Una guía para importar y exportar módulos con JavaScript

J

Juliana Amoasei

11/08/2023

Presentación

En este artículo vamos a revisar un concepto importante en el día a día del desarrollo con JavaScript: qué es la **importación y exportación de módulos** y cómo funcionan las dos formas de hacerlo: usando **CJS** (Common JavaScript) o **ESM** (EcmaScript Modules).

Pero, ¿por qué elegimos hablar de este tema? Los conceptos de módulos y la modularidad son una parte importante del desarrollo de JavaScript, y cuando tenemos más formas de hacer algo en programación, como es el caso aquí, es aún más importante que entendamos mejor los detalles.

[TOC]

El enfoque aquí está en las importaciones y exportaciones de módulos con Node.JS, pero si trabajas o estudias front-end, también puedes encontrar información interesante.

¿Vamos?

¿Qué son los módulos de JavaScript?

El concepto de modularidad es "ocultar" los detalles de implementación de las partes que componen una aplicación, y así organizar y separar mejor el código. El propósito de modularizar el código es permitir que aplicaciones mayores sean montadas de forma "modular"; es decir, a través de varias partes independientes.

La primera forma en que se adoptó JavaScript para permitir la modularidad se denominó **CJS** (Common JS, o JavaScript común), basado en la función `require()`.

CJS, aunque era conocido como el "estándar" de Node.js para módulos, nunca se definió como una herramienta oficial de modularidad del lenguaje, ya que básicamente crea exportaciones a partir del cambio del objeto global `exports` de Node.js (en lugar de utilizar métodos propios para ello).

¿Has encontrado mucho? Cálmate, veamos con más detalle lo que esto significa en el transcurso del artículo.

Además de que este método no existe, ES6 ha especificado, oficialmente, que se llama **Ecmascript Modules** (que llamamos **ESM** o, en algunas documentaciones, de *ES6 Modules*), basado en las palabras clave `import` y `export`.

Sin in embargo, existe un desfase temporal entre las especificaciones definidas para cada versión de JavaScript y la implementación de cada una de ellas, tanto en navegadores

como en Node.js (y todo su ecosistema de librerías); y por eso hasta hoy es posible encontrar algunos *bugs* y *workarounds* para usar la sintaxis ESM con Node.js.

Por lo tanto, todavía es muy común ver el uso de CJS y `require()` en Node.js. E incluso después de la implementación de ESM y la adopción de esta nueva sintaxis por parte de las bibliotecas, gran parte de la documentación aún utiliza el formulario anterior.

¿Por qué usamos módulos?

Hemos visto que JavaScript utiliza dos formas de trabajar con módulos.

Pero, al fin y al cabo, ¿qué son los módulos en programación y para qué sirven?

Vamos: Recuerda que la definición más básica de módulo es una unidad, una pieza que se puede combinar con otras pero que tiene, por sí misma, una funcionalidad. Podemos pensar en *muebles modulares*, que se pueden ensamblar entre sí según la necesidad, aunque cada parte también funciona sola (un armario, por ejemplo). ¡¿Bien?!

Ahora, traduzcamos esta idea a la programación: podemos llamar módulo a todo código que tenga una funcionalidad específica y que se implemente de forma independiente, pudiendo así modularizar, es decir, utilizar junto con otras partes del código para desarrollar un completo solicitud.

En aplicaciones desarrolladas para Node.js, es una práctica estándar dividir la funcionalidad en tantos directorios y archivos como sea necesario para mantener el código separado y organizado.

Los navegadores tienen que descargar y cargar todos los archivos necesarios para que la aplicación funcione correctamente; de ahí la utilidad de *bundlers* o empaquetadores (como el webpack que optimiza archivos para transferencia y carga rápida de navegadores). No es el caso de las aplicaciones que se ejecutan en entorno Node.js, que pueden prescindir de este proceso de empaquetado/optimización para un único archivo común a la interfaz.

Node.js trata cada archivo como un módulo separado e independiente con su propio *namespace*. Es decir, todos los códigos definidos en un archivo, ya sean variables, funciones o clases, son restringidos, es decir, **privados**, y son accedidos solo por el archivo en el que fueron creados, a menos que se **exporten** e **importen** explícitamente en otro archivo/módulo.

También se consideran módulos de biblioteca externa que importamos a nuestros proyectos según sea necesario.

Por ejemplo, construimos un módulo que valida la secuencia numérica de las tarjetas de crédito, antes de que las tarjetas sean utilizadas por el resto de la aplicación:

```
// fichero validaciones/validacionTarjeta.js

function validaTarjeta(tarjeta) {
 // lógica interna de validación
 return tarjetaEsValida
}

export default validaTarjeta;
```

Anteriormente, la función `validaTarjeta()` se exporta explícitamente a través de la línea `export default validaTarjeta;`.

Cualquier otro módulo de la aplicación que desee utilizar el validador debe importarlo:

```
// archivo index.js

import validaTarjeta from 'validaciones/validacionTarjeta.js';

function enviaDatosCliente(datosCliente) {
 // lógica interna

    const tarjetaEsValida = validaTarjeta(datosCliente.tarjeta);
 // lógica interna continúa
}
```

En el archivo/módulo `index.js` solo puede acceder a `validaTarjeta()` y usar su lógica interna desde el momento en que el módulo es **importado** de su archivo original.

Lo mismo ocurre con las bibliotecas y las dependencias externas. Por ejemplo, si queremos trabajar con la biblioteca de validación express-validator, podemos importar solo los módulos que interesan y solo en el archivo donde se utilizará:

```
// archivo validaciones/index.js
import { body } from 'express-validator';
```

La modularización en JavaScript también permite definir las funciones, clases, objetos o variables de un módulo que será exportado, manteniendo el resto de la implementación con acceso solo en el módulo donde fueron definidas:

```
// archivo validaciones/validacionTarjeta.js

function validaTarjeta(tarjetaRecebida) {
 // lógica interna de validación
 const resultado = funcionAuxiliar(tarjetaRecebida)
 // lógica interna de validación sigue
 return tarjetaEsValida
}

function funcionAuxiliar(dato) {
 // lógica interna de la función
 return resultado
}

export default validaTarjeta;
```

En el ejemplo anterior, solo se está exportando la función `validacionTarjeta()`, y la función `funcionAuxiliar()` solo está siendo utilizada internamente por el módulo `validacionTarjeta()`, no estando disponible para ser accedida por el resto de los código

Es decir, solo tenemos una “exportación estándar” en este módulo, desde la función a la que necesita acceder a otras partes de la aplicación. El resto de la lógica (ejemplificada aquí por `funcionAuxiliar`) está restringida al módulo y no se puede acceder a ella.

Usar módulos en JavaScript

Como vimos al principio, hay dos formas de trabajar con módulos en JavaScript, utilizando el formulario CJS o ESM.

¡Veamos con más detalle cómo funciona cada una de estas formas y cuáles son las diferencias entre ellas! ¡Vamos!

CJS

CJS, la sintaxis adoptada por Node.js desde antes de la implementación de ESM, utiliza el objeto global `exports` para gestionar las exportaciones de módulos y la función `require()` para gestionar las importaciones.

> *Importante: CJS que veremos ahora utiliza funciones y objetos nativos de Node.js, como el objeto global `exports`, y no funcionará de la misma manera en los navegadores.*

Sin embargo, era común usar `exports` y `require()` en aplicaciones frontend a través de *bundlers* como webpack que permitía el uso de esta función y "traducía" el código a un formato de JavaScript que los navegadores (que no tienen el objeto global `exports` y no entiende el CJS) podrían interpretar.

Con el lanzamiento de ES6 y ESM, los navegadores comenzaron a adoptar esta, que es la sintaxis "oficial" de importación y exportación con `import` y `export` (que veremos más adelante en este artículo).

El objeto global `exports` siempre se define internamente por Node.js. Por lo tanto, cuando queremos exportar varios módulos, asignamos estos módulos como propiedades del objeto `exports`:

```
// archivo 'operaciones.js'

exports.suma = function(num1, num2) {
 return num1 + num2;
}

exports.multiplica = function(num1, num2) {
 return num1 * num2;
}
```

O bien, alternativamente:

```
// archivo 'operaciones.js'

module.exports = {
  suma(num1, num2) {
    return num1 + num2;
  },
  multiplica(num1,num2) {
    return num1 * num2;
  },
};
```

En caso de querer definir un módulo que exporte solo una función o clase, en lugar de varias, la forma es muy similar:

```
// archivo 'operaciones.js'

function suma(num1, num2) {
  return num1 + num2;
}

function multiplica(num1,num2) {
  return num1 * num2;
}

function resta(num1, num2) {
  return num1 - num2;
}

module.exports = suma;
```

En la forma anterior, no es necesario crear funciones dentro del objeto que se exportará o declarar `exports` en todas las funciones.

También podemos exportar solo una parte de las funciones usando esta misma forma:

```
module.exports = { multiplica, resta };
```

El ejemplo anterior utiliza la función de desestructuración de objetos de JavaScript. Si lo necesitas, puedes consultarlo aquí, en la documentación, qué es la desestructuración y ver algunos ejemplos de uso.

Ahora que hemos exportado nuestros módulos, es hora de importarlos para que puedan ser utilizados en otras partes de la aplicación. Para ello, utilizamos la función `require()`.

Primero exportemos la función `suma()`:

```
// archivo 'operaciones.js'

function suma(num1, num2) {
 return num1 + num2;
}

module.exports = suma;
```

Ahora podemos importar `suma()` para usar en otro módulo:

```
// archivo 'index.js'

const suma = require('./operaciones.js');
```

En el ejemplo anterior, usamos `require()` para importar módulos desde nuestro propio código, pasando un *string* como parámetro con la ruta relativa al archivo donde se encuentran los módulos que queremos importar. El retorno de la función `require()` es normalmente la función, clase u objeto importado, que guardamos en la variable `suma`.

El uso del mismo nombre en la importación no es obligatorio, sino un patrón lingüístico.

Alternativamente, puede utilizar la función de desestructuración de objetos para importar solo los módulos necesarios:

```
// archivo 'operaciones.js'

module.exports = {
 suma(num1, num2) {
   return num1 + num2;
```

```
  },
  multiplica(num1, num2) {
    return num1 * num2;
  },
  resta(num1, num2) {
    return num1 - num2;
  }
};

// archivo 'index.js'

const { suma, multiplica } = require('./operaciones.js');
```

O bien, puedes importar el objeto completo y utilizar sus métodos de la siguiente manera:

```
// archivo 'index.js'

const operaciones = require('./operaciones.js');

const numerosSumados = operacionSuma(4, 2) // 6
```

El mismo principio se aplica a los módulos externos a nuestro código, ya sean nativos de Node.js o aquellos que forman partes de bibliotecas o frameworks; pero en este caso no es necesario pasar por la ruta del archivo, solo el nombre del módulo:

```
const { suma, multiplica } = require('./operaciones.js');
const fs = require('fs');
const express = require('express');
```

En el ejemplo anterior, `fs` es un módulo nativo de Node.js utilizado para que Node.js interactúe con el sistema de archivos del ordenador. Ya `express` es un framework web muy utilizado en el desarrollo de APIs con Node.js; no es un módulo nativo y debe instalarse manualmente como dependencia de un proyecto Node.js para que pueda ser importado de la forma anterior y utilizado por el código.

ESM

Desde ES6, JavaScript ha implementado soporte de modularidad real como parte del lenguaje, utilizando las palabras clave `import` y `export`.

Conceptualmente, el principio de modularidad sigue siendo el mismo que utiliza Node.js:

- Cada archivo se considera un módulo independiente;
- No se puede acceder desde el exterior a las funciones, clases y variables definidas dentro de un archivo, a menos que se exporten e importen explícitamente.

Veamos algunos ejemplos de importación y exportación de módulos con esta sintaxis. **ESM usa la palabra clave `export` antes de declarar la función, variable o clase.**

> ***Importante**: Para usar ESM en aplicaciones Node.js, debes agregar la propiedad `type: module` en el archivo `package.json`, como en este ejemplo.*

```
// exportación: archivo 'operaciones.js'

export function suma(num1, num2) {
 return num1 + num2;
}

export function multiplica(num1, num2) {
 return num1 * num2;
}
 export function resta(num1, num2) {
 return num1 - num2;
}

// importación: archivo index.js

import { suma, multiplica, resta } from './operaciones.js';
```

En el ejemplo anterior, exportamos cada una de las funciones con la palabra clave `export` y utilizamos la desestructuración de objetos para importar cada una de ellas a otro archivo.

Es posible variar la llamada de exportación. Por ejemplo, concentrar las exportaciones de módulos al final del archivo `export { suma, multiplica, resta };` y utilizar desestructuración para exportar e importar cada función por separado:

```
// exportación: archivo 'operaciones.js'

function suma(num1, num2) {
 return num1 + num2;
}

function multiplica(num1, num2) {
 return num1 * num2;
}

 function resta(num1, num2) {
 return num1 - num2;
}

export { suma, multiplica, resta };

// Importación: archivo index.js

import { suma, multiplica, resta } from './operaciones.js';
```

De esta forma, también es posible importar solo las funciones (o variables, o clases) de un módulo que se utilizará en cada parte de la aplicación; es decir, algunas funciones exportadas desde `operaciones.js` pueden ser importadas por `index.js`, otras por `app.js`:

```
// exportación: archivo index.js

import { suma } from './operaciones.js';

// Importación: archivo 'app.js' (otro archivo de aplicación)

import { multiplica, resta } from './operaciones.js';
```

También es muy común el escenario donde solo se debe exportar una función o clase, aunque hay otras funciones, clases o variables en el archivo. En este caso, usa `export default` (exportación estándar, en traducción libre) en lugar de `export`:

```
function validaTarjeta(tarjeta) {
 // lógica interna de validación
 const resultado = funcionAuxiliar(algunDato)
 // lógica interna de validación sigue
 return tarjetaEsValida
}

function  funcionAuxiliar(dato) {
 // lógica interna de la función
 return resultadoDeLaLogica
}

export default validaTarjeta;
```

Con `export default`, la desestructuración no se usa para importar el módulo, ya que el objeto exportado es el objeto completo:

```
// Importación: archivo index.js

import validaTarjeta from './validaciones.js';
```

Mientras que la exportación con `export` solo se puede usar en funciones con nombre, la exportación estándar con `export default` se puede hacer en funciones anónimas y también en objetos literales:

```
// exportación: archivo 'operacion.js'

export default function(num1, num2) {
 return num1 + num2;
}

// Importación: archivo index.js

import operacion from './operacion.js';
```

En el ejemplo anterior, podemos crear el identificador `operacion` en la importación de la función anónima; los identificadores de los `imports` se comportan como constantes.

Gran parte de las guías de estilo (*linter*) recomendarán que todos los módulos utilizados en un archivo se exporten explícitamente, con la sintaxis `import { funcion1, funcion2, funcion3 } from 'modulo'`. Sin embargo, al importar un módulo que exporta varias declaraciones, es posible escribir de forma reducida:

```
import * as operaciones from './operaciones.js';
```

En la forma anterior, todos los `export` (siempre que no sean `export default`) del módulo se importan como propiedades del objeto `operaciones`. Dado que todas las exportaciones no estándar (no `default`) son necesariamente nombradas, el identificador de la función, clase o variable se convierte en una propiedad de este objeto. Por ejemplo:

```
// archivo 'operaciones.js'

function suma(num1, num2) {
 return num1 + num2;
}

function multiplica(num1, num2) {
 return num1 * num2;
}

function resta(num1, num2) {
 return num1 - num2;
}

export { suma, multiplica, resta };

//////////////////////

// archivo 'index.js'

import * as operaciones from './operaciones.js';

operaciones.suma(1, 1) //2
```

```
operaciones.multiplica(1, 1) //1
operaciones.resta(1, 1) /0
```

También es posible, aunque no muy usual, declarar `export` y `export default` en el mismo arquivo. En este caso, la importación es:

```
// archivo 'operaciones.js'

export default function(num1, num2) {
 return num1 + num2;
}

export function multiplica(num1, num2) {
 return num1 * num2;
}

export function resta(num1, num2) {
 return num1 - num2;
}

//////////////////////

// archivo 'index.js'

import suma, { multiplica, resta } from './operaciones.js';

suma(1, 1) //2
multiplica(1, 1) //1
resta(1, 1) //0
```

La extensión `.mjs`

Hay algunos patrones que JavaScript adopta en sus extensiones de archivo para indicar los diferentes "tipos" de código.

Por ejemplo: las aplicaciones frontend que utilizan React pueden adoptar el estándar `.js` para archivos escritos en JavaScript "estándar" y `.jsx` para archivos que utilizan las

características de la extensión JSX, aunque la funcionalidad de código y archivos sigue siendo la misma.

Asimismo, es posible utilizar los estándares `.mjs` para marcar archivos JavaScript que son módulos ESM y diferenciarlos de archivos JavaScript que no utilicen módulos (manteniendo la extensión normal `.js`).

En la práctica, usar o no este estándar no trae diferencia para el desarrollo de la aplicación, pero Node.js, "por debajo de los paños", identificará las diferentes extensiones al cargar los programas e indexar los archivos. Luego, puedes usarlo, si quieres, aunque este tipo de decisión muchas veces queda a cargo de la convención utilizada por cada proyecto/empresa.

Los archivos con extensión `.cjs` siguen el mismo principio, pero para CJS. Si es necesario utilizar la sintaxis CJS en aplicaciones que ya usan ESM (y que, por tanto, tiene definido el `"type": "module"`; como propiedad en el archivo `package.json`, será necesario utilizar la extensión `.cjs`, aunque no se recomienda mezclar las dos formas de importación en el mismo proyecto.

Puntos importantes sobre el uso de ESM:

- En la sintaxis ESM es necesario incluir el nombre completo del archivo en la ruta de acceso, incluyendo la extensión `.js`. Mientras que CJS no requiere la extensión del archivo, puede utilizar `const suma = require('./operaciones');`, por ejemplo. En los ejemplos de código que utilizamos en este artículo, decidimos mantener `.js` incluso en las importaciones realizadas con `require()` para facilitar la comprensión.

- Al igual que en CJS, todas las importaciones deben declararse en la parte superior de los archivos donde se utilizarán los módulos (`*`), y no deben realizarse dentro de funciones, clases, bucles u otros bloques de código;

- Las exportaciones pueden seguir las formas indicadas en los ejemplos, tales como:

    - Agregar `export` antes de cada declaración;

    - Montando el objeto que se exportará con `export { funcion1, funcion2 }` en la última línea del archivo;

    - Declarar `export default` antes de declarar una función anónima o;

- Declarar `export default funcion` en la última línea del archivo.
  *(*) Declarar los módulos importados en la parte superior del archivo es una **convención** seguida por la comunidad, pero no es determinante para el funcionamiento del código. Los módulos importados pasan por el mismo proceso de hoisting de declaraciones de funciones y variables y se "izan" a la parte superior de los archivos.*

¿Cuál debo usar de todos modos?

Como sabemos que CJS no es, digamos, una **implementación oficial** del lenguaje para importar y exportar módulos, lo ideal es migrar nuestro código a la sintaxis ESM, esta es una funcionalidad implementada desde ES6 especialmente para lidiar con la modularidad.

Sin embargo, en la práctica, la forma CJS se ha consolidado como la "forma de Node.js" para trabajar con modularidad, pues el objeto global `exports` y la función `require()` son *built-in* (incorporadas) en Node.js, pero no en los navegadores.

CJS permitió a Node.js desarrollar un estándar de organización de archivos modular, dividiendo naturalmente una aplicación en varios archivos/módulos diferentes según la necesidad, incluso antes de que fuera posible en los navegadores - permitiendo así a los navegadores utilizar esta función a través de *bundlers*.

A partir de la versión 13, Node.js pasó a tener soporte para ESM (en el momento en que escribimos este artículo, la versión más actual es la versión 18), lo que no solucionó completamente el problema, por lo que a partir de ese momento, Node.js ahora tiene soporte obligatorio para dos formas diferentes (y no totalmente compatibles) de trabajar con módulos.

> *Además de las diferencias en palabras clave, funciones y objetos utilizados para importar y exportar módulos, CJS y ESM tienen otras diferencias estructurales: una diferencia importante es que CJS funciona de forma **sincrónica**, mientras que ESM funciona de forma **asíncrona**, lo que dificulta la compatibilidad entre las dos formas.*

Puedes consultar este tema con más detalle en este artículo sobre async/await.

En la práctica, hasta ahora gran parte de las aplicaciones, bibliotecas y frameworks Node.js todavía utilizan CJS en sus códigos. Si bien estas aplicaciones, bibliotecas, etc. se actualizan desde la versión 13 para admitir ESM, todavía se pueden encontrar algunos *bugs*

de compatibilidad entre bibliotecas, además de que la mayoría de la documentación de bibliotecas y frameworks aún utiliza la forma anterior (CJS).

Nuestra recomendación: utilice la forma ESM en proyectos con Node.js, ya que:

- El CJS está siendo reemplazado (aunque un poco lento) y;
- ESM fue implementado para ser la herramienta de modularidad de JavaScript.

Si encuentra errores en bibliotecas y dependencias (lo que todavía ocurre con cierta frecuencia), siempre es bueno buscar en la documentación de cada una de ellas en relación con el uso de módulos o en la parte de *issues* del repositorio de herramientas en GitHub (si está disponible).

Si no encuentras respuesta, siempre puedes acceder a un foro o publicar un nuevo *issue* en caso de herramientas de código abierto.

¿Y en el frontend?

Después de especificar ESM en ES6, los navegadores comenzaron a implementar las nuevas funciones en sus *engines*. Firefox, por ejemplo, implementó soporte ESM completo a partir de la versión 60 (2018) puede consultar la tabla de compatibilidad de navegadores con ESM en este enlace de MDN.

Antes de la implementación total, se utilizaban *bundlers* de código para trabajar con modularidad, pero hoy en día todos los navegadores (con excepción de Internet Explorer) ya tienen el ESM nativo. CJS nunca ha sido compatible con navegadores precisamente por utilizar elementos que son nativos solo de Node.js como el objeto global `exports`.

> *Aunque ESM actualmente se implementa de forma nativa en los navegadores, siempre es interesante considerar el uso de bundlers como webpack para generar la versión de "producción" de una aplicación frontend, ya que estas herramientas tienen funciones para aumentar el rendimiento del código que se ejecuta en el navegador.*

Hay otros detalles sobre el uso de módulos por parte de JavaScript en el frontend, como el uso de la etiqueta HTML

NAVEGACIÓN