Módulos

Tiempo estimado: 20min

Uno de los elementos más importantes de Node es el módulo. A través de los módulos, definimos piezas de código reutilizable. Otro de los elementos de reutilización es el paquete. En esta lección, exploramos los módulos detenidamente. Y dejamos los paquetes para una posterior.

La lección comienza introduciendo, por un lado, el concepto de reutilización de software, indispensable en el desarrollo de software desde hace muchísimos años y, por otro lado, el concepto de módulo en Node. A continuación, se muestra cómo podemos definir nuestros propios módulos. Después, se presenta los medios a través de los cuales podemos reutilizarlos en nuestros programas. Y finalmente, se describe el concepto de módulo principal y cómo determinar si estamos ante él o ante un módulo secundario.

Al finalizar la lección, el estudiante sabrá:

- Qué es un módulo.
- Cómo definir módulos.
- Cómo exponer la API reutilizable de los módulos.
- Cómo utilizar o importar módulos.
- Cómo se realiza la búsqueda de los módulos.
- Qué es la caché de módulos.
- Cómo determinar si estamos ante el módulo principal.

Introducción

Mediante la reutilización de software (*software resuse*) o reutilización de código (*code resuse*), se usa código ya desarrollado por nosotros mismos u otros en un determinado proyecto. Las principales razones por las que se recomienda su uso son:

- Aumento de la productividad.
- Reducción de costes.
- Reducción de los tiempos de desarrollo.
- Mejora y facilitación de las pruebas de unidad.
- Desarrollo de software más robusto.

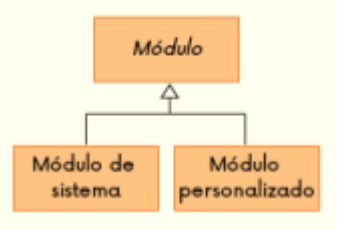
En Node, encontramos que la columna vertebral de la reutilización son los módulos y los paquetes. Un módulo (*module*) no es más que un archivo de código JavaScript que tiene asociado una determinada funcionalidad reutilizable, bien clara y definida. Y presenta una API que expone sus elementos reutilizables.

Para usar un módulo, se puede utilizar función require() o la sentencia import de la especificación ES2015 de JavaScript.

Ni los módulos ni los paquetes son conceptos específicos de JavaScript o Node. Los utilizan muchos otros lenguajes. La cuestión es cómo se hace en Node. En esta lección, vamos a aprender a desarrollar nuestros propios módulos y, en una posterior, atenderemos más detenidamente el concepto de paquete.

Módulos personalizados

Tal como acabamos de ver, un módulo no es más que un archivo de código JavaScript, que presenta objetos reutilizables. En Node, podemos distinguir básicamente dos tipos de módulos, los de sistema y los personalizados.



Un módulo de sistema (*system module* o *core module*) o módulo integrado (*built-in module*) es aquel que viene de fábrica con Node. Mientras que un módulo personalizado (*custom module*) o módulo de usuario (*user module*) es aquel que no viene de fábrica con Node y, por lo tanto, tenemos que desarrollarlo de manera específica para el proyecto o instalarlo para su utilización si ha sido desarrollado por otros.

La lista de módulos de sistema se puede encontrar en nodejs.org/dist/latest/docs/api/. Hay módulos muy variados como, por ejemplo:

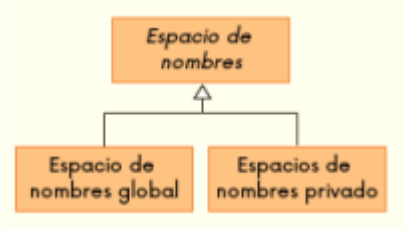
- dns para resolver nombres de máquinas y direcciones de IP.
- events para soportar el modelo de eventos de Node.
- fs para acceder al sistema de ficheros.
- http y https para realizar peticiones HTTP y HTTPS.
- net para abrir conexiones de red.
- os para obtener información del sistema operativo y la máquina.
- zlib para compilar archivos.

En esta sección, vamos a centrarnos en la definición de nuestros propios módulos. Para definir un módulo personalizado no tenemos más que crear un archivo JavaScript, con la extensión .js. Y hay que tener en cuenta que tienen principalmente:

- Un espacio de nombres privado.
- Una API a través de la cual acceder a su funcionalidad reutilizable.

Espacio de nombres privado

Un espacio de nombres (*namespace*) es una tabla de símbolos formado 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.



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

En Node, toda variable, función o clase se define dentro del espacio de nombres privado del módulo.

Variables privadas predefinidas

Todo módulo tiene definidas las siguientes variables privadas:

- global (object). Objeto que contiene el espacio de nombres privado del módulo.

Ojo, *no* contiene el contexto global de la aplicación. Sino el contexto privado del módulo.

- module (object). Objeto que representa el propio módulo.
- exports (object). Objeto que contiene la API del módulo, o sea, sus objetos reutilizables.

  Concretamente, esta variable contiene el valor de la propiedad module.exports.
- __dirname (string). Contiene la ruta al directorio en el que se encuentra el módulo.
- __filename (string). Contiene el nombre del archivo módulo. Al igual que module.filename.

Variable privada module

Node crea la variable privada module automáticamente en cada módulo. Es un objeto que representa al propio módulo. Contiene las siguientes propiedades:

- id (string). Ruta al archivo módulo.
- filename (string). Nombre del archivo módulo.
- loaded (boolean). ¿Se encuentra *ya* cargado el módulo en la caché?
- parent (Module). Módulo que realizó la primera importación del módulo y, por ende, el responsable de que se cargara en la caché del motor de Node.
- children (Module[]). Módulos que ha importado este módulo.

API del módulo

Una API (*Application Programming Interface*, Interfaz de Programación de Aplicaciones) es un conjunto de objetos que ofrece un módulo para su reutilización, en el caso de Node, para su reutilización por otros módulos y/o paquetes. En Node, 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 los demás módulos.

Variable privada exports

Node define automáticamente una variable privada exports en cada módulo. Esta variable es local al módulo y define su API. Cada vez que importamos un módulo, lo que recibimos es el valor de su variable privada exports. Ésta es la razón por la que se considera que las propiedades de este objeto forman la API del módulo. Por lo tanto, si nuestro módulo define, por ejemplo, tres funciones y sólo deseamos exportar dos, habrá que asignar estas dos como propiedades del objeto exports.

Supongamos que deseamos definir un módulo que realiza las operaciones aritméticas de suma y resta. Si deseamos exportar o definir la API con las funciones suma() y resta(), podríamos hacer lo siguiente:

```
//api
exports.suma = suma;
exports.resta = resta;

//operación de suma
function suma(x, y) {
  return x + y;
}

//operación de resta
function resta(x, y) {
  return x - y;
}
```

Propiedad module.exports

De manera predeterminada, la variable exports contiene el valor de la propiedad exports del objeto module. Es recomendable que la propiedad module.exports y la variable privada exports referencien *siempre* al mismo objeto. Así, el ejemplo anterior debería definirse como sigue:

```
//api
module.exports = exports = {};
```



```
exports.suma = suma;
exports.resta = resta;

//operación de suma
function suma(x, y) {
  return x + y;
}

//operación de resta
function resta(x, y) {
  return x - y;
}
```

Sentencia export

Cuando se desarrolla en Node, se recomienda hacerlo mediante la especificación ES2015 o superior. En estos casos, se prefiere el uso de la sentencia export para definir la API de los módulos. El *transpiler* convertirá la sentencia export a la correspondiente proposición que ataca a la propiedad exports del módulo. Así, por ejemplo, lo siguiente es un ejemplo, bajo la especificación ES2015, del ejemplo anterior:

```
//operación de suma
export function suma(x, y) {
  return x + y;
}

//operación de resta
export function resta(x, y) {
  return x - y;
}
```

La API del módulo está formada por aquellos objetos definidos mediante la sentencia export.

Importación de módulos

Para utilizar un módulo, hay que realizar una importación (*import*), petición al motor de Node para que busque el módulo en nuestra instalación, lo cargue y, a continuación, devuelva su objeto API con el que usarlo en nuestro código. Para este fin, hay que utilizar la función global require(). También es posible utilizar la sentencia import de la especificación ES2015 de JavaScript.

El proceso de importación de un módulo es como sigue:

1. Consulta de la caché de módulos cargados.
2. Si no se encuentra en la caché, búsqueda del módulo en el sistema.
3. Si se encuentra, carga del módulo en la caché para su utilización y así evitar que próximas importaciones requieran su carga de nuevo.
4. Devolución del objeto que contiene la API del paquete.

Función global require()

La función global require() se utiliza para importar un determinado módulo. Su signatura es la siguiente:

```
function require(id) : object
```

Parámetro	Tipo de datos	Descripción
id	String	Nombre del módulo a importar.

Si la función no lo encuentra, propagará un Error donde su propiedad code contendrá MODULE_NOT_FOUND.

Esta función importa el módulo en el sistema y devuelve el objeto API que debemos utilizar para reutilizar su código. A continuación, se muestra un ejemplo con el que importar el módulo integrado fs:

```
//importación anterior a ES2015
const fs = require("fs");

//importación si sentencia import, de ES2015, soportada
import fs from "fs";
```


Por convenio y buenas prácticas, se recomienda que el objeto devuelto por la función require() se asigne a una constante cuyo nombre sea el mismo que el del módulo importado, siempre que sea posible.

Una vez importado, podemos utilizar su objeto API para acceder a los elementos reutilizables. He aquí un ejemplo ilustrativo de cómo usar la función readFileSync() del módulo fs para obtener el contenido de un archivo:

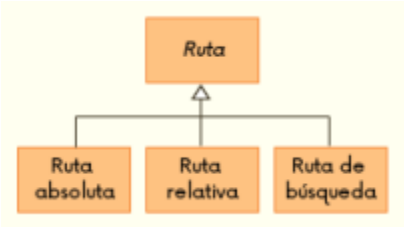
```
//importación
const fs = require("fs");

//uso
var contenido = fs.readFileSync("/ruta/al/archivo.txt");
```

Búsqueda de módulo

Para encontrar los módulos, el cargador realiza lo que se conoce como búsqueda de módulo (*module lookup*). Consiste en hallar la ubicación del módulo en el disco, atendiendo a lo indicado por el usuario en la función require(). Para ello, se sirve de la variable de entorno NODE_PATH, que contiene la lista de directorios en los que buscar.

La ruta de importación (*import path*) es la ruta que se especifica en la función require() y que utiliza Node para determinar dónde debe buscar el módulo, en caso de que *no* se haya cargado todavía en la caché. Se distingue tres tipos de ruta: absoluta, relativa y de búsqueda.



Veamos cada una de ellas detenidamente.

Ruta absoluta

Una ruta absoluta (*absolute path*) es aquella que comienza por una barra (/) e indica la ubicación exacta del módulo en disco, comenzando en la raíz. Es muy raro el uso de este tipo de rutas, pero muy útil para algunos proyectos.

Ejemplo:

```
const calcul = require("/opt/node/lib/calcul");
```

Ruta relativa

Una ruta relativa (*relative path*) es aquella que es relativa al directorio actual, a aquel desde el que realizamos la importación. Puede comenzar en el directorio actual mediante un punto y la barra (./) o bien en el directorio padre mediante dos puntos y una barra (../).

Veamos unos ejemplos ilustrativos:

```
var calcul = require("./lib/calcul");  //desde el directorio actual
var calcul = require("../lib/calcul"); //desde el directorio padre al actual
```

Ruta de búsqueda

Una ruta de búsqueda (*search path*) indica únicamente un nombre del módulo. Nada más. En este caso, el sistema sigue un proceso de búsqueda hasta que se hace con él:

1. Realiza una búsqueda entre los módulos integrados o de sistema, recordemos, aquellos que vienen de fábrica con Node.
2. Si no se encuentra en el paso anterior, pasa a buscarlos en la carpeta node_modules del directorio en el que se encuentra el módulo desde el que se hace la importación.
3. Si no se encuentra en el paso anterior, se busca en la carpeta node_modules del directorio padre. Así hasta alcanzar el directorio raíz del disco.

4. Si no se encuentra en el paso anterior, se busca en los directorios indicados en la variable de entorno NODE_PATH.

Con este procedimiento, los módulos integrados tienen prioridad sobre los demás. A continuación, los locales al proyecto. Mientras que los paquetes, ubicados en los directorios indicados en la variable de entorno NODE_PATH, son los últimos donde se busca.

Por ejemplo, para la siguiente importación:

```
const lodash = require("lodash");
```

Suponiendo que el directorio actual es /home/me/myapp, la anterior importación consistiría en:

1. Búsqueda de lodash entre los módulos integrados.
2. Búsqueda de lodash en el directorio /home/me/myapp/node_modules/.
3. Búsqueda de lodash en los directorios anteriores hasta la raíz:
   i. /home/me/node_modules/.
   ii. /home/node_modules/.
   iii. /node_modules/.
4. Búsqueda de lodash en los directorios indicados en la variable de entorno NODE_PATH.

Caché de módulos

La caché de módulos (*module cache*) es un componente interno del motor de Node que contiene los paquetes y módulos importados por la función require(). Cuando se solicita una importación, require() busca primero el módulo en la caché. Si lo encuentra, usa el objeto API que obtuvo cuando lo cargó y lo devuelve. No lo vuelve a cargar. En cambio, si no lo encuentra, realiza una carga del módulo, para ello, lo busca, lo lee y lo ejecuta. Una vez ejecutado, registra una entrada para ese módulo en la caché y le asocia su objeto API correspondiente. Y entonces, lo devuelve.

Es importante tener claro que los módulos sólo se cargan una única vez. Pero se pueden importar y usar tantas como sea necesario.

Carga de módulo

Tal y como acabamos de mostrar, el primer paso para poder reutilizar un paquete o módulo es encontrarlo y cargarlo. Al proceso mediante el cual Node lee su contenido, se le conoce formalmente como carga de módulo (*module load*). Durante esta carga, Node crea un objeto Module en el que el paquete o módulo cargará su API, que es lo que finalmente devolverá require(). Las cargas las realiza un componente interno de Node conocido como cargador de módulos (*module loader*).

Supresión de módulo de la caché

Para suprimir un módulo de la caché, hay que utilizar la sentencia delete de JavaScript. He aquí un ejemplo ilustrativo:

```
delete require("módulo");
```

Al suprimirlo de la caché, la próxima vez que se importe, el motor de Node tendrá que buscarlo y cargarlo.

Módulo principal

El módulo principal (*main module*) es aquel que ejecuta node cuando lo invocamos. O sea, el punto de entrada a nuestra aplicación o paquete. En resumen, el que pasamos al comando node en la línea de comandos. Cualquier otro módulo se conoce formalmente como módulo secundario (*secondary module*).

Propiedad require.main

En algunas ocasiones, es necesario saber si estamos ante el módulo principal o, por el contrario, ante uno secundario. Para ayudarnos, Node asigna el objeto módulo del módulo principal a la propiedad require.main. Así pues, si se da el caso en el que necesitamos saber si nos encontramos en el módulo principal o bien el módulo ha sido importado como secundario, se puede usar el operador de

comparación estricta (===) para comprobarlo fácilmente. He aquí un sencillo ejemplo ilustrativo:

```
if (require.main === module) {
  //código a ejecutar si el módulo actual es el principal
} else {
  //código a ejecutar si el módulo actual es secundario
}
```

