se ejecuta en el servidor, main. Si la propiedad browser no existe, en el navegador se utilizará main.

Definición de la API

Una API (*Application Programming Interface*, Interfaz de Programación de Aplicaciones) es un conjunto de objetos que ofrece un paquete, un módulo o una biblioteca para su reutilización, en nuestro caso, para su reutilización por aplicaciones u otros módulos. Recordemos que todo módulo tiene su propio espacio de nombres privado. Cada vez que definimos una variable o un objeto como, por ejemplo, una función a nivel de módulo, esta variable se define como una variable local del módulo. Es necesario, pues, definir el conjunto de objetos que expone el módulo al exterior, formando su API y que pueden utilizar las aplicaciones u otros módulos.

Actualmente, se recomienda encarecidamente desarrollar las aplicaciones React mediante la especificación ES2015, también conocida como ES6, o superior. Así pues, el espacio de nombres privado de un módulo estará formado por todos aquellos objetos que *no* se definan mediante la sentencia export. Por su parte, la API del módulo, su parte pública, estará formada por todo aquello definido con la sentencia export.

Si lo deseamos, también podemos fijar la API mediante la propiedad exports del objeto module que tiene asociado de manera implícita todo módulo.

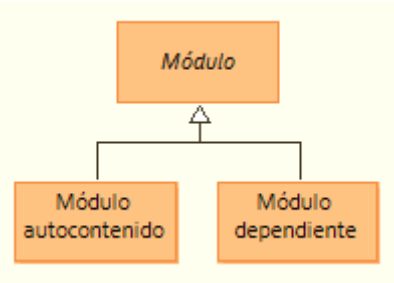
Espacio de nombres privado

Un espacio de nombres (*namespace*) es una tabla de símbolos formada por pares nombre-objeto. Cada nombre es el nombre de una variable. Y su valor, el objeto referenciado por la variable. Se puede distinguir entre espacio de nombres global y espacio de nombres privado.

Existe un único espacio de nombres global (*global namespace*) en el que se alojan los objetos que pueden ser accedidos por todos los módulos. Mientras que cada módulo tiene asociado su propio espacio de nombres privado (*private namespace*), es decir, la tabla de símbolos en la que se almacena las variables privadas del módulo y los objetos definidos en él, ajena a la global y a la del resto de módulos. Esto permite que dos o más módulos tengan objetos homónimos en sus respectivos espacios de nombres privados.

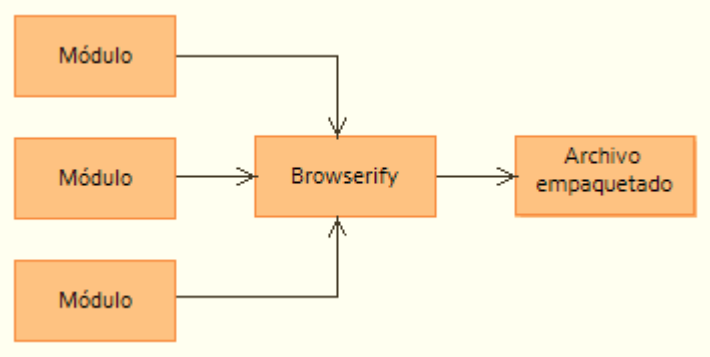
Módulos dependientes

Básicamente, se puede distinguir dos tipos de módulos, los autocontenidos y los dependientes.



Un módulo autocontenido (*self-contained module*) es aquel que no depende de ningún otro módulo. Mientras que un módulo dependiente (*dependent module*) es aquel que utiliza otros módulos.

Una vez implementada la aplicación, lo que hacemos es crear un archivo empaquetado (*bundle file* o *browserify file*) que contenga el código específico de la aplicación y de los módulos externos utilizados por ella. Así, cuando incorporemos el archivo empaquetado a la página web mediante un elemento <script>, lo que estaremos haciendo es cargar tanto el código específico de la aplicación como el externo.



Cuando se crea un módulo autocontenido, Browserify generará un archivo empaquetado sólo con el código del módulo, pues no necesita más. En cambio, cuando creamos un módulo dependiente, como es el caso de una aplicación React, Browserify añadirá al archivo empaquetado tanto el código de la aplicación o componente React como aquello importado. Asegurando así que el archivo empaquetado es autocontenido. Contiene todo: la parte particular y lo importado.

Importación de módulos

Como ya sabemos de JavaScript, para utilizar un módulo, hay que importarlo. La importación (*import*) o inclusión (*include*) es la operación mediante la cual incorporamos un módulo a nuestra aplicación u otro módulo.

En React, utilizaremos la especificación ES2015, por lo que la importación la realizaremos mediante la sentencia import. Pero también se puede utilizar la función require() de Node.

Para concretar el funcionamiento de Browserify, lo que hace es recorrer las importaciones que llevamos a cabo en la aplicación. Cuando importamos un módulo externo, lo añade a su lista. Una vez recorrido todos los módulos de la aplicación, generará un archivo. Este archivo empaquetado contendrá: por un lado, los módulos específicos de la aplicación, o sea, los que se han desarrollado particularmente para ella; y por otro lado, todos aquellos módulos externos utilizados por la aplicación. Todo ello, en un único archivo, haciéndolo autocontenido.

Así, por ejemplo, cuando utilicemos el módulo events, que viene de fábrica con Browserify y simula el módulo homónimo de Node, Browserify lo detectará en su recorrido por el código fuente de la aplicación y lo añadirá automáticamente al archivo empaquetado. Pero ojo, si no lo usamos, no lo añadirá.

Instalación de Browserify

Para instalar Browserify en nuestra máquina, hay que utilizar la utilidad npm de Node e instalar el paquete browserify, recomendándose una instalación global:

```
npm install -g browserify
```


Una vez instalado, se recomienda comprobar que la utilidad browserify es accesible. Para ello, se suele comprobar la versión instalada:

```
browserify --version
```

browserify

Una vez instalado Browserify, hay que utilizar la aplicación de línea de comandos browserify para generar el archivo empaquetado. Para conocer la lista de opciones del comando, utilizar la opción --help o -h:

```
browserify -h
```

Generación del archivo empaquetado

Una vez redactado el módulo y antes de poder utilizarlo en el navegador, es necesario obtener el archivo empaquetado. Para ello, tenemos que ejecutar el comando browserify para generarlo, recordemos, cargando en él tanto el código específico de la aplicación o componente como aquellos utilizados por ella.

Para generar el archivo empaquetado, utilizaremos la siguiente sintaxis de browserify:

```
browserify archivoPrincipal [opciones]
```

Entre las opciones más utilizadas encontramos:

Opción	Descripción
--outfile archivo -o archivo	Ruta del archivo empaquetado a generar.
--require módulo -r módulo	Módulos a incluir.
--ignore archivo -i archivo	Archivo a ignorar.
--exclude archivo -e archivo	Archivo a excluir del archivo empaquetado.
--standalone nombre -s nombre	Nombre del módulo de cara al navegador.
--transform módulo -t módulo	Módulo que invocará previamente browserify para obtener los archivos finales con los que debe trabajar.

¿Cuál es la diferencia entre archivos ignorados y excluidos? Ambos omiten un módulo o archivo, pero lo hacen de manera distinta. Browserify sustituye los archivos ignorados (*ignored files*) por objetos vacíos en el archivo empaquetado. Cuando lo importemos, obtendremos un objeto vacío. En cambio, un archivo excluido (*excluded file*) es aquel que no se añade al archivo empaquetado. Si lo intentamos importar, obtendremos un error.

He aquí un ejemplo ilustrativo que muestra cómo crear el archivo empaquetado myapp.js a partir de los archivos .js del directorio actual, excluyendo archivo.js:

```
browserify -t babelify *.js -e archivo.js -o miapp.js
```

En el ejemplo anterior, se asume que los archivos de JavaScript usan la especificación ES6, utilizándose babelify para convertirlos a la especificación ES5 entendida por todos los motores de JavaScript.

La opción --standalone se utiliza cuando se está generando un módulo que se ejecutará en el navegador. Tengamos en cuenta que cuando generamos un archivo empaquetado, éste tendrá una API. Si lo utilizamos en Node, lo haremos tal cual. Como cualquier otro módulo, salvando que todo su código se encuentra en un único archivo. Pero si lo usamos en un navegador, ¿cómo accedemos al módulo y los objetos exportados por ella, o sea, a su API? Con --standalone lo que hacemos es fijar el nombre de la variable global que deseamos se cree en el navegador cuando importemos el módulo con el elemento <script>. Si por ejemplo fijamos el nombre mymod, Browserify añadirá el código necesario al archivo empaquetado para que detecte que el archivo se está ejecutando en un navegador y, entonces, cree una variable global con ese nombre. Así pues, si el módulo expone una función, digamos fn, para acceder a ella en el navegador usaremos mymod.fn.

En la opción --standalone, podemos usar nombres específicos o de espacios de nombres como, por

ejemplo, a, a.b y a.b.c.

babelify

Por convenio, las aplicaciones React se desarrollan usando la especificación ES2015 o superior de JavaScript. Esto hace que no se puedan ejecutar en todos los navegadores directamente, por lo que habrá que compilarlas, recomendándose Babel como *transpiler*. Es importante tener en cuenta que Browserify no es un *transpiler* de JavaScript. Sólo es un generador de archivos empaquetados. Recordemos que Browserify recorre los archivos en busca de los módulos importados para, de esta manera, extraer los módulos externos e incorporarlos al archivo empaquetado. Por lo que tendremos que dárselos bajo una especificación JavaScript con la que sepa trabajar.

Para Browserify, un transformador (*transformer*) es un componente que procesa, previamente a su procesamiento por browserify, un archivo JavaScript. En nuestro caso, utilizaremos babelify como transformador. Cuando Browserify lee un archivo, primero se lo pasa a babelify para que le devuelva el archivo bajo una especificación con la que pueda trabajar. A continuación, recorre este código generado y va añadiendo los módulos utilizados al grafo de dependencias.

Mediante los transformadores, se puede extender la funcionalidad de Browserify.

Los transformadores se fijan mediante las opciones --transform o -t. En nuestro caso, usaremos:

```
-t babelify
```

Para poder utilizarlo, es necesario instalarlo mediante npm, recomendándose una instalación local a la aplicación React. O sea, hay que añadirlo a la propiedad devDependencies del archivo package.json de la aplicación.

Si lo deseamos, podemos utilizar la propiedad browserify del archivo package.json para indicar los transformadores:

```
browserify: {
  transform: ["babelify"]
}
```

Módulos predefinidos

Browserify viene de fábrica con varios módulos que podemos reutilizar en nuestros propios proyectos, conocidos formalmente como módulos predefinidos (*built-in modules*). Algunos son implementaciones, a nivel de navegador, de módulos de Node, con el objeto de poder utilizarlos también en el navegador con su misma API, ayudándonos así a reducir la curva de aprendizaje.

A continuación, se muestra la lista de los módulos predefinidos más importantes:

Módulo	Descripción
assert	Módulo de aserción.
buffer	Implementación del módulo buffer de Node.
console	Implementación del objeto console.
crypto	Implementación del módulo crypto de Node.
events	Implementación de la clase EventEmitter de Node.
http	Implementación del módulo http de Node.
https	Implementación del módulo https de Node.
os	Implementación del módulo os de Node.
path	Implementación del módulo path de Node.
querystring	Implementación del módulo querystring de Node.
stream	Implementación del módulo stream de Node.
timers	Implementación del módulo timers de Node.
tty	Implementación del módulo tty de Node.
url	Implementación del módulo url de Node.

util	Implementación del módulo util de Node.
vm	Implementación del módulo vm de Node.
zlib	Implementación del módulo zlib de Node.

Cuando se utiliza uno de estos módulos, browserify añade su código al archivo empaquetado, permitiendo así su uso.

Si estamos empaquetando un módulo que sabemos que se usará sólo en Node, se recomienda usar la opción --no-builtins de browserify para que no añada ningún módulo predefinido al empaquetado. Tengamos en cuenta que estos módulos se añaden para su uso en el navegador, no en Node, pues Node ya los tiene.

Por otra parte, si usamos las variables global, process, Buffer, window, __filename y __dirname, éstas serán procesadas internamente por Browserify, añadiendo sus propias definiciones. Por ejemplo, en el navegador, global se implementa como un alias de window; y process define la función nextTick() al igual que en Node. Con respecto a las variables globales process, global, __filename y __dirname, browserify proporciona varias opciones de línea de comandos:

- --insert-globals, --ig o --fast. Añade la definición particular de estas variables para su uso en el navegador, tanto si son accedidas en el componente empaquetado como si no lo son.
- --insert-globals-vars o --igv. Indica una lista separada por comas de las variables a detectar y definir si es necesario.
- --detect-globals o --dg. Indica que se detecte las variables globales y se defina aquellas que sean accedidas.

Plugins de automatización

La automatización (*automation*) es el proceso mediante el cual se convierte tareas manuales en automáticas. Al automatizar las tareas, las podemos ejecutar más fácil y rápidamente que si tuviéramos que hacerlo una y otra vez manualmente.

La automatización se lleva a cabo mediante herramientas de automatización (*automation tools*), aplicaciones de software que permiten convertir e implementar tareas manuales repetitivas en automáticas. Entre otras, tenemos Ansible, Grunt, Gulp, Justo.js o Puppet, cada una de ellas orientadas a un determinado mercado. Entre ellas, Grunt, Gulp y Justo.js disponen de *plugins* que permiten automatizar la ejecución de Browserify.

A lo largo del presente curso, se usará Justo.js.

