

INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

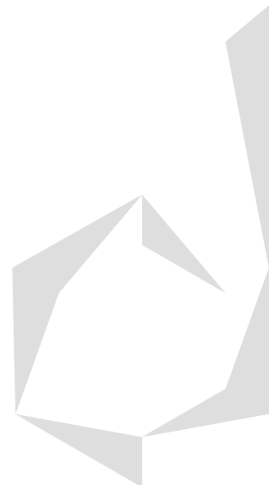
PROGRAMACIÓN WEB - BACKEND

JAVASCRIPT - NODE.JS

Introducción Node.js

Contenido

Introducción a Node.js	2
Características:	2
Instalación y Uso:	4
Nodemon y PM2	4
Callbacks y Promesas	6
Callbacks: Manejo del flujo de ejecución	6
Promesas: Manejo de errores y el flujo de ejecución	7
API Node.js	9
Módulo Global: Métodos y atributos por default en la API de Node.js	9
Módulo File System (FS): Permite acceder a archivos del sistema	10
Referencias:	10

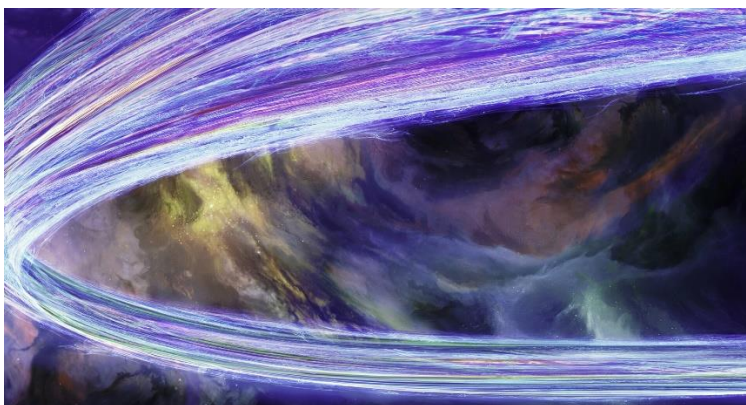


Introducción a Node.js

Node.js es un entorno de ejecución de JavaScript creado en 2009 que corre fuera del navegador y está orientado a servidores, por lo cual se utiliza como código de Backend, en vez del código JavaScript normal, el cual corre en el navegador y se suele utilizar como Frontend.

Características:

- **JavaScript fuera de navegadores:** Es importante que Node.js corra fuera del navegador (Chrome, Firefox, Edge, etc.) porque de esa manera es como se logra montar código JavaScript en servidores, microcontroladores para aplicaciones IoT o en cualquier otro dispositivo donde se pueda correr software.
- **Concurrencia:** Es un concepto en informática que se refiere a la capacidad de un sistema para ejecutar múltiples tareas simultáneamente. En un entorno concurrente, varias tareas pueden progresar de forma aparentemente simultánea, lo que puede mejorar la eficiencia y el rendimiento del sistema al aprovechar mejor los recursos disponibles, como la CPU y los dispositivos de entrada/salida.
 - **Monohilo:** Aunque Node.js es concurrente, solo cuenta con un único hilo de ejecución, donde las tareas se ejecutan secuencialmente una tras otra, en un solo flujo de ejecución. Pero en ese mismo hilo todas sus entradas y salidas son asíncronas, de esta forma se puede correr un único proceso o hilo en el procesador, que programa varias tareas a la vez.

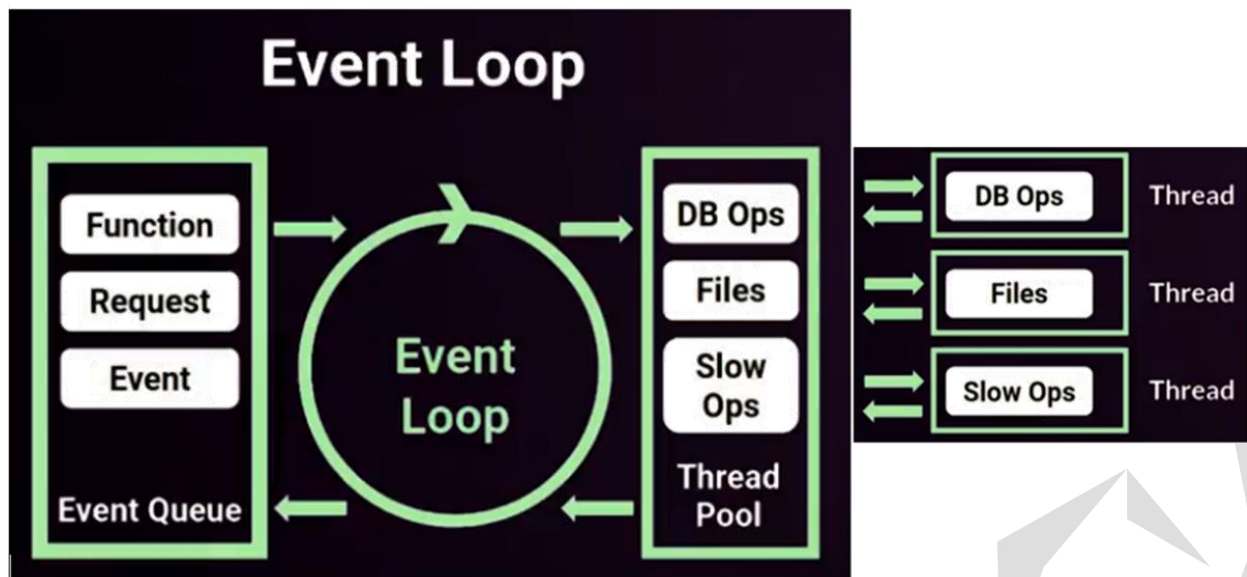


- El monohilo asíncrono lo que permitirá es tener muchas conexiones de entradas/salidas de archivos, servidores FTP, instrumentación virtual, etc.
- **Motor V8:** El motor V8 open source desarrollado en C++ por Google se diseñó para optimizar el rendimiento de JavaScript en el navegador web Google Chrome y para la ejecución de su framework Node.js. Su funcionamiento consta de dos etapas:
 - Primero, interpreta el programa JavaScript para generar un código bytecode intermedio.
 - Y luego, emplea su motor JIT (compilación Just-In-Time) para compilar selectivamente partes del código bytecode a código máquina, lo que resulta en una ejecución más rápida y eficiente del programa.
- **Modular:** Node.js basa su funcionamiento en archivos de código muy pequeños llamados módulos, que pueden venir por defecto en el paquete de node, se pueden importar por medio de gestores de paquetes o podemos crear e incorporar nuestros propios módulos.

- **Orientado a Eventos:** En este enfoque, el flujo de control del programa se determina principalmente por la ocurrencia de eventos, como acciones del usuario (clics, presionar el teclado, movimiento del mouse, etc.), cambios de estado en la aplicación o interacciones con sistemas externos. Esto nos permite programar de forma reactiva a través de algo llamado Event Loop o bucle de eventos.
 - **Event Loop:** Es un simple bucle que todo el tiempo está ejecutándose y se encarga de gestionar la ejecución de las operaciones del programa de forma asíncrona, como las entradas/salidas, las operaciones de red, las llamadas a funciones, etc.
 - **Event Queue:** Los eventos del programa llegan al **Event Loop** a través del **Event Queue**, el cual lee todas las líneas del código Node.js (funciones, peticiones, etc.) y genera un evento por cada una de ellas. Estos eventos se irán formando en una cola e irán pasando uno a uno al **Event Loop**.
 - **Thread Pool:** Si la tarea que es mandada de: **Node.js** → **Event Queue** → **Event Loop**, puede ser resuelta de forma rápida, esta se procesará, pero si no lo es, se mandará al **Thread Pool**, donde se gestionará de forma asíncrona a través de hilos (threads) individuales, ejecutándola a su propio ritmo, pero sin que le estorbe a las nuevas tareas que lleguen o a las que se puedan resolver de forma más rápida, cuando se termine de ejecutar la tarea de algún thread, se lanzará un evento que la devolverá al **Event Loop** o al **Event Queue** (si es que algún otro evento depende de ella).

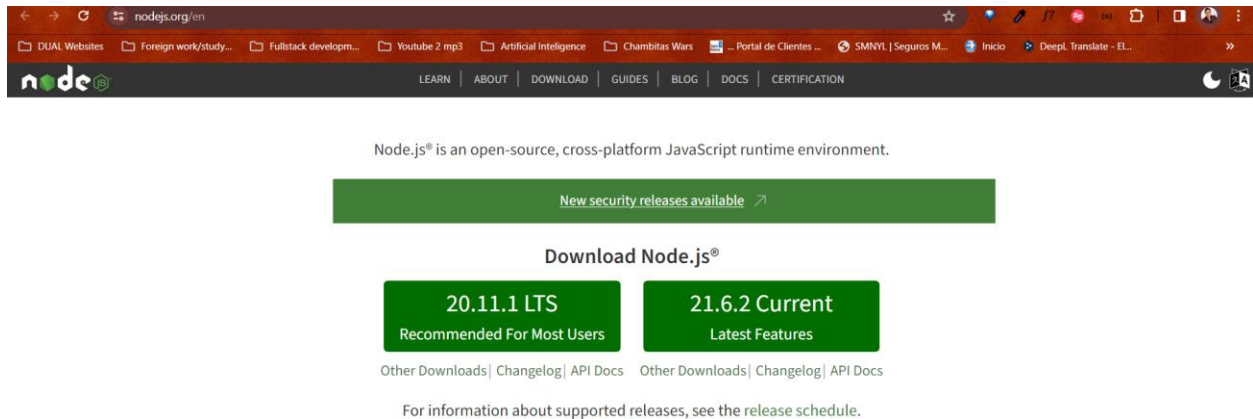
Aquí es donde realmente se visualiza la funcionalidad asíncrona de Node.js y las tareas que se podrían llegar a mandar al **Thread Pool** son operaciones con bases de datos, escritura/lectura de archivos, etc.

 - Aquí es donde el utilizar Node.js presenta grandes ventajas en comparación con otros lenguajes de servidor como PHP, Python, Java, etc. donde las tareas siempre se ejecutan de forma síncrona, una tras la otra.
 - Este proceso donde se involucra el Event Loop, Event Queue y Thread Pool es ejecutado por medio del motor V8 mencionado previamente.



Instalación y Uso:

Para poder utilizar Node.js, primero debemos instalar el paquete de Node.js: <https://nodejs.org/en>



Y posteriormente ejecutar los comandos para comprobar que en efecto se han instalado los paquetes:

node -v

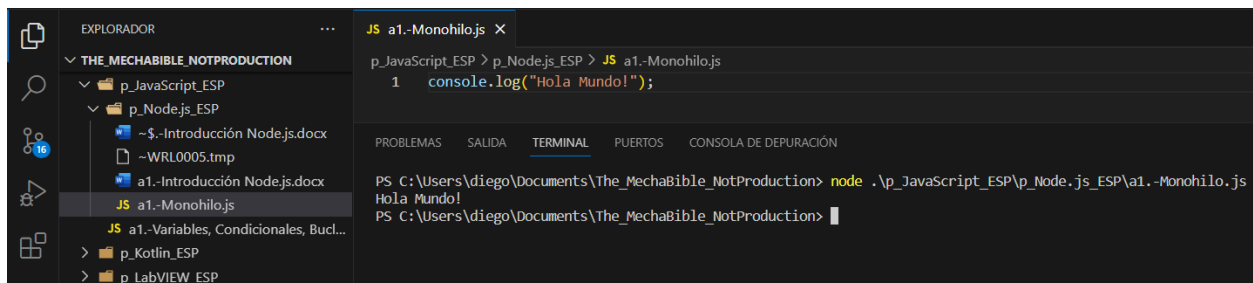
npm -v

Finalmente, para correr un programa de JavaScript con Node.js debemos ejecutar el siguiente comando en consola:

node \directorio\archivoJavaScript.js

Al ejecutarse el archivo JavaScript se inicializa un proceso de node, descrito anteriormente del **motor V8** y cuando se termine de ejecutar el programa, se cerrará el proceso de node inicializado:

Inicio de proceso node → Código JavaScript.js → Código bytecodes → Motor JIT (Compilación Just-In-Time) → Partes del código bytecode a código máquina → Fin de proceso node.



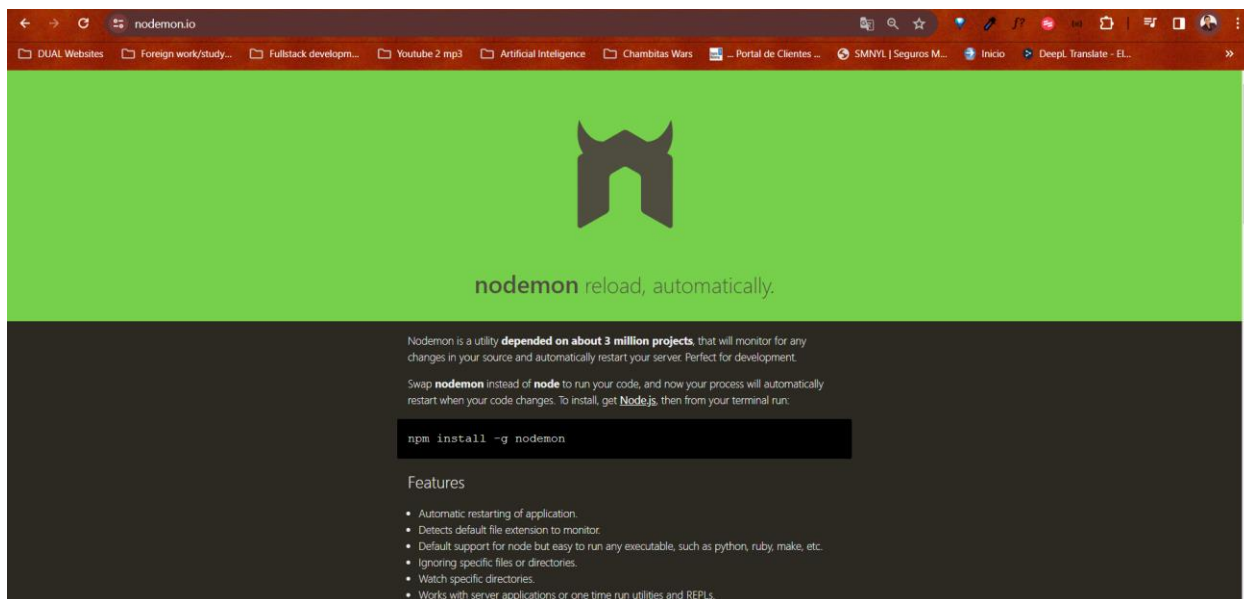
Nodemon y PM2

Es tedioso tener que estar ejecutando este comando cada vez que queramos correr nuestros programas de Node.js:

node \directorio\archivoJavaScript.js

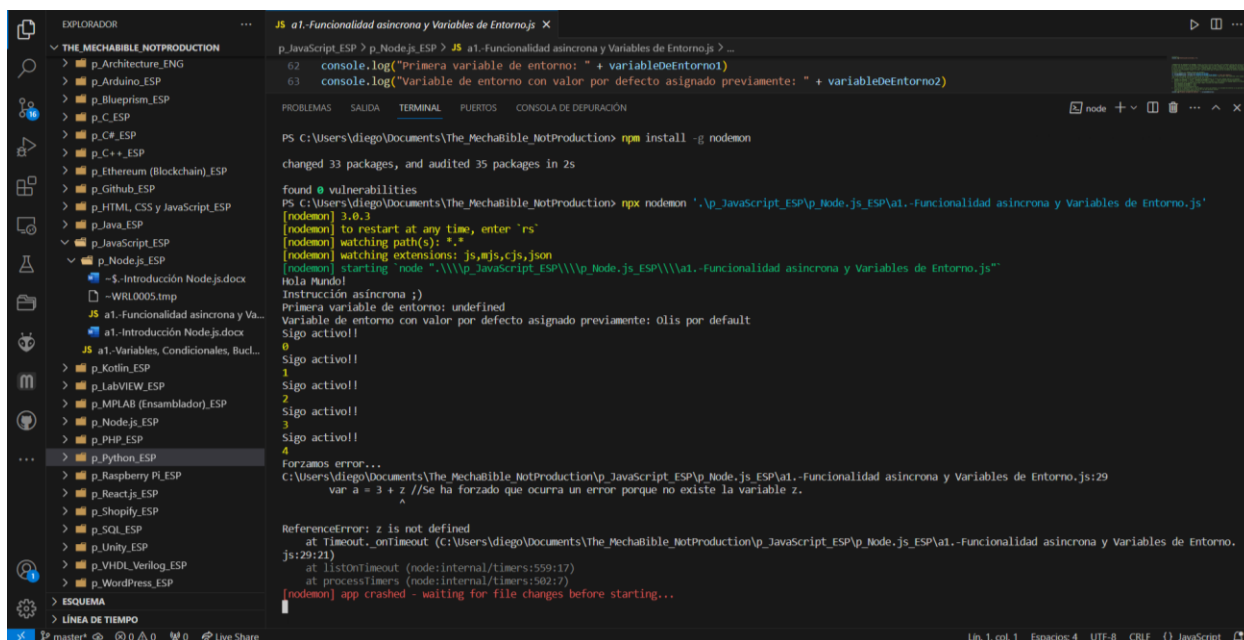
Por lo que existen herramientas como Nodemon y PM2, las cuales al detectar que ha existido un cambio en el código principal o en alguna de sus dependencias (módulos de código de los cuales depende el código principal), ejecutará de forma automática el código Node.js, para su instalación simplemente en consola debemos ejecutar el siguiente comando, si obtenemos un error, se debe ejecutar como administrador:

npm install -g nodemon

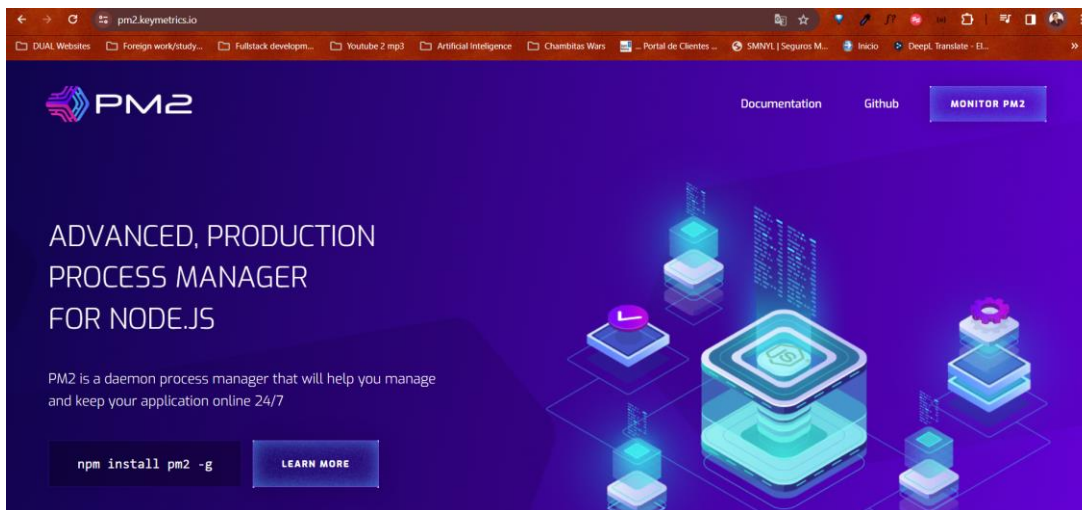


Para que este funcione en Windows, en vez de ejecutar el archivo con la palabra reservada node, se ejecutará con npx nodemon y en el que momento que el código cambie y lo guarde, este se ejecutará por sí solo:

npx nodemon \directorio\archivoJavaScript.js



Una alternativa similar para producción es PM2, pero esta no se debe utilizar en desarrollo, ya que hacerlo así se complica mucho.



Callbacks y Promesas

Callbacks: Manejo del flujo de ejecución

Como dentro de Node.js todas las funciones se ejecutan al mismo tiempo de forma asíncrona, para controlar su flujo de ejecución (orden en el que corren) existen los **callbacks**, los cuales son simplemente parámetros de una **función asíncrona** que **a su vez son una función**. Estos se deben ejecutar dentro de **la misma función** para que **cuando esta se utilice en otro lugar del programa** se ejecute la acción descrita en esa función.

Además, se hace uso de dos posibles métodos que crean un temporizador o ejecutan de forma repetida una función cada cierto tiempo:

- **setTimeout()**: Función de Node.js que ejecuta una vez la función que recibe como primer parámetro, después de haber transcurrido el tiempo indicado en milisegundos de su segundo parámetro, funcionando así como un **temporizador**.
- **setInterval()**: Función de Node.js que se utiliza para ejecutar varias veces la función que recibe como primer parámetro, con un intervalo de espera entre cada ejecución indicado en milisegundos por su segundo parámetro.

A continuación, se presenta un ejemplo con pseudocódigo, donde primero se presenta la declaración del **callback** dentro de la **función asíncrona** y luego su utilización:

```
function asíncrona(parámetro_1, callback, parámetro_n){  
  /*Acción o acciones a ejecutar de forma desordenada con los parámetros de la función asíncrona  
  y sin temporizador.*/  
  setTimeout(function(){
```



```

/*Acción o Acciones que utilizan los parámetros de la función asíncrona y se ejecutan de
forma desordenada con temporizador.*/

/*El callback se puede declarar en cualquier lado de la función asíncrona, pero esa es la
acción que estará ejecutándose de forma ordenada, justo en la línea de código donde se
declare. Además, esta puede recibir parámetros propios que se deberán declarar cuando se
utilice la función asíncrona.*/

callback(parámetroCallback_1, parámetroCallback_n);

}, tiempo de espera en milisegundos del temporizador);

}

asincrona(parámetro_1, function(parámetroCallback_1, parámetroCallback_n){

    /*Acción o Acciones que utilizan los parámetros del callback y se ejecutan justo en la parte
    donde se haya declarado el callback dentro de la función asíncrona.*/

    });

}, parámetro_n)

```

Promesas: Manejo de errores y el flujo de ejecución

Cuando pueda ocurrir un problema al ejecutar una **función asíncrona** es cuando más conviene utilizar **promesas**, las cuales utilizan la instrucción **return**, son instancias de la clase **Promise** y sirven más que nada para el manejo de excepciones a través de dos parámetros llamados **resolve** y **reject**, los cuales se declaran dentro de un **callback**.

- **resolve**: Parámetro del **callback** de la **promesa** que se utiliza para devolver una respuesta cuando la tarea se ha completado sin errores (excepciones).
- **reject**: Parámetro del **callback** de la **promesa** que se utiliza para devolver una respuesta cuando la tarea no se ha podido completar debido a una excepción.

```

function asincronaConPosiblesErrores(parámetro_1, parámetro_exito, parámetro_n){

    return new Promise(function(resolve, reject){

        /*Acción o acciones a ejecutar de forma desordenada con los parámetros de la función
        asíncrona y sin temporizador.*/

        setTimeout(function(){

            /*Acción o Acciones que utilizan los parámetros de la función asíncrona y se ejecutan de
            forma desordenada con temporizador.*/

            /*En las promesas es de buenas prácticas evaluar el estado de algún parámetro de la

```


función asíncrona que indique si la operación que se busca realizar fue **exitosa** o si ocurrió alguna **excepción** a través de un condicional if/else y dentro de él utilizar los parámetros **resolve** o **reject** respectivamente.

Cabe mencionar que la acción o acciones indicadas por los parámetros **resolve** o **reject** serán los que se ejecutarán de forma ordenada, justo en la línea de código donde se declaren. Además, pueden recibir **parámetros propios** que se deberán declarar cuando se utilice la **función asíncrona que pueda ocasionar o no una excepción**.*/

```
if(parámetro_exito == true){
    /*Los parámetros de las instrucciones resolve y reject representan el valor que se
    pasa al manejador de éxito .then() o al de error .catch(), respectivamente. */
    resolve(parámetroExito_1, parámetroExito_n);
} else{
    reject(parámetroExcepcion_1, parámetroExcepcion _n);
}
}, tiempo de espera en milisegundos del temporizador);
});
}

asincronaConPosiblesErrores(parámetro_1, parámetro_exito, parámetro_n)

/*Acción o Acciones que utilizan los parámetros de las instrucciones resolve o reject dependiendo
de si la operación que se quería realizar con el callback dentro de la función asíncrona ocasionó
una excepción o no, esto se realiza a través de las instrucciones .then y .catch respectivamente.*/
.then(function(parámetroExito_1, parámetroExito_n){
    /*Acción o Acciones que utilizan los parámetros de éxito de la instrucción resolve dentro del
callback de la función asíncrona.*/
})

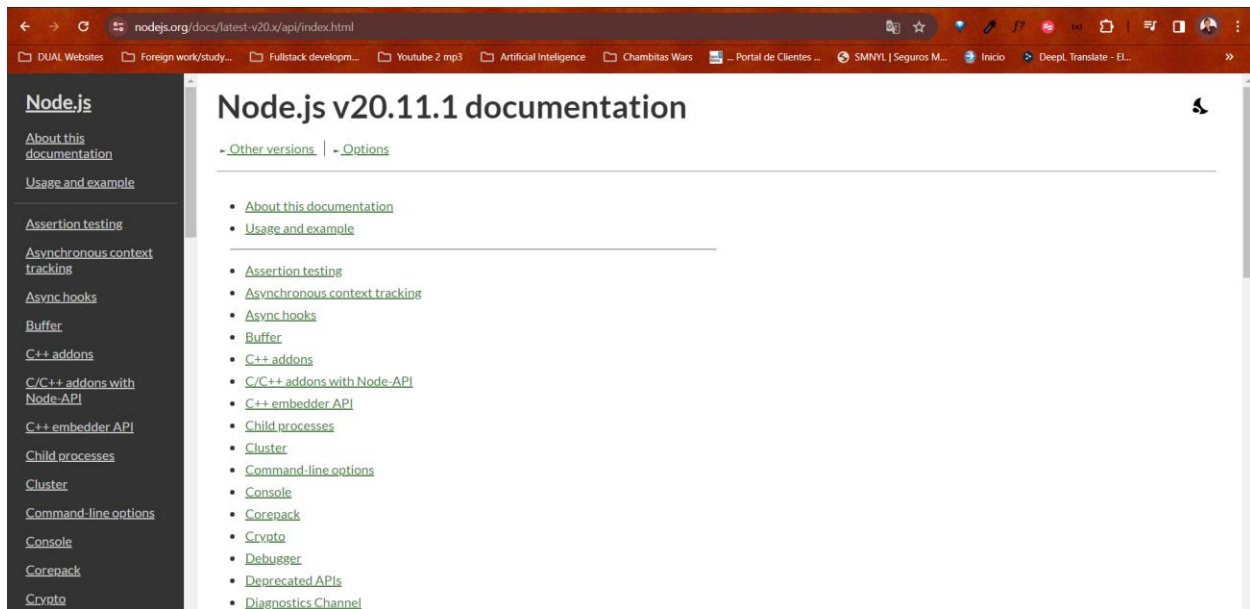
.catch(function(parámetroExcepcion_1, parámetroExcepcion _n){
    /*Acción o Acciones que utilizan los parámetros de excepción de la instrucción reject
dentro del callback de la función asíncrona.*/
});
```



API Node.js

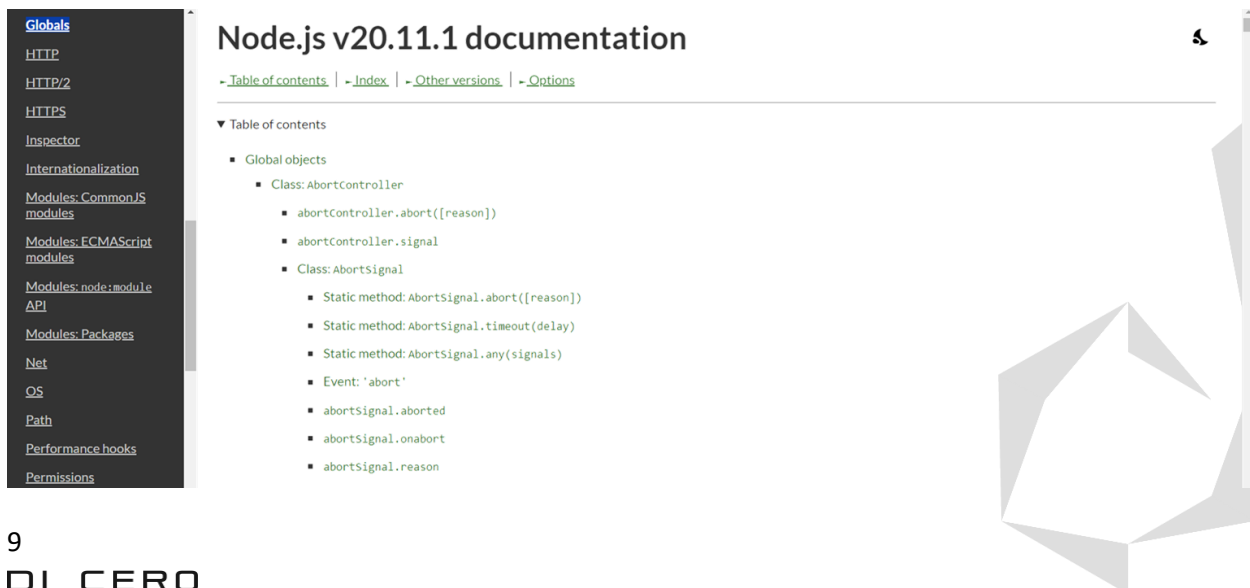
La API con la documentación oficial de Node.js se encuentra en el siguiente link:

<https://nodejs.org/docs/latest-v20.x/api/index.html>



Módulo Global: Métodos y atributos por default en la API de Node.js

Hay algunas cosas que no necesitan de ninguna importación para ejecutarse, como lo son los métodos `setTimeout()`, `setInterval()`, `console.log()`, `clearInterval()`, `clearTimeout()`, `queueMicrotask()`, `clearImmediate()` y `setImmediate()`, algunos de ellos fueron previamente utilizados y mencionados en las **promesas** y **callbacks**, a estos se les llama **métodos o módulos globales** y se accede a todos ellos a través de la palabra reservada **global**, *si queremos ver los métodos/atributos relacionados al paquete podemos imprimir en consola esa palabra reservada*, o simplemente podemos buscar en parte donde dice Globals dentro de la API de Node.js.



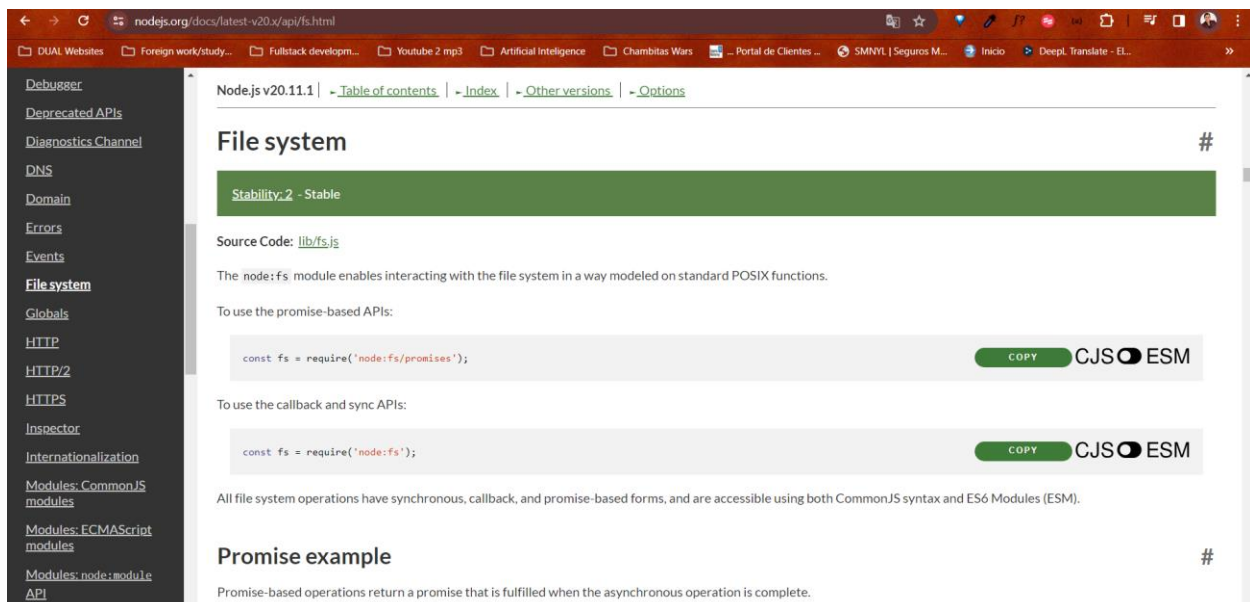
Pero como en todo lenguaje de programación o framework, hay ciertos módulos o paquetes con sus respectivos métodos o atributos que necesitan ser importados a través de su API para poder ser utilizados, veremos algunos de ellos a continuación.

Módulo File System (FS): Permite acceder a archivos del sistema

Los métodos y atributos del módulo File System o FS sirve para leer, escribir o modificar archivos del sistema.

Hay varias formas de importar un módulo dentro de nuestro programa de Node.js, la más popular es la que utiliza la instrucción **require()**; y el resultado que devuelva esta instrucción se deberá guardar en una variable, para que a través de ella se pueda utilizar la nomenclatura del punto para utilizar los métodos y atributos del módulo.

La forma en la que específicamente se importa cada módulo se puede observar en la API oficial:

The image is a screenshot of the Node.js documentation website for the File System module. The browser's address bar shows 'nodejs.org/docs/latest-v20.x/api/fs.html'. The left sidebar contains a navigation menu with links like 'Debugger', 'Deprecated APIs', 'Diagnostics Channel', 'DNS', 'Domain', 'Errors', 'Events', 'File system' (which is highlighted), 'Globals', 'HTTP', 'HTTP/2', 'HTTPS', 'Inspector', 'Internationalization', 'Modules: CommonJS modules', 'Modules: ECMAScript modules', and 'Modules: node:module API'. The main content area is titled 'File system' and includes a green bar indicating 'Stability: 2 - Stable'. It provides the source code 'lib/fs.js' and explains that the 'node:fs' module enables interacting with the file system in a way modeled on standard POSIX functions. Two code blocks are shown: one for promise-based APIs ('const fs = require('node:fs/promises');') and one for callback and sync APIs ('const fs = require('node:fs');'). Both blocks have 'COPY' buttons and radio buttons for 'CJS' and 'ESM'. A note states that all file system operations have synchronous, callback, and promise-based forms, and are accessible using both CommonJS syntax and ES6 Modules (ESM). A 'Promise example' section is also visible at the bottom.

En el caso del módulo File system la instrucción que se debe utilizar es:

```
const fs = require('node:fs');
```

Referencias:

Platzi, Carlos Hernández, “Curso de Fundamentos de Node.js”, 2023 [Online], Available: <https://platzi.com/new-home/clases/1759-fundamentos-node/25184-node-origenes-y-filosofia/>