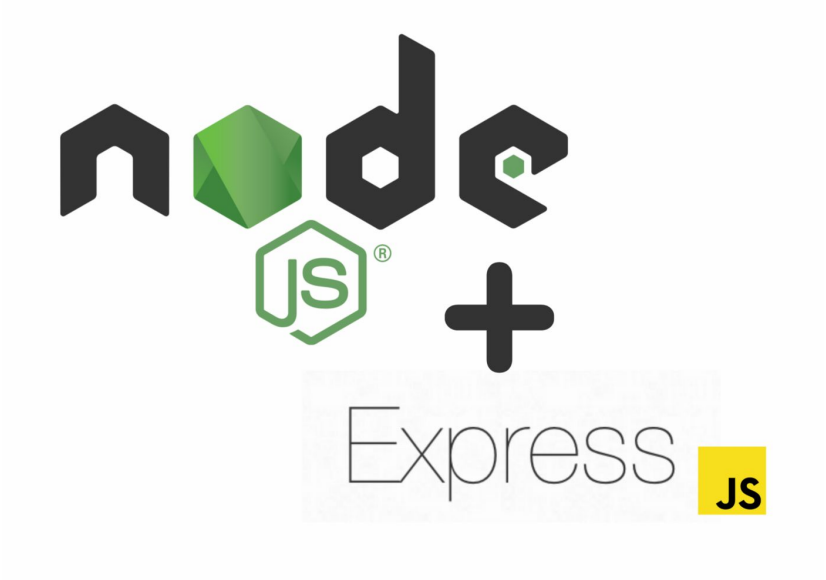
releevant.



linkedin.com/in/maortizolid

maortizolid@gmail.com

releevant.

Módulos nativos en NodeJS

Los módulos nativos de Node.js son conjuntos de funcionalidades incorporadas en el núcleo del entorno Node.js.

Estos módulos proporcionan una variedad de capacidades esenciales para el desarrollo de aplicaciones, abarcando desde operaciones de entrada/salida hasta manipulación de archivos, manejo de eventos, networking y más.

A diferencia de los módulos de terceros, los módulos nativos no requieren instalación adicional y están disponibles de inmediato al usar Node.js.

releevant.

Algunos ejemplos de módulos nativos de Node.js:

- **fs (File System)**: Proporciona funciones para interactuar con el sistema de archivos, como leer y escribir archivos, crear directorios, etc.
- **http y https:** Permiten la creación de servidores web y la realización de solicitudes HTTP o HTTPS.
- **events:** Facilita la implementación y manipulación de eventos, esencial para la programación basada en eventos en Node.js.
- **path:** Ayuda en la manipulación y construcción de rutas de archivos y directorios de manera segura y consistente entre plataformas.
- **os:** Proporciona información sobre el sistema operativo, como la arquitectura de la CPU, la memoria libre, etc.
- **crypto:** Ofrece funciones criptográficas, como la creación de hashes y la generación de claves criptográficas

Estos módulos nativos forman parte integral del ecosistema Node.js y son esenciales para el desarrollo de aplicaciones eficientes y robustas. Al comprender y utilizar estos módulos, podemos aprovechar al máximo la potencia y versatilidad de Node.js para construir aplicaciones escalables y de alto rendimiento.

releevant.

Vamos a ver algunos de ellos para entender mejor su funcionamiento.

os

Proporciona información sobre el sistema operativo, como la versión, arquitectura de la CPU, la memoria libre, número de procesadores, etc.

1.os-info.js

```
const os = require('node:os')

console.log('Sistema operativo:', os.platform())
console.log('Version del sistema operativo:', os.release())
console.log('Arquitectura del sistema operativo:', os.arch())
console.log('Memoria total:', os.totalmem() / 1024 / 1024)
console.log('Memoria libre:', os.freemem() / 1024 / 1024)
console.log('Número de procesadores:', os.cpus())
console.log('uptime:', os.uptime() / 60 / 60)
```

releevant.

fs

El módulo `fs` (File System) en Node.js proporciona una interfaz para interactuar con el sistema de archivos del sistema operativo. Este módulo ofrece métodos tanto síncronos como asíncronos para realizar operaciones relacionadas con archivos, como leer, escribir, modificar y eliminar archivos, así como manipular directorios. Vamos a ver algunos de sus métodos.

`fs.statSync`

Es un método del módulo `fs` (File System) en Node.js que se utiliza para obtener información sobre un archivo o directorio de manera síncrona y bloqueante (detendrá la ejecución del programa hasta que la información del archivo esté disponible). Este método devuelve un objeto stats que contiene detalles como el tamaño del archivo, el tiempo de creación, el tiempo de modificación, etc.

2.fs-stat.js

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

const stats = fs.statSync('./archivo.txt')

console.log(
  stats.isFile(), // si es un fichero
  stats.isDirectory(), // si es un directorio
  stats.isSymbolicLink(), // si es un enlace simbolico
  stats.size // tamaño en bytes
)
```

releevant.

`fs.readFileSync`

Se utiliza para leer el contenido de un archivo de manera síncrona. A diferencia de fs.readFile, que opera de forma asíncrona y utiliza devoluciones de llamada, `fs.readFileSync` bloquea la ejecución del programa hasta que la operación de lectura del archivo se completa.

`3.1.fs-readFileSync.js`

```
// fs.readFileSync -> Síncrono
const fs = require('node:fs')

console.log('leyendo el primer archivo...')
const text = fs.readFileSync('./archivo.txt', 'utf-8')
console.log('primer archivo:', text)

console.log('.....haciendo cosas mientras leemos el archivo...')

console.log('leyendo el segundo archivo...')
const text2 = fs.readFileSync('./archivo2.txt', 'utf-8')
console.log('segundo archivo:', text2)
```

releevant.

fs.readFile

Se utiliza para leer el contenido de un archivo de manera asíncrona. A diferencia de la versión síncrona `fs.readFileSync`, `fs.readFile` no bloquea la ejecución del programa y utiliza una devolución de llamada para manejar el resultado de la operación de lectura.

3.2.fs-readFile.js

```
// fs.readFile -> Asíncrono
const fs = require('node:fs')

console.log('leyendo el primer archivo...')
fs.readFile('./archivo.txt', 'utf-8', (err, data) => {
  if (err) throw err
  console.log('primer archivo:', data)
})

console.log('.....haciendo cosas mientras leemos el archivo...')

console.log('leyendo el segundo archivo...')
fs.readFile('./archivo2.txt', 'utf-8', (err, data) => {
  if (err) throw err
  console.log('segundo archivo:', data)
})
```

releevant.

Vamos a ver el uso de **promesas** y operaciones asíncronas con **async/await** usando el método `readFile` del módulo `fs`

Promesas

4.1.fs-promise.js

```
const fs = require('node:fs/promises')

console.log('leyendo el primer archivo...')
fs.readFile('./archivo.txt', 'utf-8')
  .then(data => {
    console.log('primer archivo:', data)
  })

console.log('.....haciendo cosas mientras leemos el archivo...')

console.log('leyendo el segundo archivo...')
fs.readFile('./archivo2.txt', 'utf-8')
  .then(data => {
    console.log('segundo archivo:', data)
  })
```

releevant.

async/await

En este ejemplo utilizamos una función anónima asíncrona que nos permite utilizar await. Los paréntesis finales indican que la función debe ejecutarse inmediatamente después de ser definida.

4.2.fs-async-await.js

```
const { readFile } = require('node:fs/promises');

(async () => {
  console.log('leyendo el primer archivo...')
  const text = await readFile('./archivo.txt', 'utf-8')
  console.log('primer archivo:', text)

  console.log('.....haciendo cosas mientras leemos el archivo...')

  console.log('leyendo el segundo archivo...')
  const text2 = await readFile('./archivo2.txt', 'utf-8')
  console.log('segundo archivo:', text2)
})()
```

releevant.

Vamos a ver el ejemplo anterior utilizando ES Modules (ESM)

4.3.fs-async-await.mjs

```
import { readFile } from 'node:fs/promises'

console.log('leyendo el primer archivo...')
const text = await readFile('./archivo.txt', 'utf-8')
console.log('primer archivo:', text)

console.log('.....haciendo cosas mientras leemos el archivo...')

console.log('leyendo el segundo archivo...')
const text2 = await readFile('./archivo2.txt', 'utf-8')
console.log('segundo archivo:', text2)
```

releevant.

Promesas en paralelo (`Promise.all`)

Se utiliza para manejar múltiples promesas simultáneamente. Recibe un array de promesas y devuelve una nueva promesa que se resuelve cuando todas las promesas en el array se han resuelto, o se rechaza si alguna de ellas se rechaza.

4.4.fs-async-await-parallel.mjs

```
import { readFile } from 'node:fs/promises'

Promise.all([
  readFile('./archivo.txt', 'utf-8'),
  readFile('./archivo2.txt', 'utf-8')
]).then(([text, text2]) => {
  console.log('primer archivo:', text)
  console.log('segundo archivo:', text2)
})
```

En resumen, `Promise.all` se utiliza aquí para manejar la lectura simultánea de los dos archivos de manera eficiente y esperar a que ambas operaciones se completen antes de continuar con la ejecución del código.

releevant.

fs.readdir

Se utiliza para leer el contenido de un directorio de manera asíncrona. Proporciona una lista de los nombres de archivos y subdirectorios presentes en el directorio especificado

4.5.ls.js

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

fs.readdir('.', (err, files) => {
  if (err) throw err
  files.forEach(file => {
    console.log(file)
  })
})
```

releevant.

path

El módulo path en Node.js proporciona utilidades para trabajar con rutas de archivos y directorios de una manera segura y consistente entre plataformas. Algunos de los métodos más utilizados son:

- **path.sep():** Se utiliza para obtener el carácter que se utiliza como separador entre directorios en las rutas de archivos. Por ejemplo, en sistemas operativos basados en Unix (como Linux o macOS), path.sep sería '/' (barra diagonal), mientras que en sistemas operativos basados en Windows, sería '\' (barra invertida).
- **path.join()**: Combina segmentos de ruta para formar una ruta completa, teniendo en cuenta las diferencias en las barras diagonales entre sistemas operativos.
- **path.resolve()**: Resuelve una ruta absoluta basada en la ruta actual y las rutas proporcionadas.
- **path.basename()**: Extrae el nombre de archivo de una ruta.
- **path.dirname()**: Extrae el directorio de una ruta.
- **path.extname()**: Extrae la extensión de archivo de una ruta.

releevant.

5.path.js

```
const path = require('node:path')

// barra separadora de directorios según el sistema operativo
console.log(path.sep)

// unir las rutas con path.join()
const filePath = path.join('content', 'subfolder', 'archivo.txt')
console.log(filePath)

const base = path.basename('tmp/migue/secret/password.txt', '.txt')
console.log(base)

const extension = path.extname('image.jpg')
console.log(extension)
```

releevant.

process

Proporciona información y control sobre el proceso actual de Node.js en ejecución. Aquí hay algunas características clave de process:

- **process.argv:** Un array que contiene los argumentos de la línea de comandos con los que se inició Node.js. El primer elemento es la ruta del ejecutable de Node.js y el segundo elemento es la ruta del script en ejecución.
- **process.env:** Un objeto que contiene las variables de entorno del sistema.
- **process.cwd():** Devuelve el directorio de trabajo actual del proceso.
- **process.exit():** Termina el proceso actual.
- **process.on():** Permite registrar manejadores de eventos para varios eventos del proceso, como 'exit', 'uncaughtException', entre otros.

Estos son solo algunos ejemplos, pero process ofrece una variedad de funcionalidades que permiten interactuar y controlar el entorno y comportamiento del proceso de Node.js en ejecución. Es una parte fundamental para gestionar la ejecución de scripts y aplicaciones en Node.js.

releevant.

6.process.js

```
// argumentos de entrada
console.log(process.argv);

// controlar el proceso y su salida
process.exit(1);

// controlar eventos del proceso
process.on('exit', () => {
   // p.e. limpiar los recursos
})

// current working directory
console.log(process.cwd())
```

releevant.

Vamos a utilizar algunos de estos módulos nativos y métodos que hemos visto para hacer un ejercicio donde listamos un directorio y sus archivos con el siguiente formato (si es un archivo o directorio, tamaño, fecha y hora de modificación):

```
f 2.fs-stat.js                272 22/9/2023, 4:32:24
f 3.1.fs-readFileSync.js      420 6/12/2023, 19:54:08
f 3.2.fs-readFile.js          470 6/12/2023, 19:59:06
f 4.1.fs-promise.js           417 6/12/2023, 20:01:47
```

releevant.

7.ls-advance.js

```
const fs = require('node:fs/promises')
const path = require('node:path')

// Extraemos el segundo argumento de process.argv (el directorio). Utilizamos el operador nullish coalescing para evitar un error de undefined,que si se cumple se asigna el punto como directorio.
const folder = process.argv[2] ?? '.'

async function ls (directory) {
  let files
  try {
    files = await fs.readdir(folder)
  } catch (err) {
    console.error(`error al leer el directorio ${folder}`, err)
    process.exit(1)
  }
```

releevant.

```
  const filesPromises = files.map(async file => {
    const filePath = path.join(folder, file)
    let stats
    try {
      stats = await fs.stat(filePath) // información del archivo
    } catch (err) {
      console.error(`error al leer el archivo ${filePath}`)
      process.exit(1)
    }

    const isDirectory = stats.isDirectory()
    const fileType = isDirectory ? 'd' : 'f'
    const fileSize = stats.size.toString()
    const fileModified = stats.mtime.toLocaleString()

    return `${fileType} ${file.padEnd(20)} ${fileSize.padStart(10)} ${fileModified}`
  })

  const filesInfo = await Promise.all(filesPromises)

  filesInfo.forEach(file => console.log(file))
}

ls(folder)
// node 7.ls-advance.js ../01_modulos_nodejs
```

Ejecutaremos el archivo pasándole un segundo argumento, que será el directorio que queremos leer, o bien sin pasárselo (por defecto leerá el directorio actual).

```
node .\7.ls-advance.js
node .\7.ls-advance.js ../01_modulos_nodejs (por ejemplo otro directorio como segundo argumento)
```

Como véis, el manejo de los módulos nativos en Node.js es esencial, ya que brindan herramientas integradas para tareas fundamentales, como operaciones de archivos y creación de servidores web.

Estos módulos, como fs y http, eliminan la necesidad de depender excesivamente de bibliotecas externas, mejorando la estabilidad de las aplicaciones al utilizar características clave desde la instalación de Node.js.

Esto simplifica el desarrollo, optimiza el rendimiento y contribuye a un código más eficiente y confiable.