

Introducción a NodeJS

Objetivos

- Conocer que es NodeJS y que podemos hacer con el
- Ejecutar nuestro primer script con NodeJS
- ¿Qué es Npm?
- File System
- Consideraciones y recomendaciones al usar File System
- Argumentos por línea de comando

¿Qué es NodeJS?

Node.js es esencialmente un entorno de ejecución para JavaScript en el lado del servidor que utiliza el motor de JavaScript V8 de Google Chrome. Lo que lo hace único es su capacidad para ejecutar código de manera asincrónica, lo que significa que puede manejar múltiples operaciones simultáneamente sin bloquear el hilo de ejecución principal. Esto lo hace especialmente efectivo para aplicaciones en tiempo real y para manejar grandes cantidades de solicitudes concurrentes.

Una de las características distintivas de Node.js es su modelo de E/S sin bloqueo, que utiliza eventos y devoluciones de llamada para gestionar operaciones asíncronas. Esto permite que Node.js sea extremadamente eficiente en términos de recursos y escalabilidad.

Además, Node.js cuenta con un administrador de paquetes llamado npm (Node Package Manager), que facilita la instalación y gestión de bibliotecas y dependencias externas.

Que puedo hacer con NodeJS

- **Crear servidores:** De forma nativa con el módulo http o con frameworks como Express.js
- **Crear y consumir APIs REST:** Para consumirlas podríamos usar **AXIOS** al igual que en el lado del cliente y para crearlas de forma nativa o con Express.js.
- **Conexiones con bases de datos:** Con módulos como **mongoose** o **pg** podremos conectarnos con motores como MongoDB o PostgreSQL, y en sí con diferentes bases de datos tanto relacionales como no relacionales.

- **Aplicaciones multiplataformas:** Con integraciones de diferentes SDK's, librerías y frameworks, podremos exportar aplicaciones híbridas de teléfonos y de escritorio creadas con tecnologías web.
- **Chats:** Generando persistencia de datos para los mensajes, podríamos crear un chat en tiempo real con el uso de sockets entre diferentes usuarios con la tecnología socket.io.
- **Conexiones con electrónica:** Con módulos como serialport se pueden hacer conexiones con sistemas arduinos.
- **Gestión de archivos:** Podemos hacer un CRUD con archivos del sistema operativo con el módulo **File System**.
- **Testing de aplicaciones:** Al igual que en el lado del frontend, podríamos hacer testing en el lado del backend con las conocidas librerías **mocha** y **jest**.
- **Envíos de correos electrónicos:** Conectados a diferentes proveedores de correos electrónicos y con la ayuda de una interesante variedad de paquetes en NPM, podemos enviar correos electrónicos desde Node.

JavaScript en el navegador VS JavaScript en el servidor

Existen diferencias importantes entre lo que podemos hacer al escribir código JavaScript en el navegador versus código escrito en el servidor con Node Js.

De las principales diferencias está la inexistencia del objeto global window en el lado del servidor, debido a que no existe un árbol de nodos ni elementos HTML que manipular, así como tampoco el uso de CSS.

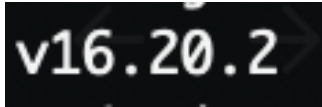
Los métodos para interactuar con los usuarios en los navegadores como alert() o prompt() son exclusivos del desarrollo frontend. Mientras que en Node js contamos con módulos como fs (File System) o la programación modular por defecto.

Ejecutando nuestro primer script

Antes de ejecutar nuestro primer script nos debemos asegurar de tener NodeJS instalado para esto abre la terminal que prefieras y ejecuta la siguiente línea de comando:

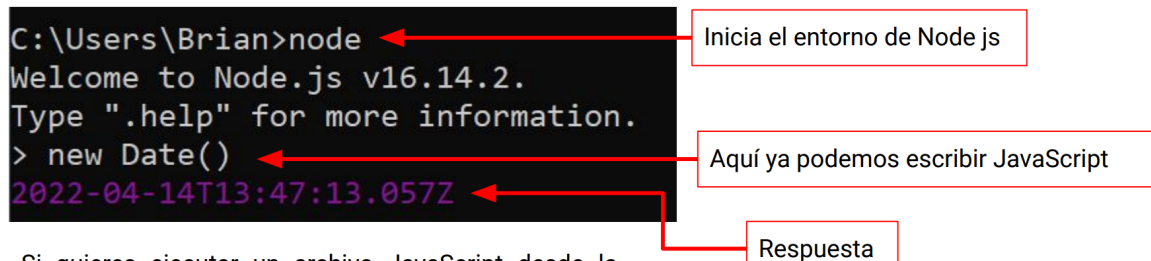


Si Node está instalado en nuestro sistema deberíamos ver una respuesta similar a esta:



```
v16.20.2
```

Para ejecutar JavaScript desde la terminal podemos simplemente ejecutar la palabra node.

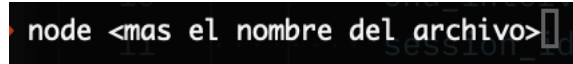


```
C:\Users\Brian>node
Welcome to Node.js v16.14.2.
Type ".help" for more information.
> new Date()
2022-04-14T13:47:13.057Z
```

Annotations:

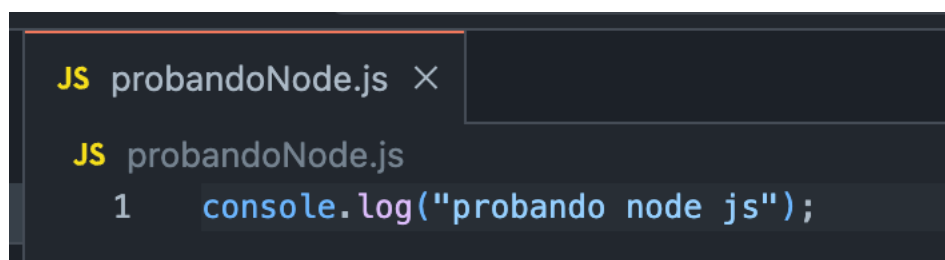
- Inicia el entorno de Node.js (pointing to 'node')
- Aquí ya podemos escribir JavaScript (pointing to 'new Date()')
- Respuesta (pointing to '2022-04-14T13:47:13.057Z')

Si quieres ejecutar un archivo JavaScript desde la terminal, deberás ubicarte en la carpeta donde esté este archivo y escribir:



```
node <mas el nombre del archivo>
```

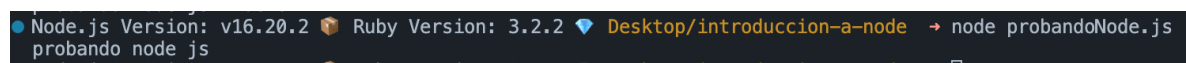
Hagamos una prueba, para esto crea una carpeta y en su interior, crea un archivo con el nombre **probandoNode.js**, podemos agregar un `console.log` con un texto en su interior, posterior a eso vamos a abrir una terminal, luego vamos a navegar hasta el directorio que acabamos de crear y vamos a ejecutar node, como en la instrucción anterior y veras el resultado del texto agregado en la consola.



```
JS probandoNode.js
JS probandoNode.js
1 console.log("probando node js");
```



```
/introduccion-a-node -> node probandoNode.js
```



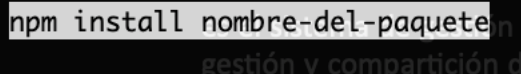
```
Node.js Version: v16.20.2 Ruby Version: 3.2.2 Desktop/introduccion-a-node -> node probandoNode.js
probando node js
```

Npm (Node Package Manager)

es el sistema de gestión de paquetes para Node.js. Fue creado para facilitar la instalación, gestión y compartición de módulos y bibliotecas de JavaScript en el entorno de desarrollo de Node.js. npm se instala automáticamente cuando descargas e instalas Node.js en tu sistema.

Las principales funciones de npm incluyen:

- **Instalación de Paquetes:** npm permite instalar paquetes y módulos de JavaScript de manera sencilla. Puedes instalar un paquete específico o, al proporcionar un archivo package.json, instalar todas las dependencias enumeradas en ese archivo. Para instalar un paquete solo debes estar en la carpeta del proyecto y ejecutar lo siguiente en tu terminal.

A screenshot of a terminal window with a black background. The text 'npm install nombre-del-paquete' is displayed in a light gray font. Below it, the text 'gestión y compartición d' is partially visible in a smaller, lighter font.

- **Gestión de dependencias:** npm ayuda a gestionar las dependencias de un proyecto. Estas dependencias pueden especificarse en el archivo package.json, y npm se encargará de instalar las versiones correctas de cada paquete.
- **Scripts y Comandos:** npm permite definir scripts en el archivo package.json que se pueden ejecutar con comandos específicos. Esto facilita la automatización de tareas comunes en el desarrollo.

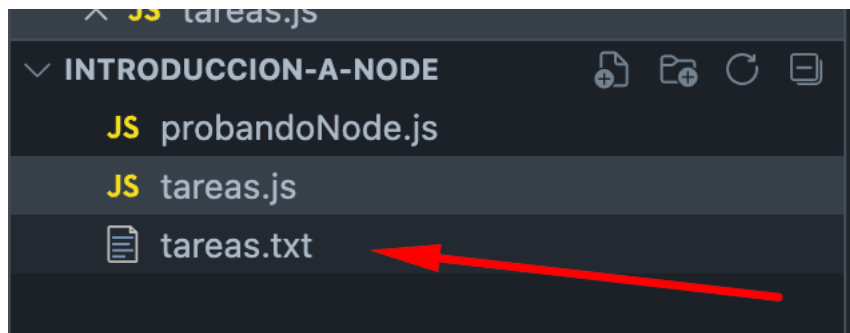
File System

El módulo fs en Node.js facilita la interacción con archivos y directorios. Al incluir este módulo en tu aplicación, puedes realizar tareas como leer el contenido de un archivo, escribir en un archivo, crear y eliminar directorios, entre otras operaciones relacionadas con el sistema de archivos del servidor.

Veamos un ejemplo, creando un archivo de texto que almacene tareas usando el método **writeFileSync**, para esto vamos a crear un archivo llamado tareas.js en la misma carpeta en la que estamos practicando y escribiremos el siguiente código:

```
const fs = require("fs");
const tareas = `
1. Ir al gimnasio
2. Buscar al niño al colegio
3. Pagar los gastos comunes
`;
fs.writeFileSync("tareas.txt", tareas);
```

Para finalizar ejecutamos en una terminal `node tareas.js` y veremos cómo se crea el archivo `tareas.txt`.



Para entender mejor cómo sucedió lo anterior, veamos lo siguiente:

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

Se importa el módulo `fs` (File System)
Esta forma de importaciones se llama **CommonJS**.

```
const tareas = `
1. Ir al gimnasio
2. Buscar al niño al colegio
3. Pagar los gastos comunes
`
```

Se prepara el contenido del archivo a crear

```
fs.writeFileSync('tareas.txt', tareas)
```

Se ejecuta el método `writeFileSync` de File System

El método `writeFileSync` tiene la siguiente anatomía:

```
fs.writeFileSync( "ruta y nombre del archivo", "contenido del archivo y su extension" )
```

Ambos argumentos deben ser estrictamente en formato `String`. Si quisiéramos almacenar un objeto o un arreglo, debemos convertirlo primero en `String` para poder usarlos como contenido. Para esto podemos ocupar el método `JSON.stringify()`.

Ahora procedamos a crear un nuevo archivo de tareas pero ahora en formato `JSON` y escribiremos el siguiente código.

```
const fs = require('fs')
const tareas = [
  { text: 'Ir al gimnasio' },
  { text: 'Buscar al niño al colegio' },
  { text: 'Pagar los gastos comunes' },
]
fs.writeFileSync('tareas.json', JSON.stringify(tareas))
```

Al

ejecutar el script anterior, verás cómo se crea un archivo `tareas.json` que contiene el arreglo de objetos de las tareas.

Lectura de archivos con el módulo File System

Así como podemos crear archivos con el método `writeFileSync`, también podemos leerlos con el método `readFileSync`.

Este método tiene la siguiente anatomía:

```
fs.readFileSync( "ruta y nombre del archivo", "codificación del contenido" )
```

Por defecto, siempre especificaremos como codificación `'UTF8'`, puesto que esta es la codificación de caracteres para nuestro idioma.

Para probar la lectura de archivos vamos a leer el archivo `tareas.txt` que creamos anteriormente, para esto vamos a crear un archivo llamado `lecturaTareas.js` y escribiremos el siguiente código.

```
const fs = require('fs')
const tareas = fs.readFileSync('tareas.txt', 'utf8')
console.log(tareas)
```

Al ejecutarlo en la terminal nos debería mostrar el resultado de nuestra lectura.

```
Node.js Version: v16.20.2 Ruby Version: 3.2.2 Desktop/introduccion-a-node → node lecturaTareas.js
1. Ir al gimnasio
2. Buscar al niño al colegio
3. Pagar los gastos comunes
```

En el caso de que estemos leyendo un archivo JSON debemos transformar el contenido o parsear los datos a un objeto JavaScript de lo contrario obtendremos un error.

Ej:

```
const fs = require('fs')
const tareas = fs.readFileSync('tareas.json', 'utf8')
console.log(JSON.parse(tareas));
```

Consideraciones y recomendaciones al usar File System

El módulo fs (File System) en Node.js es poderoso y flexible, pero hay algunas limitaciones y consideraciones importantes a tener en cuenta al trabajar con él:

- **Operaciones Síncronas Bloqueantes:** Las operaciones síncronas en el módulo fs son bloqueantes, lo que significa que el hilo principal de Node.js se bloqueará hasta que se complete la operación. Esto puede afectar el rendimiento de la aplicación, especialmente en escenarios con muchas operaciones de lectura/escritura concurrentes.
- **Manejo de Errores:** Las operaciones asíncronas en fs utilizan devoluciones de llamada para manejar errores. Olvidar gestionar errores adecuadamente puede llevar a problemas difíciles de diagnosticar. Es crucial incluir manejo de errores en todas las operaciones asíncronas.

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

- **Acceso a Rutas Relativas o Absolutas:** La representación de las rutas de archivos puede variar entre sistemas operativos. Utiliza métodos como `path.join()` para construir rutas de manera que sean compatibles con múltiples plataformas.

Importación y exportación de Módulos

Node.js utiliza un sistema de módulos que permite dividir el código en archivos más pequeños y organizados. La importación y exportación de módulos es esencial para compartir y reutilizar código en aplicaciones Node.js.

Nos referiremos como módulos a los archivos que estén exportando variables o funciones. Ocupando esta práctica gozaremos de un mayor orden y manejo de todo el código que escribimos.

Las instrucciones o herramientas que nos ayudarán a aplicar la programación modular serán:

- `require()`
- `module.exports`

Siguiendo el estándar CommonJS que viene por defecto en Node.js

Vamos crear nuestros primeros módulos y veremos cómo va quedando más limpio nuestro proyecto.

Para lo anterior vamos a crear un archivo `index.js` y otro archivo que vamos a nombrar `modales.js` en el cual vamos a crear dos funciones que luego vamos a importar y ejecutar desde `index.js`

Nuestro código se debería ver así:

```
1  const saludar = (nombre) => {  
2    console.log(`Hola ${nombre}, ¿cómo estás? `);  
3  };  
4  const darLasGracias = (nombre) => {  
5    console.log(`Muchas gracias, ${nombre}`);  
6  };  
7  module.exports = { saludar, darLasGracias };  
8  |  
modales.js
```



```
const saludar = (nombre) => {  
  console.log(`Hola ${nombre}, ¿cómo estás? 😊`)  
}  
  
const darLasGracias = (nombre) => {  
  console.log(`Muchas gracias, ${nombre}`)  
}  
  
module.exports = { saludar, darLasGracias }
```

Con **module.exports** podemos declarar qué variables o funciones queremos exportar de este archivo.

Es recomendable contener todas las exportaciones en un objeto para facilitar la importación de los mismos.

Ahora en el index.js, así como importamos el módulo fs, importemos las funciones del archivo modales.js. Para esto solo necesitamos requerir el archivo y desestructurar su contenido.

```
index.js / ...  
  
const { saludar, darLasGracias } = require('./modales.js')  
saludar('Gonzalo')  
darLasGracias('Javiera')
```

index.js

Al ejecutarlo con node deberíamos obtener el siguiente resultado:

```
Node.js Version: v16.20.2 📦 Ruby Version: 3.2.2 💎 Desktop/introduccion-a-node → node index.js  
Hola Gonzalo, ¿cómo estás?  
Muchas gracias, Javiera
```

Argumentos por línea de comando

En programación, los "argumentos por línea de comandos" se refieren a los valores que se pasan a un programa o script cuando se inicia desde la línea de comandos o terminal. Estos argumentos permiten a los usuarios proporcionar información específica al programa al momento de la ejecución.

En el contexto de Node.js, los argumentos de línea de comandos son accesibles a través del objeto process.argv. Este objeto proporciona una matriz que contiene los argumentos pasados al programa Node.js cuando se ejecuta.

Estructura de process.argv:

La matriz process.argv tiene al menos tres elementos:

- Índice 0: Ruta al ejecutable de Node.js.
- Índice 1: Ruta al script que se está ejecutando.
- A partir del Índice 2: Los argumentos proporcionados por el usuario.

Ejemplo Práctico:

Supongamos que tienes un script Node.js llamado lineaDeComandos.js que realiza alguna operación en base a los argumentos proporcionados:

```
1 // Imprimir todos los argumentos
2 console.log(process.argv);
3
4 // Obtener argumentos específicos
5 const primerArgumento = process.argv[2];
6 const segundoArgumento = process.argv[3];
7
8 console.log(`Primer argumento: ${primerArgumento}`);
9 console.log(`Segundo argumento: ${segundoArgumento}`);
10
```

Desde la línea de comandos, puedes ejecutar este script con argumentos personalizados:

```
Node.js Version: v16.20.2 Ruby Version: 3.2.2 Desktop/introduccion-a-node → node app.js argumento1 argumento2
```

Esto producirá la siguiente respuesta:

```
Node.js Version: v16.20.2 Ruby Version: 3.2.2 Desktop/introduccion-a-node → node lineaDeComandos.js argumento1 argumento
[
  '/Users/fabianpino/.nvm/versions/node/v16.20.2/bin/node',
  '/Users/fabianpino/Desktop/introduccion-a-node/lineaDeComandos.js',
  'argumento1',
  'argumento'
]
Primer argumento: argumento1
Segundo argumento: argumento
```