Curso Node:

Lo primero que hacemos es instalar node versión manager (Administrador de versiones de Node).

https://github.com/coreybutler/nvm-windows/releases

¿Qué hace NVM?

Te permite instalar múltiples versiones de Node.js en tu máquina.

Cambiar fácilmente entre ellas con un solo comando:

nvm use 18.20.4

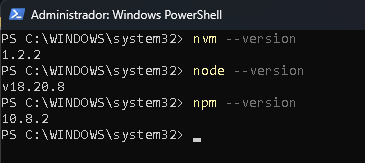
nvm use 22.18.0

Mantiene cada versión aislada, sin conflictos.

También instala el npm correspondiente a cada versión de Node automáticamente.

**En resumen:** NVM es como un “switch” de versiones de Node, ideal para desarrolladores que trabajan en varios proyectos.

Una vez instalada, para verificar que sea asi nos vamos a consola y escribimos:



Esto nos asegura de que esta correctamente instalada, si marca errores de que no esta instalada ve a:

C:\Program Files

Y revisa que existan las carpetas: nvm y nodejs

Si no existe instala el setup de la página oficial y configura las rutas:  
C:\Program Files\nvm

C:\Program Files\nodejs

Después configura esas mismas rutas en las Variables de Entorno en el Sistema.

Despues ejecuta powershell como administrador y descarga las versiones de node que quieras con:  
nvm install <numeroVersion>

Puedes ver las versiones instaladas con:  
nvm ls

Y puedes usar la versión que quieras con:

nvm use <numeroVersion>

Preguntas comunes:

¿Qué es NodeJS?

Es un “runtime enviroment” o “ambiente de ejecución” de js en nuestra computadora.

Básicamente es un ambiente de ejecución para ejecutar js desligado del navegador.

Non—Blocking I/O:

Node puede ejecutar varios procesos en paralelo gracias a una librería llamada “Libuv”

NPM: Node Package Manager.

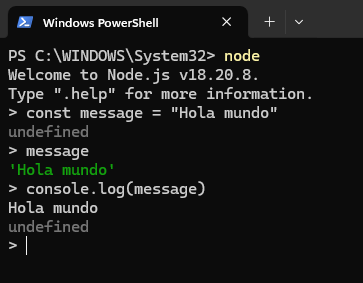
NPX: Node Package Execute (Permite ejecutar ciertos paquetes sin necesidad literalmente de instalarlos globalmente en nuestro equipo).

I/O: Entradas y salidas, se refiere principalmente a la interacción con el disco duro del sistema o a procedimientos en los cuales la librería de “libuv” va a entrar para que podamos tener esta asincronía o procedimientos que se ejecutan en paralelo.

Blocking: Básicamente es cuando la ejecución del código debe esperar a que se complete el proceso, pero este a su vez, impide que se sigan ejecutando otras instrucciones en paralelo.

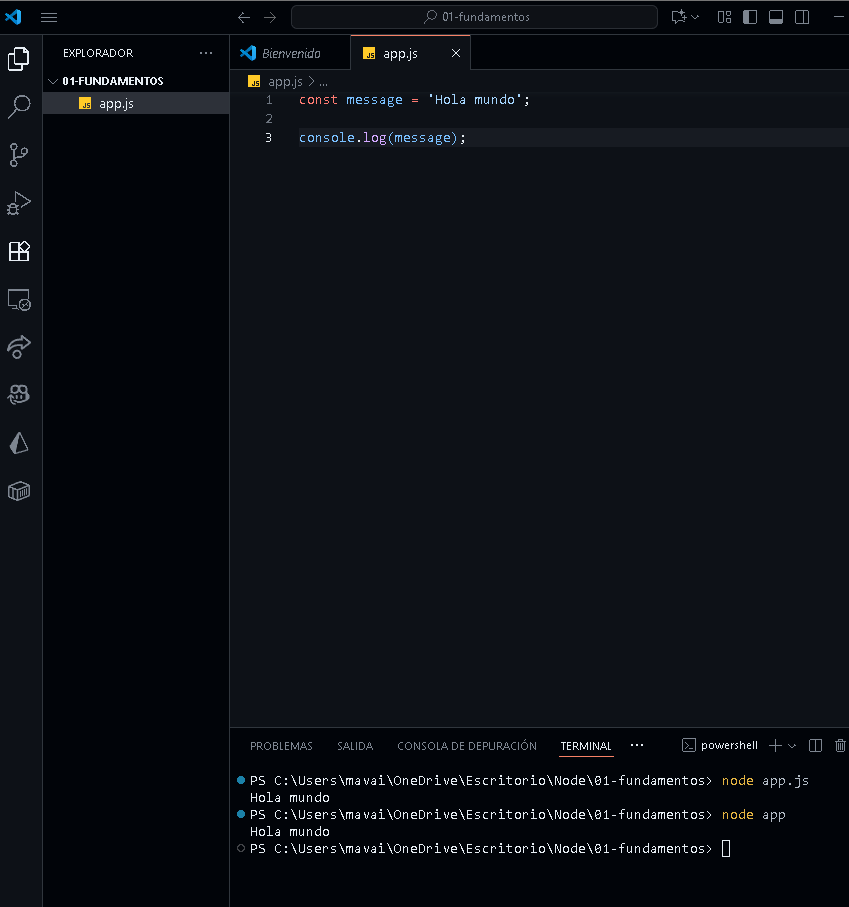
Un ejemplo de un código bloqueante es el “asyncAwait”.

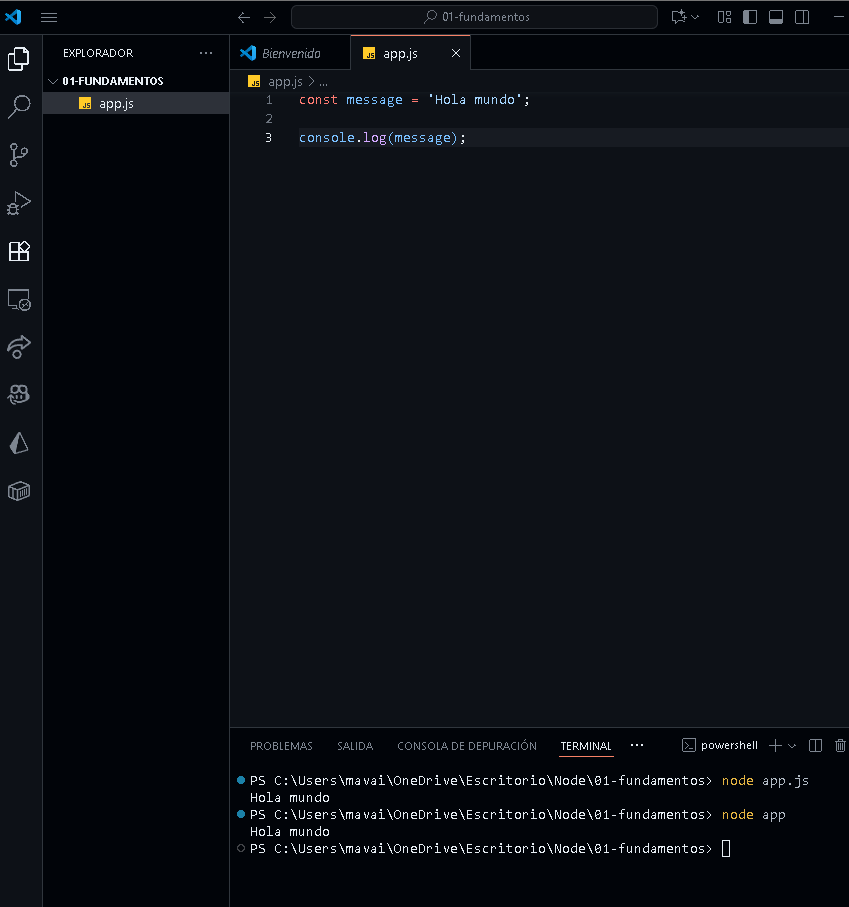
Al momento de escribir solamente “node” en nuestra consola, pwshell o en cualquier terminal que reconozca el comando “node --version", no abre la terminal interactiva de Node:



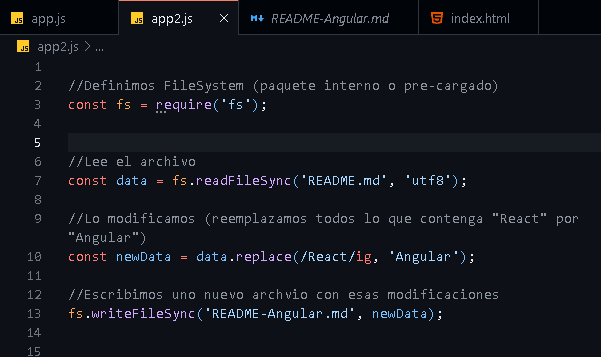
Para ejecutar un archivo en js sin la necesidad de conectarlo a un servidor o pagina web podemos escribir en la terminal:

node <nombreArchivo>

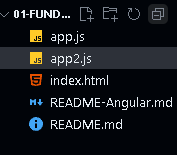




FileSystem:



Al ejecutarse con “node app2” se crea un archivo que toma el README y reemplaza lo que contiene dentro que contenga la palabra “React” por “Angular” y se crea un nuevo archivo con esas modificaciones.



Ejercicio – Encontrar la palabra react:

//Definimos FileSystem (paquete interno o pre-cargado)

const fs = require('fs');

//Lee el archivo

const content = fs.readFileSync('README.md', 'utf8');

const words = content.split(' '); //Regresa un arreglo

//Metodo1

const wordCountReact = words.filter(word => word.toLowerCase() === 'react').length;

//Metodo2

const wordCountReact2 = words.filter(word => word.toLowerCase().includes('react')).length;

//Con expresion regular

//match - forma de evaluar expresiones regulares

// Esto naturalmente regresa un arreglo, pero si no encuentra ninguno nos aseguramos que regrese un array vacio

const reactWordCount = content.match(/react/gi ?? [] ).length;

console.log(wordCountReact);

console.log(wordCountReact2);

console.log(reactWordCount);

Orden de Ejecución:



Dado que al no tener un timeout se ejecuta primero los 2 console log y después los timeouts.

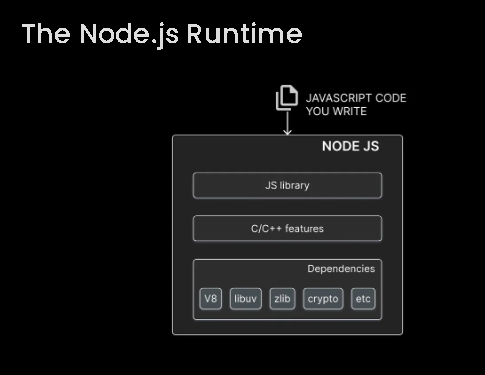
Event loop y Code Execution:

Casi todo del js es blocking o bloqueante.

Y también es "single-threaded" dado a que js solo sigue un hilo.

3 componentes principals de Node:

* Dependencias externas
* Caracteristicas de C++
* Librerias de JS que se conectan con C++ desde nuestro código



Nuestro archivo pasa primero por librerías de js, después por características de C/C++ y al final por dependencias de node.

LIBUV es la dependencia que le va a permitir trabajar a Node en tareas asíncronas, callbacks y todo lo que nosotros estaríamos acostumbrados a esperar si tuviéramos una aplicación.

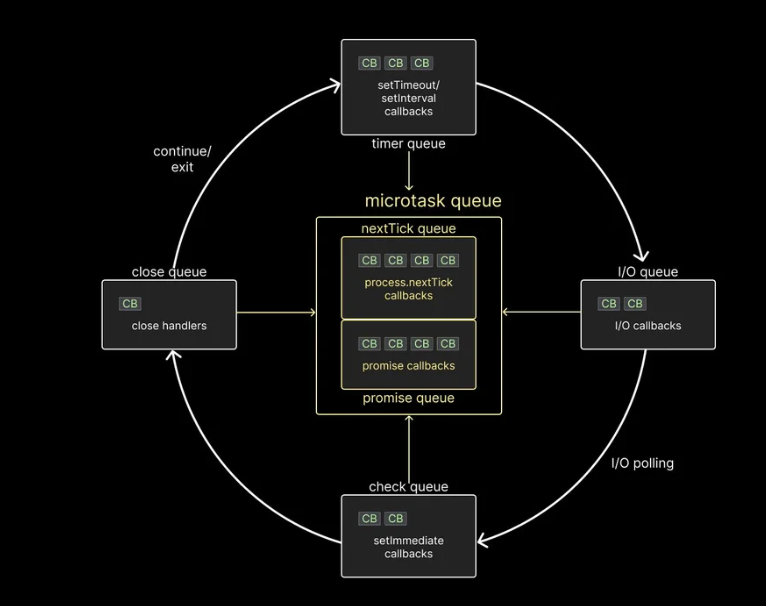
Event Loop:

El evento loop es el que decide el orden de ejecución:

Call stack (pila de ejecución)

¿Qué pasa si dos Timeouts terminan al mismo tiempo?

¿Qué pasa si una promesa termina al mismo tiempo que otro callback?



<https://www.builder.io/blog/visual-guide-to-nodejs-event-loop>

microtask queue:

Hace referencia a una cola de espera, una cola de microtareas, en las cuales ya están listas para que sean ejecutadas por el thread principal, podríamos verlo de esa manera.

Todo lo que esta dentro del circulo ya no es controlado por el Libuv, esto ya es el proceso principal.

El evento loop sigue las siguientes reglas:

* Callbacks en el microtask se ejecutan primero. Es decir, vamos a considerar que todo el trabajo síncrono es ejecutado de primera instancia. Nuevamente, los callbacks en el microtask se ejecutan primero.
* Luego, todos los callbacks dentro del “timer queue” se ejecutan (Es decir, después de los callbacks y código síncrono, que vendrían siendo los “setTimeouts” se vendrían ejecutando inmediatamente después)
* Callbacks en el microtask queue (si hay) se ejecutan después de los callbacks timers, primero tareas en el nextTIck queue y luego tareas en el promise queue (el promise queue siempre va a venir después de los callbacks, es decir, si 2 tareas terminan al mismo tiempo, una es una promesa y la otra es un procedimiento de algún callback, es decir una función que tenemos que ejecutar, primero va a ejecutar el callback y después el promise queue, tienen prioridad en este caso los callbacks) ).
* Callbacks de I/O se ejecutan
* Callbacks en el microtask queue se ejecutan (si hay), y luego promise queue (si hay)
* Todos los callbacks en el check queue se ejecutan.
* Callbacks en el microtask se ejecutan después de cada callback en el check queue. (Siguiendo el mismo orden anterior, nextTick y luego promise).
* Por último, todos los callbacks en el close queue son ejecutados.
* Por ultima vez en el mismo ciclo, los microtask queue son ejecutados de la misma forma, nextTick y luego promise queue.

Cuando se completa una tarea asíncrona en el libuv, ¿En qué momento decide Node ejecutar la función callback asociada en la pila de llamadas (callstack)?

R: Callbacks son ejecutados solo cuando el callstack esta vacio.

¿Node espera a que la pila de llamadas este vacia antes de ejecutar un callback o interrumple el flujo normal de ejecución para ejecutar el callback?:

R: El flujo normal de ejecución no se interrumpirá para ejecutar una función de devolución de llamada.

¿Qué pasa con otros métodos asíncronos como setTimeout y setInterval, que también retrasan la ejecución de un callback?

R: Los callbacks de setTimeouts y setIntervals tienen prioridad, pero no interrumpen el flujo normal.

Si dos tareas asíncronas como setTimeout y readFile se completan al mismo tiempo, ¿Cómo decide Node que callback ejecutar primero en la pila de llamadas? ¿Uno tiene prioridad sobre otro?:

R: Los callbacks de temporizador se ejecutan antes que las devoluciones de llamada de I/O, incluso si ambas están listas exactamente al mismo tiempo.

Ejemplo de tareas sincornas y asíncronas:

Tareas síncronas:

Las tareas síncronas se ejecutan una tras otra, en orden, bloqueando la ejecución hasta que terminan.



Tareas asíncronas:

Las tareas asíncronas no bloquean el flujo principal.

Se programan para ejecutarse más tarde, permitiendo que el resto del código siga corriendo.

Hay varios tipos: callbacks, Promises, async/await, setTimeout, fetch, etc.

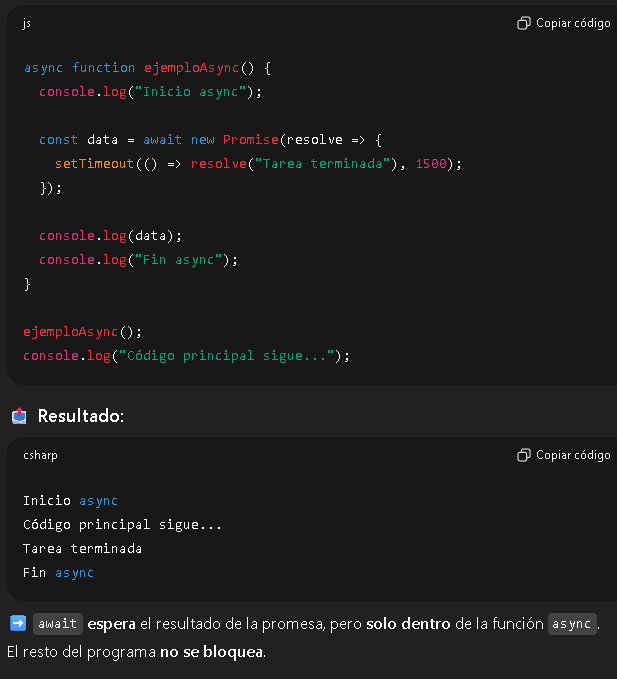
Ejemplo 1(setTimeout):



Ejemplo 2: (promesas):



Ejemplo 3 (async/await):



En resumen:

| **Tipo de tarea** | **Ejemplo** | **¿Bloquea ejecución?** | **Cuándo se ejecuta** |
| --- | --- | --- | --- |
| **Síncrona** | console.log(), cálculos, bucles for | ✅ Sí | Inmediatamente |
| **Asíncrona** | setTimeout, fetch, Promise, async/await | ❌ No | Más tarde (Event Loop) |