

NODE

Uso de Node.js

Definición o concepto: Node.js es un entorno de ejecución para JavaScript que se basa en el motor V8 de Google Chrome. Permite ejecutar código JavaScript en el lado del servidor o en entornos fuera del navegador. Nació con el propósito de construir aplicaciones de red escalables y rápidas, aprovechando un modelo asíncrono y orientado a eventos.

Uso con ESM:

- Debes contar con "type": "module" en tu package.json o usar archivos con extensión .mjs.
- En lugar de require('modulo'), utilizas import modulo from 'modulo'; O import { algo } from 'modulo';.

Explicación de uso:

- **Servidor web:** Node.js se utiliza comúnmente para crear servidores HTTP, APIs REST y aplicaciones backend altamente escalables.
- **Herramientas de desarrollo:** Muchas herramientas, empaquetadores (webpack, esbuild), y generadores de sitios (Gatsby, Next.js) están construidas sobre Node.js.
- **Automatización de tareas:** Scripts para integración continua, despliegues, compilación de assets, etc.
- **I/O asíncrono:** Node.js utiliza operaciones de entrada/salida no bloqueantes, lo que permite manejar un gran número de conexiones simultáneas sin saturar un solo hilo de ejecución.

Ejemplo sencillo:

```
// server.js
// Ejemplo de un servidor HTTP básico con Node
const http = require('http');

const server = http.createServer((req, res) => {
  res.end('Hola, mundo desde Node.js!');
});

server.listen(3000, () => {
  console.log('Servidor escuchando en http://localhost:3000');
});
```

Mismo ejemplo pero con Ecma Scripts Modules:

```
// server.js
// server.mjs (o server.js con "type": "module" en package.json)
import http from 'node:http';

const server = http.createServer((req, res) => {
  res.end('Hola, mundo desde Node.js (con ESM)!');
});

server.listen(3000, () => {
  console.log('Servidor escuchando en http://localhost:3000');
});
```

Aplicaciones de consola

Definición o concepto: Una de las ventajas de Node.js es que se puede usar para crear herramientas y aplicaciones que corren directamente en la línea de comandos (CLI). Estas aplicaciones pueden leer argumentos, interactuar con el sistema, generar salidas formateadas, entre otras cosas.

Uso:

- **Scripts utilitarios:** Por ejemplo, un script que convierta archivos, minifique imágenes o compile plantillas.
- **Interacción con APIs:** Puedes crear clientes de línea de comando para consumir servicios externos.
- **Automatización de flujos de trabajo:** Herramientas internas de empresas que corren en CI/CD o actualizan datos.

Recursos:

- Node provee el objeto `process` que permite acceder a argumentos (`process.argv`), variables de entorno (`process.env`), y eventos del proceso.
- Puedes usar librerías como `yargs` o `commander` para crear CLI con parámetros y opciones bien definidas.

Ejemplo:

```
// server.js
console.log('Argumentos:', process.argv);
```

Para ejecutar

```
User@DESKTOP-K0C1M9L MINGW64 ~/Documents/ine
$ node server.js --nombre Fabricio --edad 80
Argumentos: [
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\User\\Documents\\ine\\server.js',
  '--nombre',
  'Fabricio',
  '--edad',
  '80'
]

User@DESKTOP-K0C1M9L MINGW64 ~/Documents/ine
```

Leer y grabar archivos en File System

Definición o concepto: Node.js provee el módulo `fs` que permite interactuar con el sistema de archivos. Puedes leer, escribir, crear, borrar y renombrar archivos y directorios.

Modos de operación (fs):

- **Asíncrono con callbacks:** `fs.readFile`, `fs.writeFile`.
- **Asíncrono con promesas:** `fs.promises.readFile`, `fs.promises.writeFile`.
- **Síncrono (no recomendado en producción):** `fs.readFileSync`, `fs.writeFileSync`.

Con Node.js ESM, simplemente importas el módulo.

Uso:

- `import fs from 'node:fs/promises';` para usar las promesas.
- `import { readFile, writeFile } from 'node:fs/promises';` si quieres deestructurar.

Ejemplo de lectura asíncrona:

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

fs.readFile('archivo.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error leyendo el archivo:', err);
    return;
  }
  console.log('Contenido del archivo:', data);
});
```

```
User@DESKTOP-K0C1M9L MINGW64 ~/Documents/ine
$ node server.js
Contenido del archivo: Lorem ipsum dolor sit amet consectetur adipisicing elit. Optio, ipsum necessitatibus unde ex
ercitationem consequuntur, architecto laborum ipsam quo perspiciatis molestiae possimus quia reiciendis corporis re
pudiandae tempore corrupti nisi illo dolorum.
```

Ejemplo de escritura con promesas:

```
const fs = require('fs').promises;

Explain
async function escribirArchivo() {
  try {
    await fs.writeFile('nuevo.txt', 'Contenido de prueba', 'utf8');
    console.log('Archivo escrito con éxito');
  } catch (err) {
    console.error('Error escribiendo el archivo:', err);
  }
}

escribirArchivo();
```

```
User@DESKTOP-K0C1M9L MINGW64 ~/Documents/ine
$ node server.js
Archivo escrito con éxito
```

Ejemplo de lectura asíncrona con promesas

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

try {
  const data = await readFile('archivo.txt', 'utf8');
  console.log('Contenido del archivo:', data);
} catch (err) {
  console.error('Error leyendo el archivo:', err);
}
```

Ejemplo de escritura:

```
import { writeFile } from 'node:fs/promises';

try {
  await writeFile('nuevo.txt', 'Contenido de prueba', 'utf8');
  console.log('Archivo escrito con éxito');
} catch (err) {
  console.error('Error escribiendo el archivo:', err);
}
```


Code Execution y Event Loop de Node

Definición o concepto: Node.js utiliza un modelo de ejecución basado en el **Event Loop**, que es un bucle que se ejecuta continuamente mientras haya callbacks pendientes, operaciones asíncronas o eventos por procesar. A diferencia de modelos multihilo tradicionales, Node.js corre en un solo hilo, aprovechando operaciones de I/O asíncronas para no bloquear ese hilo.

Funcionamiento:

- El código JavaScript se ejecuta en un solo hilo.
- Cuando se solicita una operación asíncrona (como leer un archivo o realizar una petición HTTP), Node delega la operación al sistema operativo o a hilos internos (controlados por la librería libuv).
- Una vez completada la operación, se registra un callback en la cola de eventos y el Event Loop lo ejecuta cuando puede.
- Este modelo permite manejar miles de conexiones simultáneas sin consumir grandes cantidades de recursos.

Fases del Event Loop:

1. **Timers:** Ejecuta callbacks programados con `setTimeout` y `setInterval`.
2. **I/O callbacks:** Ejecuta callbacks pendientes de I/O completadas.
3. **Idle, prepare:** Usado internamente.
4. **Poll:** Espera y obtiene nuevos eventos I/O.
5. **Check:** Ejecuta callbacks de `setImmediate`.
6. **Close callbacks:** Ejecuta callbacks de cierre, como `socket.on('close', ...)`.

```
console.log('Inicio');

setTimeout(() => {
  console.log('Timeout 1s');
}, 1000);

console.log('Fin');
```

```
User@DESKTOP-K0C1M9L MINGW64 ~/Documents/ine
$ node server.js
Inicio
Fin
Timeout 1s
```

Instalación de paquetes de NPM

Definición o concepto: NPM (Node Package Manager) es el gestor de paquetes por defecto de Node.js. Permite instalar librerías y frameworks desarrollados por la comunidad. Con NPM puedes crear, compartir y administrar dependencias en tu proyecto.

Comandos básicos:

- npm init: Crear un archivo package.json para un proyecto.
- npm install <paquete>: Instalar un paquete desde el registro de NPM.
- npm install --save-dev <paquete>: Instalar una dependencia de desarrollo.
- npm install: Instalar todas las dependencias listadas en package.json.
- npm uninstall <paquete>: Desinstalar un paquete.
- npm update: Actualizar paquetes a sus últimas versiones compatibles.

Ejemplo:

```
# Inicializar proyecto:
npm init -y

# Instalar express:
npm install express

# Correr el proyecto:
node app.js
```

```
1  import { randomInt } from "crypto";
2  import express from "express";
3  const app = express();
4  const puerto = randomInt(3000,3010)
5  app.get("/", (peticion, respuesta) => {
6      respuesta.send("Hola, estoy usando express")
7  })
8  app.listen(puerto, () => {
9      console.log("Servidor desde el puerto " + puerto);
10 })
11
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS SQL CONSOLE

```
User@DESKTOP-K0C1M9L MINGW64 ~/Documents/ine
$ node server.js
Servidor desde el puerto 3002
```

Variables de entorno

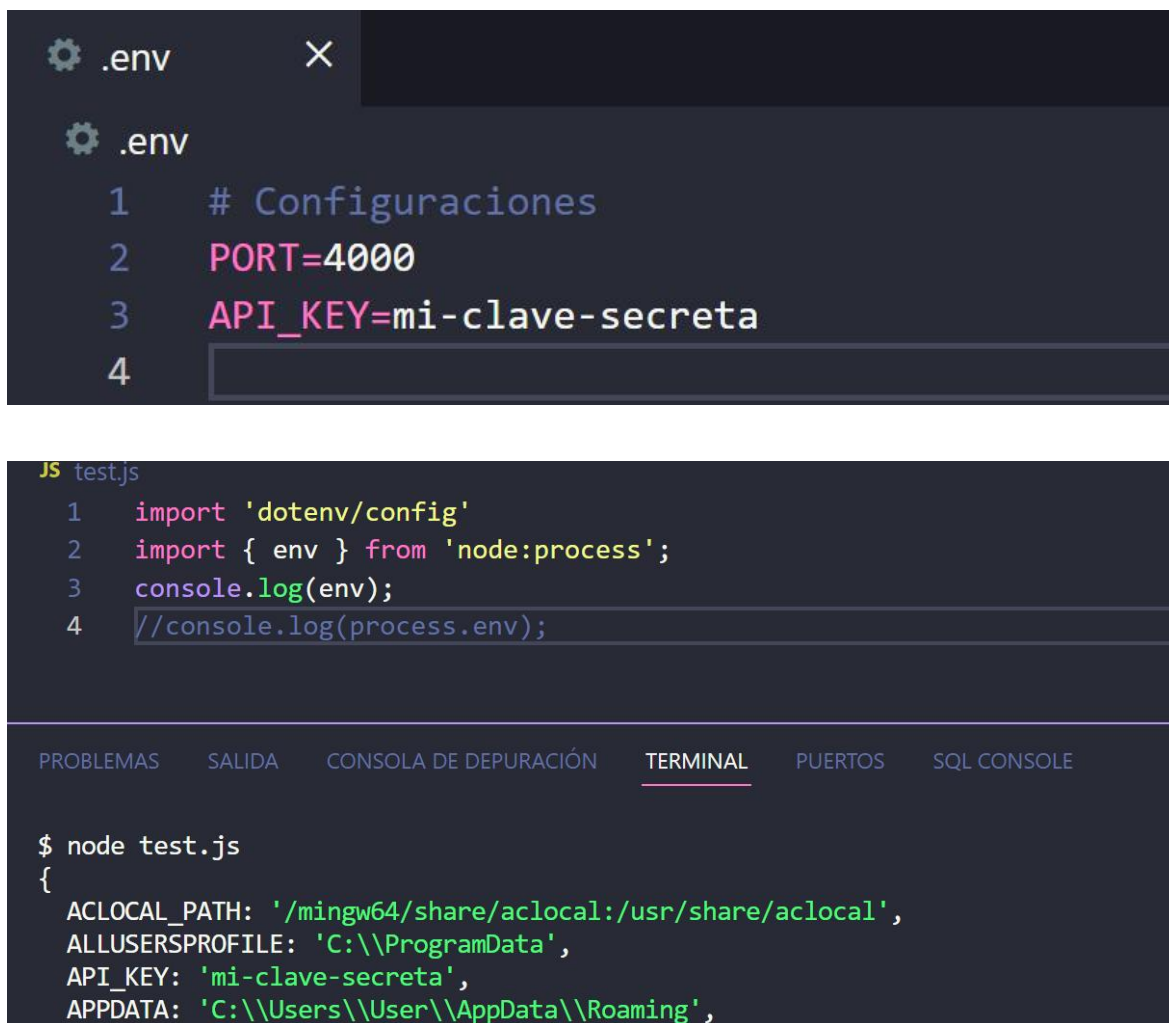
Definición o concepto: Las variables de entorno son valores externos al código fuente que configuran el comportamiento de la aplicación. Permiten:

- Cambiar configuraciones sin modificar el código.
- Mantener credenciales sensibles fuera del repositorio.
- Ajustar el entorno (desarrollo, pruebas, producción) sin hardcodear valores.

Uso:

- Se leen desde process.env en Node.js.
- Pueden definirse en el sistema operativo, en archivos .env (con librerías como dotenv), o en el entorno de despliegue (ej: Heroku, Docker).

Ejemplo con .env:



```
.env
# Configuraciones
PORT=4000
API_KEY=mi-clave-secreta

JS test.js
1 import 'dotenv/config'
2 import { env } from 'node:process';
3 console.log(env);
4 //console.log(process.env);

$ node test.js
{
  ACLOCAL_PATH: '/mingw64/share/aclocal:/usr/share/aclocal',
  ALLUSERSPROFILE: 'C:\\ProgramData',
  API_KEY: 'mi-clave-secreta',
  APPDATA: 'C:\\Users\\User\\AppData\\Roaming',
```