Usando nuestros propios módulos

La instrucción `require` que hemos utilizado en el documento anterior para utilizar módulos incorporados en el núcleo de Node.js también debe utilizarse para incluir un archivo nuestro dentro de otro, de forma que podemos (debemos) descomponer nuestra aplicación en diferentes archivos, lo que la hará más fácil de mantener.

1. Un ejemplo sencillo

Por ejemplo, vamos a crear un proyecto llamado "*PruebasRequire*" en nuestra carpeta de "*Pruebas*", con dos archivos de momento: uno llamado `principal.js` que tendrá el código principal de funcionamiento del programa, y otro llamado `utilidades.js` con una serie de funciones o propiedades auxiliares. Desde el archivo `principal.js`, podemos incluir el de utilidades con la instrucción `require`, del mismo modo que lo hicimos antes, pero indicando la ruta relativa del archivo a incluir. En este caso quedaría así:

```
const utilidades = require('./utilidades.js');
```

Es posible también suprimir la extensión del archivo en el caso de archivos JavaScript, por lo que la instrucción anterior sería equivalente a esta otra (Node sobreentiende que se trata de un archivo JavaScript):

```
const utilidades = require('./utilidades');
```

El contenido del archivo `utilidades.js` debe tener una estructura determinada. Si, por ejemplo, el archivo tiene este contenido:

```
console.log('Entrando en utilidades.js');
```

Entonces el mero hecho de incluirlo con `require` mostrará por pantalla el mensaje "Entrando en utilidades.js" al ejecutar la aplicación. Es decir, cualquier instrucción directa que contenga el archivo incluido se ejecuta al incluirlo. Lo normal, no obstante, es que este archivo no contenga instrucciones directas, sino una serie de propiedades y métodos que puedan ser accesibles desde el archivo que lo incluye.

Vamos a ello, supongamos que el archivo `utilidades.js` tiene unas funciones matemáticas para sumar y restar dos números y devolver el resultado. Algo así:

```
let sumar = (num1, num2) => num1 + num2;
let restar = (num1, num2) => num1 - num2;
```

Lo lógico sería pensar que, al incluir este archivo con `require` desde `principal.js`, podamos acceder a las funciones `sumar` y `restar` que hemos definido... pero no es así.

1.1. Exportando contenido con *module.exports*

Para poder hacer los métodos o propiedades de un archivo visibles desde otro que lo incluya, debemos añadirlos como elementos del objeto `module.exports`. Así, las dos funciones anteriores se deberían definir de esta forma:

```
module.exports.sumar = (num1, num2) => num1 + num2;
module.exports.restar = (num1, num2) => num1 - num2;
```

Es habitual definir un objeto en `module.exports`, y añadir dentro todo lo que queramos exportar. De esta forma, tendremos por un lado el código de nuestro módulo, y por otro la parte que queremos exportar. El módulo quedaría así, en este caso:

```
let sumar = (num1, num2) => num1 + num2;
let restar = (num1, num2) => num1 - num2;

module.exports = {
    sumar: sumar,
    restar: restar
};
```

En cualquier caso, ahora sí podemos utilizar estas funciones desde el programa `principal.js`:

```
const utilidades = require('./utilidades');
console.log(utilidades.sumar(3, 2));
```

Notar que el objeto `module.exports` admite tanto funciones como atributos o propiedades. Por ejemplo, podríamos definir una propiedad para almacenar el valor del número "pi":

```
module.exports = {
    pi: 3.1416,
    sumar: sumar,
    restar: restar
};
```

... y acceder a ella desde el módulo principal:

```
console.log(utilidades.pi);
```

2. Incluir carpetas enteras

En el caso de que nuestro proyecto contenga varios módulos, es recomendable agruparlos en carpetas, y en este caso es posible incluir una carpeta entera de módulos, siguiendo una nomenclatura específica. Los pasos a seguir son:

- Añadir todos los módulos (ficheros *.js*) que queramos dentro de la carpeta deseada
- Crear en esa carpeta un archivo llamado `index.js`. Este será el archivo que se incluirá en nombre de toda la carpeta
- Dentro de este archivo `index.js`, incluir (con `require`) todos los demás módulos de la carpeta, y exportar lo que se considere.

Desde el programa principal (u otro lugar que necesite incluir la carpeta entera), incluir el nombre de la carpeta. Automáticamente se localizará e incluirá el archivo `index.js`, con todos los módulos que éste haya incluido dentro.

Veamos un ejemplo: vamos a nuestra carpeta "*PruebasRequire*" creada en el ejemplo anterior, y crea una carpeta llamada "*idiomas*". Dentro, crea estos tres archivos, con el siguiente contenido:

Archivo *es.js*

```
module.exports = {
    saludo : "Hola"
};
```

Archivo *en.js*

```
module.exports = {
    saludo : "Hello"
};
```

Archivo *index.js*

```
const en = require('./en');
const es = require('./es');

module.exports = {
    es : es,
    en : en
};
```

Ahora, en la carpeta raíz de "*PruebasRequire*" crea un archivo llamado `saludo_idioma.js`, con este contenido:

```
const idiomas = require('./idiomas');

console.log("English:", idiomas.en.saludo);
console.log("Español:", idiomas.es.saludo);
```

Como puedes ver, desde el archivo principal sólo hemos incluido la carpeta, y con eso automáticamente incluimos el archivo `index.js` que, a su vez, incluye a los demás. Una vez hecho esto, y tal y como hemos exportado las propiedades en `index.js`, podemos acceder al saludo en cada idioma.

3. Incluir archivos JSON

Los archivos JSON son especialmente útiles, como veremos, para definir cierta configuración básica (no encriptada) en las aplicaciones, además de para enviar información entre partes de la aplicación (lo que veremos también más adelante). Por ejemplo, y siguiendo con el ejemplo anterior, podríamos sacar a un archivo JSON el texto del saludo en cada idioma. Añadamos un archivo llamado `saludos.json` dentro de nuestra subcarpeta "idiomas":

```
{
    "es" : "Hola",
    "en" : "Hello"
}
```

Después, podemos modificar el contenido de los archivos `es.js` y `en.js` para que no pongan literalmente el texto, sino que lo cojan del archivo JSON, incluyéndolo. Nos quedarían así:

Archivo *es.js*:

```
const textos = require('./saludos.json');

module.exports = {
    saludo : textos.es
};
```

Archivo *en.js*:

```
const textos = require('./saludos.json');

module.exports = {
    saludo : textos.en
};
```

La forma de acceder a los textos desde el programa principal no cambia, sólo lo ha hecho la forma de almacenarlos, que queda centralizada en un archivo JSON, en lugar de en múltiples archivos Javascript. De este modo, ante cualquier errata o actualización, sólo tenemos que modificar el texto en el archivo JSON y no ir buscando archivo por archivo. Además, nos evita el problema de las *magic strings* (cadenas que los programadores ponen a mano donde toca, suponiendo que están bien escritas y que no van a hacer falta desde otra parte de la aplicación).

4. Más sobre inclusión de módulos locales

Para finalizar con este subapartado de inclusión de módulos locales de nuestra aplicación (o división de nuestra aplicación en diversos ficheros fuente, según cómo queramos verlo), conviene tener en cuenta un par de matices adicionales:

4.1. Rutas relativas y *require*

Hasta ahora, cuando hemos empleado la instrucción `require` para incluir un módulo de nuestro propio proyecto, hemos partido de la carpeta actual. Por ejemplo:

```
const utilidades = require('./utilidades');
```

Este código funcionará siempre que ejecutemos la aplicación Node desde su misma carpeta:

```
node principal.js
```

Pero si estamos en otra carpeta y ejecutamos la aplicación desde allí...

```
node /Users/nacho/Proyectos/PruebasRequire/principal.js
```

... entonces `require` hará referencia a la carpeta desde donde estamos ejecutando, y no encontrará el archivo "*utilidades.js*", en este caso. Para evitar este problema, podemos emplear la propiedad `__dirname`, que hace referencia a la carpeta del módulo que se está ejecutando (`principal.js`, en este caso):

```
const utilidades = require(__dirname + '/utilidades');
```

4.2. Sobre *module.exports*

Quizá algunos de vosotros os habréis preguntado... ¿cómo es que puedo tener accesibles variables o métodos que yo no he definido al empezar, como `require`, o `module.exports`?

Cuando se ejecuta nuestro código, Node.js lo encapsula dentro de una función, y le pasa como parámetros los elementos externos que puede necesitar, como por ejemplo `require`, o `module` (en cuyo interior encontraremos `exports`). Sin embargo, en algunos ejemplos en Internet también podemos encontrar que se hace uso de una propiedad `exports`, en lugar de `module.exports`. Entonces...

- ¿Hay un `exports` por un lado y un `module.exports` por otro? La respuesta es que SI
- ¿Existe diferencia entre ambos? La respuesta también es que SI. A priori, ambos elementos apuntan al mismo objeto en memoria, es decir, `exports` es un atajo para no tener que escribir `module.exports`. Pero...
    - Si cometemos el error de reasignar la variable (por ejemplo, haciendo `exports = a`), entonces las referencias dejan de ser iguales.
    - Por otra parte, si queréis ver el código fuente de la función `require`, veréis que lo que devuelve es `module.exports`, por lo que, en caso de reasignar la variable `exports`, no nos serviría de nada.

La moraleja de todo esto es que, en principio, `exports` y `module.exports` sirven para lo mismo siempre que no las reasignemos. Pero durante todo este curso seguiremos nuestro propio consejo: usaremos siempre `module.exports` para evitar problemas.