**Import y Export en Javascript (ES Modules) – Maria Monchot - Katerina**

La siguiente sección está inspirada en el video del Youtuber Fazt : <https://www.youtube.com/watch?v=0t-Le4kdaMg&ab_channel=FaztCode>

Antes de **2015**, momento en el que nace **ECMAScript 2015** (antes llamado ES6) con multitud de cambios y novedades, Javascript carecía de sistema de módulos oficial. Aunque pueda parecer extraño, tiene su sentido: Javascript nació como un lenguaje de programación para el navegador, que servía de apoyo a las páginas HTML+CSS para dotar de interactividad y de mayor dinamismo a su funcionamiento.

Durante mucho tiempo, esta fue la única manera de importar archivos de js:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Import-Export en JavaScript</title>

</head>

<body>

    <script src="./index.js"></script>

</body>

</html>

Con el tiempo, se comienza a utilizar más y más Javascript en los sitios web. Pero donde se produce un antes y un después, es cuando se hace posible utilizar **Javascript fuera del navegador**. Su implementación más popular es [NodeJS](https://nodejs.org/es/), aunque actualmente existen otras como por ejemplo [Deno](https://deno.land/). Todo esto, junto a la fuerte evolución de Javascript, vuelve muy necesario tener un sistema para incluir código desde ficheros externos y permitir organizar mejor código Javascript que comenzaba a ser muy extenso. Esto generó las preguntas: ¿Cómo exportamos el código que hicimos en un archivo, en otro? Y ¿cómo utilizamos este código importado?

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Import-Export en JavaScript</title>

</head>

<body>

    <script src="./math.js"></script>

    <script src="./index.js"></script>

</body>

</html>

//dentro de math.js

const secret\_number = 30;

//dentro de index.js

console.log(secret\_number);

//en la consola: 30

Si bien el anterior código podría parecer una buena solución, hay maneras más eficientes.

MODULOS

¿Qué es un módulo?

Un módulo es un archivo de JavaScript que agrupa funciones, clases, variables que luego pueden ser exportadas y utilizadas en otras partes de nuestra aplicación. Un módulo permite ocultar funcionalidad del mismo y solo exportar aquello para lo que ha sido implementado.

COMMON JS

Surge cerca de 2009 como una serie de pautas para crear un sistema de módulos en el ecosistema Javascript. Algo más tarde, el equipo de **NodeJS** implementó parcialmente una versión síncrona de CommonJS, por lo que consigue popularizar un sistema de módulos no oficial como el que puedes ver a continuación:

// module-name.js

module.exports = {

/\* ... \*/

}

// index.js

const module = require("./module-name.js");

const package = require("package");

module.method();

De esta forma, haciendo un require() podemos importar módulos CommonJS que se exportan con un module.exports desde otros archivos. También es habitual encontrar importaciones de paquetes que habitualmente residen en la carpeta [node\_modules](https://lenguajejs.com/npm/administracion/carpeta-node_modules/), obteniendo la carpeta principal del campo main del [package.json](https://lenguajejs.com/npm/administracion/package-json/). Este sistema se conocería más tarde como [bare imports](https://lenguajejs.com/javascript/caracteristicas/modulos-es6/" \l "bare-imports) (importaciones desnudas), haciendo referencia a que no se indica una ruta de un archivo, sino un  con el nombre del paquete.

Sin embargo, estos require() son creados por NodeJS y no son compatibles directamente en navegadores, salvo que se preprocese o transpile antes con alguna herramienta como podría ser un empaquetador o automatizador del estilo de Webpack, Parcel, Rollup, Babel o similar. Estas herramientas buscan los require() y los sustituyen por el código del fichero correspondiente, uniendo y empatando todos los archivos Javascript necesarios de nuestra aplicación web en un sólo archivo Javascript llamado bundle.

ES MODULES (ESM)

En 2015, aterriza **ECMAScript 2015** (antiguamente ES6) y con ella multitud de novedades en Javascript. Una de ellas, el sistema de módulos nativos de Javascript. Los tienes detalladamente explicados en [Módulos ECMAScript (ESM)](https://lenguajejs.com/javascript/caracteristicas/modulos-es6/), pero básicamente, son una evolución de lo mejor de los anteriores, en versión simplificada:

// module.js

export const data = 42;

export const method = () => console.log("Hello");

// index.js

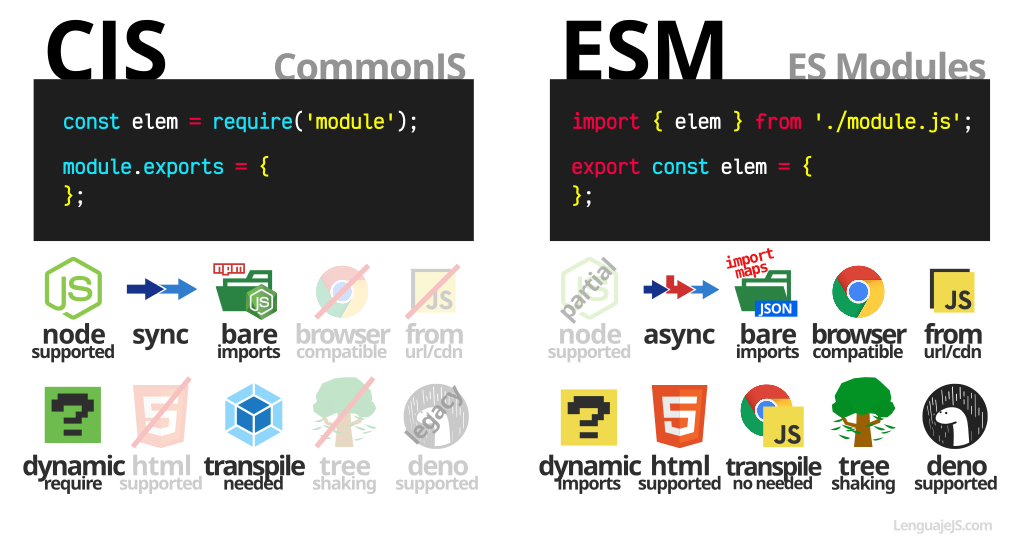
import { data, method } from "./module.js";

Este sistema de módulos nativo por fin nos permite cargar módulos externos con una sintaxis simple y de forma síncrona y asíncrona. Eso sí, con una pequeña pega que seguiremos sufriendo durante un tiempo: esperar que la industria vaya abandonando **CommonJS** a favor de **ESM**.

COMMONJS VS ESM

Hoy en día, de todo lo anterior, lo más común suele ser utilizar **CommonJS** o **ESM**. En ecosistemas donde predomina la utilización de **NodeJS**, es más frecuente encontrarse usando **CommonJS**, mientras que en sistemas más modernos, de navegador o, por ejemplo, **Deno**, es más habitual utilizar el enfoque de **ESM**.

Al margen de su sintaxis, la cuál ya hemos visto en los ejemplos anteriores, ambos tienen sus diferencias pero las más populares son las siguientes:



* [NodeJS](https://nodejs.org/en/) soporta tradicionalmente la sintaxis require de **CommonJS**, y aunque cada vez soporta mejor **ESM**, aún el soporte no es completo y tiene una amplia comunidad con paquetes utilizando **CommonJS** a través de **NPM**.
* **CommonJS** sólo permite cargar módulos de forma síncrona, mientras que **ESM** permite carga síncrona y asíncrona.
* **NodeJS** permite hacer require() de [bare imports](https://lenguajejs.com/javascript/caracteristicas/modulos-es6/" \l "bare-imports) utilizando npm mientras que **ESM** puede hacerlo mediante [import maps](https://wicg.github.io/import-maps/), un fichero .json que incluye la URL de referencias a los nombres de los paquetes «desnudos».
* Los require de **CommonJS** no son compatibles en el navegador de forma directa, mientras que los import de **ESM** si lo son si se indica el atributo <script type="module"> en los scripts que los utilicen.
* **CommonJS** no permite cargar directamente desde una URL o CDN un módulo, mientras que con **ESM** puedes hacerlo sin problemas y funciona directamente desde un navegador.
* Con **ESM** es posible hacer **tree-shaking** (eliminación de código no utilizado) de serie, mientras que en cambio con **CommonJS** no es posible, aunque se puede conseguir utilizando plugins de terceros de Webpack como [webpack-common-shake](https://github.com/indutny/webpack-common-shake).
* **CommonJS** se utiliza en sistemas que generan bundles y utilizan técnicas de preprocesado o transpilado para generar builds. Por otro lado, **ESM** puede utilizarse tanto en entornos de procesados/transpilado o directamente desde el navegador, sin necesidad de transpilar. [SkyPack.dev](https://www.skypack.dev/) es un proyecto que pretende fomentar y popularizar el uso de paquetes de npm optimizados para utilizar sin necesidad de herramientas de preprocesado.
* [Deno utiliza ESM](https://deno.land/manual#comparison-to-nodejs) por defecto, y no soporta los require de **CommonJS**. Sin embargo, pueden soportarse con un [módulo para Deno de compatibilidad con Node](https://github.com/denoland/deno_std/tree/main/node).

CÓMO EXPORTAR/IMPORTAR ARCHIVOS EN ESMODULES

//index.html

<body>

    <script type="module" src="./index.js"></script>

</body>

//math.js

export const secret\_number = 30;

//index.js

import {secret\_number} from "./math.js";

console.log(secret\_number);

Si tenemos varias funciones y queremos que cualquier archivo pueda hacer uso de ellas:

//math.js

export function add( x , y ){

    return x + y

};

export function substract( x , y ){

    return x - y

};

export function multiply( x , y ){

    return x \* y

};

export function divide( x , y ){

    return x / y

};

//index.js

import {add, substract, multiply,divide} from "./math.js";

console.log(add(1,2));

console.log(substract(1,2));

console.log(multiply(1,2));

console.log(divide(1,2));

Cabe destacar que no solo podemos importar/exportar constantes y funciones. También podemos hacerlo con variables, clases, etc.

Cuando se tienen muchas importaciones, el código puede verse un poco largo. Este es un ejemplo de cómo puedes organizarlo mejor:

import {

    add,

    substract,

    multiply,

    divide,

    Math

} from "./math.js";

En el fragmento de código anterior, importamos la clase Math. Pero, ¿Cómo la usamos?

Tenemos que instanciarla:

*//la clase Math en el archivo math.js:*

export class Math {

    double(x){

        return x \* x

    };

};

*//archivo index.js:*

*//Las clases son una plantilla, no un objeto que podamos utilizar.*

*//instanciando un objeto de tipo de clase Math, podremos acceder a sus propiedades*

*//y métodos.*

const math = new Math();

console.log(math.double(4));

//en consola: 16

También podríamos exportar “por defecto”. ¿Qué significa esto?

Significa que si se llama al módulo por un nombre general, se va a importar algo por defecto. ¿Qué sería ese algo? Bueno, lo tenemos que definir nosotros.

*//index.js*

*//para importar por default lo que está en un módulo:*

import myModuleMath from "./math.js";

console.log(myModuleMath);

*//De esta manera hacemos uso de las funciones que se encuentran dentro del módulo:*

console.log(myModuleMath.substract(100,50));

*//math.js*

*//para exportar por default lo que está en un módulo:*

*//por ej, por default se exportará todo menos la clase Math*

export default {

    add,

    substract,

    multiply,

    divide

}

Se puede importar por defecto o manualmente, por lo que no está mal utilizar export en cada función.

En más, a veces, quizá sólo planeemos usar una de las funciones, variables, clases, etc. Que se exportan por defecto, y llamándolas manualmente evitamos ese export por default de cosas que no utilizaremos.

También se puede exportar por defecto y manualmente al mismo tiempo:

import myModuleMath, {add, substract} from "./math.js";

Ahora, si quisiéramos importar todo de un módulo, podríamos usar:

import \* as math from "./math.js";

//para luego usar la clase que se encuentra en ese módulo podemos hacer:

const math2 = new math.Math();

console.log(math2.double(4));

Hasta ahora, todo lo que hemos visto estaba destinado al Frontend mediante ESModules.

Sin embargo, cuando nosotros comencemos a codear para el Backend, notaremos que el ‘import’ nos dará un error. Esto es debido a que Node se maneja con CommonJS.

Para Seguir codeando (pero en Backend) tendremos que hacer lo siguiente:

En nuestro proyecto de servidor, por ejemplo en este caso, en la carpeta ‘server’ hay que crear un archivo json. Esto lo realizamos abriendo la terminal y posicionándonos en la carpeta server.

Una manera rápida de hacerlo es : npm init –y

Es importante que luego entremos en dicho archivo y añadamos el “type”:”module” así:

//package.json

{

  "name": "server",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "type":"module",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "keywords": [],

  "author": "",

  "license": "ISC"

}

Habiendo realizado esto, ya podremos usar el famoso ‘import’.

Como ejemplo, dentro de la carpeta server, haz un archivo math.js y un index.js.

//server/math.js

export const add = (x,y) => x + y;

//server/index.js

import {add} from './math.js';

console.log(add(10,3));

Bibliografía:

* <https://www.tutorialesprogramacionya.com/javascriptya/temarios/descripcion.php?inicio=100&cod=102#:~:text=Un%20m%C3%B3dulo%20es%20un%20archivo,lo%20que%20ha%20sido%20implementado>.
* <https://lenguajejs.com/automatizadores/introduccion/commonjs-vs-es-modules/>