

Guía de estudio 1- Introducción a Node



¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo practicar y ejercitar los contenidos que hemos visto en clase, además de profundizar temas adicionales que complementan aquellos que revisamos.

Dentro de los cuales se encuentran:

- Ejecutar scripts con Node Js desde la terminal.
- Crear archivos con el módulo File System.
- Leer archivos con el módulo File System.
- Importar y exportar módulos en Node Js.

¡Vamos con todo!



Tabla de contenidos

¿Qué es Node?	3
Node y npm	4
Revisando la instalación	4
¡Hola Mundo!	4
¡Manos a la obra! - Fecha por consola	4
Creación de archivos	5
¡Manos a la obra! - Método WriteFileSync .doc	7
¡Manos a la obra! - Método WriteFileSync .xls	8
¡Manos a la obra! - Autos.json	8
Sobrescritura de archivos	9
¡Manos a la obra! - Autos.json	10
Lectura de archivos	10
¡Manos a la obra! - readFileSync	11
Lectura y manipulación de archivos JSON	12
¡Manos a la obra! - script con readFileSync	13
Importación y exportación de módulos	13
¡Manos a la obra! - Operaciones.js	16
Argumentos por línea de comando	16
¡Manos a la obra! - Operaciones	18
Realicemos un check de los aprendizajes claves.	18



¡Comencemos!

¿Qué es Node?

Node, es un entorno de ejecución para crear aplicaciones en el lado del servidor.

A continuación te mostramos una lista de algunos de los problemas o necesidades que podrías resolver desarrollando bajo este entorno:

- **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.

Node y npm

Al instalar Node Js estamos también instalando su gestor de paquetes por defecto llamado **NPM**, con el que podremos instalar nuevas funcionalidades o módulos, además de manejar las dependencias del proyecto sin necesidad de una herramienta externa.

Revisando la instalación

A continuación empezaremos a realizar ejercicios prácticos, por lo que te recomendamos asegúrate de tener instalado Node en tu sistema operativo.

Para comprobar que Node está instalado abre la terminal y ejecuta `node --version`, si recibes una versión como respuesta quiere decir que tienes instalado Node correctamente, de lo contrario deberás descargarlo de su [página oficial](#) y posteriormente instalarlo.

¡Hola Mundo!

Hagamos nuestro ¡Hola Mundo! con Node Js, para esto:

1. Crea una nueva carpeta en el escritorio
2. Abre la carpeta con VSC
3. Abre la terminal de VSC y agrega un archivo llamado `index.js` y agrega lo siguiente:

```
console.log('Hola Node Js 🙌')
```

Posteriormente, ejecuta este archivo por la terminal y verás lo siguiente:

```
Brian@LAPTOP-7I8PV10A MINGW64 ~/Desktop/Node
$ node index.js
Hola Node Js 🙌
```

Imagen 1. ¡Hola Mundo! con Node

Fuente: Desafío Latam



¡Manos a la obra! - Fecha por consola

Muestra la fecha de hoy por consola con Node Js siguiendo estos pasos:

1. Crea una constante *fecha* que sea instancia del objeto `Date()`

```
const fecha = new Date()
```

2. Usa un `console.log()` para mostrar por consola la constante *fecha*
3. Ejecuta el script por la terminal
4. Observa la respuesta

```
$ node index.js
2022-04-18T19:07:25.465Z
```

Imagen 2. Fecha mostrada por consola
Fuente: Desafío Latam

Creación de archivos

Con Node podemos leer, crear, eliminar y realizar varias otras operaciones relacionadas con el sistema de archivos de nuestro computador a través de un módulo llamado File System que se abrevia como “fs”.

Veamos un ejemplo escribiendo un nuevo script *index.js* que tenga como objetivo crear un archivo de texto usando el método *writeFileSync*:

```
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)
```

Ejecuta este código y verás que ha aparecido un nuevo archivo de nombre *tareas.txt*, ábrelo y encontrarás las tareas declaradas en el script anterior.

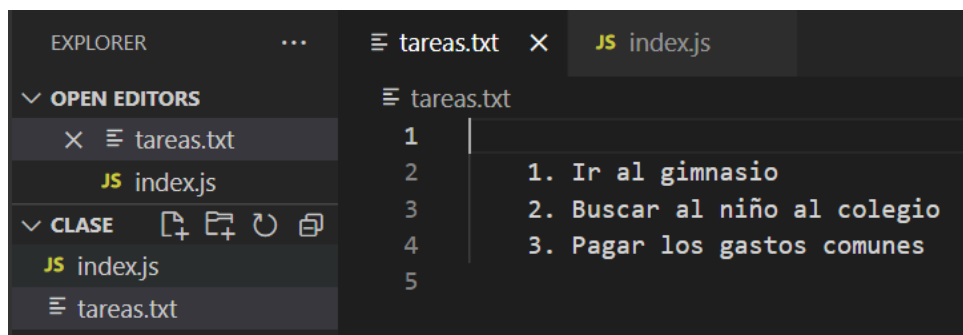


Imagen 3. Tareas declaradas
Fuente: Desafío Latam

Para entender mejor cómo sucedió lo anterior, dividamos el código por partes:

1. Importamos el módulo File System usando el método *require()*

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

2. Preparamos el contenido del archivo a crear

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

3. Ejecutamos el método *writeFileSync* de la instancia de FileSystem

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

Como resultado, con muy pocas líneas de código al ejecutar este script, se creará un archivo nuevo en la carpeta llamado *tareas.txt* cuyo contenido es de tipo **String**.

Ahora procederemos a crear un nuevo archivo de tareas, pero ahora usando un arreglo de objetos como contenido.

Para esto utiliza este nuevo script:

```
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) )
```



El método `JSON.stringify()` convertirá a formato JSON cualquier arreglo u objeto que se le asigne como argumento.

Ahora ejecuta el script anterior y verás cómo se crea un nuevo archivo de nombre *tareas.json* que contiene el arreglo de objetos de las tareas

```
{ } tareas.json X JS index.js
{ } tareas.json > ...
1 [
2   { "text": "Ir al gimnasio" },
3   { "text": "Buscar al niño al colegio" },
4   { "text": "Pagar los gastos comunes" }
5 ]
6
```

Imagen 3. Tareas declaradas
Fuente: Desafío Latam



El **JSON.stringify()** nos permite convertir un arreglo o un objeto en un String en formato JSON, de esta manera podremos almacenarlo en un archivo o compartirlo con otras aplicaciones.



¡Manos a la obra! - Método WriteFileSync .doc

Crea un nuevo archivo usando el método *writeFileSync* con extensión **.doc** para abrirlo con Word o Google Docs o cualquier otro editor de texto:

1. Crea una constante con un String "Node es genial!".
2. Usa el método *writeFileSync* para crear un archivo de nombre *test.doc* usando la constante creada como contenido.
 - a. En esta ocasión no es necesario usar el `JSON.stringify()` ya que estamos directamente pasando un String.
3. Ejecuta el script por la terminal.
4. Usa Word o Google Drive o cualquier otro editor para abrir el archivo creado.

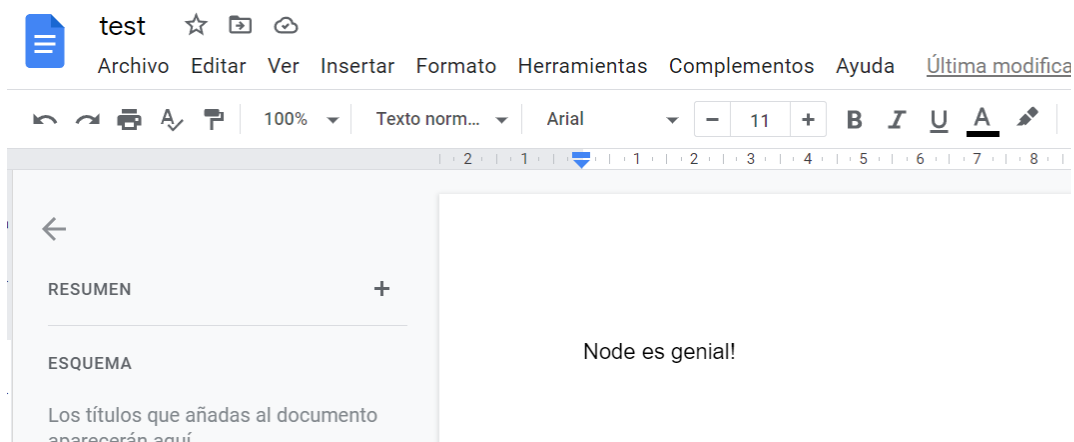


Imagen 4. Editor de textos de Google
Fuente: Desafío Latam



¡Manos a la obra! - Método WriteFileSync .xls

Crea un nuevo archivo usando el método `writeFileSync` con extensión `.xls` para abrirlo con Excel o Google Sheets:

1. Crea una constante con un String "Node Js \t Express Js".
2. Usa el método `writeFileSync` para crear un archivo de nombre `test.xls` usando la constante creada como contenido.
3. Ejecuta el script por la terminal.
4. Usa Excel o Google Sheets para abrir el archivo creado.

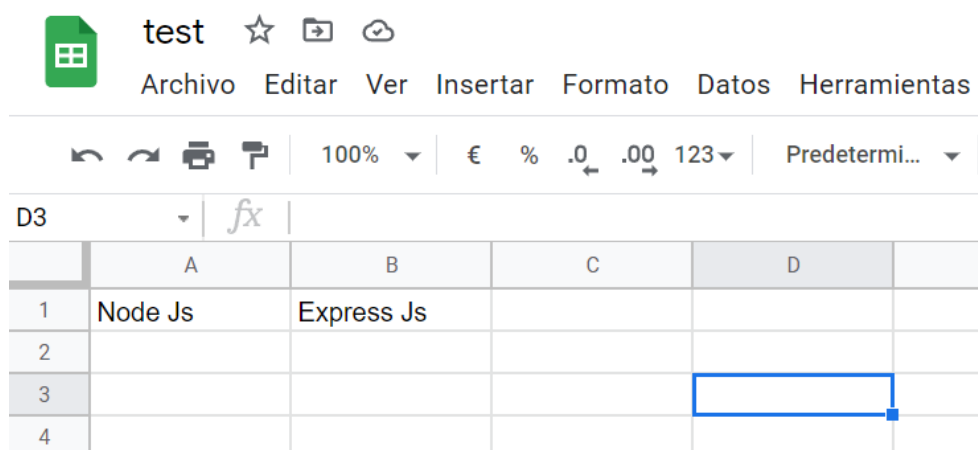


Imagen 4. Hoja de cálculo de Google
Fuente: Desafío Latam

La razón por la que la aplicación de hoja de cálculo puede abrir este archivo se debe a que el contenido está en un formato compatible CSV que ocupa la tabulación como separado.



Si tu editor no puede abrir el archivo, intenta importarlo como CSV.



¡Manos a la obra! - Autos.json

Ejecuta un script que cree un archivo `autos.json` con el método `writeFileSync` y el siguiente arreglo de objetos:

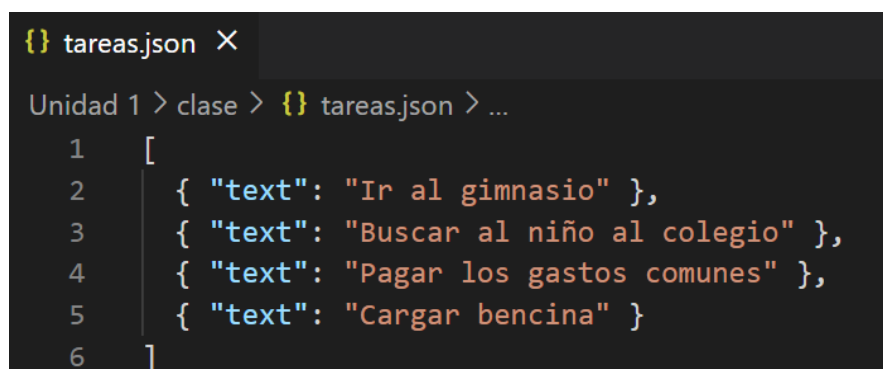

```
const autos = [  
  {  
    marca: 'Susuki',  
    modelo: 'Kicks',  
  },  
  {  
    marca: 'Toyota',  
    modelo: 'Corola',  
  },  
  {  
    marca: 'GAC',  
    modelo: 'GS4',  
  }  
]
```

Sobrescritura de archivos

Cuando queramos actualizar el contenido de un archivo, utilizaremos nuevamente el `writeFileSync` para sobrescribirlo con los nuevos datos.

Probemos sobrescribiendo el archivo `tareas.json` agregando una nueva tarea.

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



```
{ } tareas.json X  
Unidad 1 > clase > { } tareas.json > ...  
1  [  
2    { "text": "Ir al gimnasio" },  
3    { "text": "Buscar al niño al colegio" },  
4    { "text": "Pagar los gastos comunes" },  
5    { "text": "Cargar bencina" }  
6  ]
```

Imagen 5. Archivo sobrescrito.

Fuente: Desafío Latam



¡Manos a la obra! - Autos.json

Ejecuta un script que sobrescriba el archivo *autos.json* con el método *writeFileSync* para agregar un auto más al arreglo de objetos:

```
const autos = [  
  {  
    marca: 'Susuki',  
    modelo: 'Kicks',  
  },  
  {  
    marca: 'Toyota',  
    modelo: 'Corola',  
  },  
  {  
    marca: 'GAC',  
    modelo: 'GS4',  
  },  
  {  
    marca: 'Changan',  
    modelo: 'CS35 Plus',  
  }  
]
```

Lectura de archivos

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

Este método tiene la siguiente anatomía:

```
fs.readFileSync( <nombre del archivo>, <codificación del contenido> )
```

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

Al ejecutar este método, recibiremos como retorno el contenido del archivo.

Veamos un ejemplo leyendo uno de los archivos que creamos previamente, el archivo *tareas.json*.

Para esto crea un nuevo script que contenga el siguiente código:

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

const tareas = fs.readFileSync('tareas.json', 'utf8')

console.log(tareas)
```

El objetivo de este código será mostrar por consola el contenido del archivo *tareas.json*.

Ejecutando el script anterior podremos ver por consola lo siguiente:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Brian@LAPTOP-7I8PV10A MINGW64 ~/Desktop/Node
$ node index.js
[
  { "text": "Ir al gimnasio" },
  { "text": "Buscar al niño al colegio" },
  { "text": "Pagar los gastos comunes" },
  { "text": "Cargar bencina" }
]
```

Imagen 6. Archivo sobreescrito.
Fuente: Desafío Latam

La variable *tareas* almacena el contenido del archivo en string, más adelante en esta guía lo convertiremos en un arreglo.



¡Manos a la obra! - readFileSync

Crea un documento HTML con una estructura básica y luego escribe un script que lea su contenido usando el método *readFileSync*.

```
$ node index
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
</head>

<body></body>

</html>
```

Imagen 7. Archivo html.
Fuente: Desafío Latam

Lectura y manipulación de archivos JSON

JSON o JavaScript object notation es un formato que se guarda como texto plano, al leer uno de estos archivos desde JavaScript o desde cualquier otro lenguaje traeremos el dato como un string el cual posteriormente transformaremos.

Antes de aprender a transformarlo veamos que sucede si no lo hacemos.

```
tareas.forEach( (tarea) => {
  console.log(tarea)
})
```

Recibiremos este mensaje por la terminal:

```
tareas.forEach((tarea) => {
  ^
TypeError: tareas.forEach is not a function
```

Imagen 8. Mensaje de la terminal.
Fuente: Desafío Latam

Lo cual sucede porque tareas es un **String** y este tipo de dato no cuenta con el método *forEach* como los arreglos.



A pesar de esto, en JavaScript contamos con un método para convertir un String de formato JSON a un arreglo u objeto, este método se llama `JSON.parse()` y sirve como la inversa del método `JSON.stringify()`.

Para recorrer correctamente el arreglo de tareas en formato JSON entonces podemos hacer lo siguiente:

```
JSON.parse(tareas).forEach((tarea) => {  
  console.log(tarea)  
})
```

Al ejecutarlo podremos ver que la consola nos mostrará correctamente cada una de las tareas:

```
$ node index.js  
{ text: 'Ir al gimnasio' }  
{ text: 'Buscar al niño al colegio' }  
{ text: 'Pagar los gastos comunes' }  
{ text: 'Cargar bencina' }
```

Imagen 9. Tareas mostradas por consola.
Fuente: Desafío Latam



¡Manos a la obra! - script con `readFileSync`

Escribe un script que use el método `readFileSync` para leer el archivo `autos.json` sobrescrito en el ejercicio. Posteriormente, utiliza el método `forEach` para mostrar por consola el nombre de cada modelo en mayúsculas

```
$ node autos.js  
KICKS  
COROLA  
GS4  
CS35 PLUS
```

Imagen 10. Resultado esperado.
Fuente: Desafío Latam

Importación y exportación de módulos

En la medida que vayamos desarrollando nuestras aplicaciones con Node Js nuestro `index.js` va a crecer en líneas de código y empezará a ser cada vez más incómodo leerlo y ubicar la lógica que vamos agregando.

Para que esto no sea un problema existe una buena e importante práctica que consiste en dividir la lógica de la aplicación en diferentes archivos, en donde cada uno podrá exportar e importar la lógica del otro.

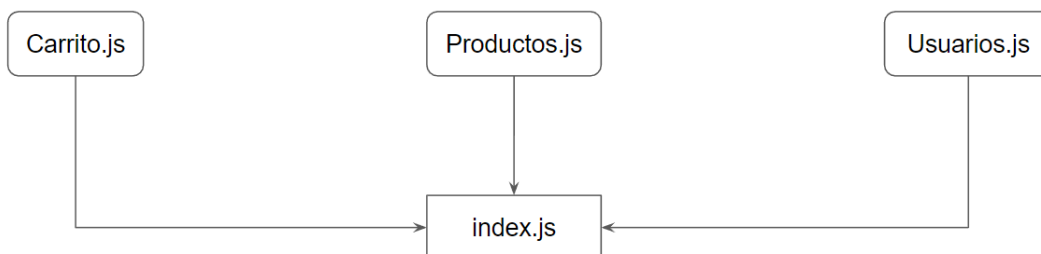


Imagen 11. Lógica de la aplicación dividida.
Fuente: Desafío Latam

Realicemos un ejemplo creando un archivo llamado `modales.js` que contenga 2 funciones para mostrar por consola un saludo cordial y un gracias respectivamente:

```
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 opcionalmente desestructurar su contenido. Con las funciones importadas, probemos ejecutarlas pasando como argumento un nombre aleatorio:

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

Ahora ejecutemos el `index.js` y deberíamos ver lo siguiente por la terminal:

```
Unidad 1 > clase > JS index.js > ...
1  const { saludar, darLasGracias } = require("./modales.js")
2
3  saludar("Gonzalo")
4
5  darLasGracias("Javiera")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Brian@LAPTOP-7I8PV10A MINGW64 ~/Desktop/Node y Express/Unidad 1/clase
$ node index.js
Hola Gonzalo, ¿cómo estás? 😊
Muchas gracias, Javiera
```

Imagen 12. Saludo cordial, por la consola.
Fuente: Desafío Latam.

Realicemos otro ejemplo donde tengamos un archivo `funciones.js` que contenga una única función para sobrescribir y vaciar los datos de un archivo JSON, es decir, reasignar su contenido a un arreglo vacío `[]`

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

const vaciarJSON = (nombreDelArchivo) => {
  fs.writeFileSync(nombreDelArchivo, '[]')
}

module.exports = { vaciarJSON }
```

Ahora en nuestro `index.js` importemos la función `vaciarJSON` y ejecutémosla pasando como argumento el nombre del archivo `autos.json`

```
const { vaciarJSON } = require('./funciones')

vaciarJSON("autos.json")
```

Ahora si abrimos manualmente el archivo `autos.json` veremos que fue vaciado y ahora solo contiene un arreglo vacío.

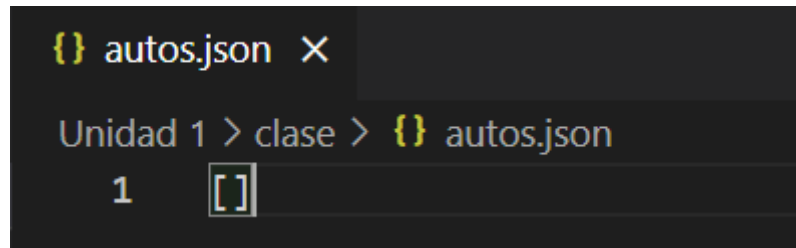


Imagen 13. Arreglo vacío.
Fuente: Desafío Latam.



¡Manos a la obra! - Operaciones.js

1. Crea un archivo **Operaciones.js** que cree las siguientes funciones:
 - **createFile**: Recibe 2 argumentos (nombre del archivo, contenido del archivo), crea el archivo y muestra por consola un mensaje indicando que el archivo fue creado.
 - **readFile**: Recibe 1 argumento (nombre del archivo), lee el archivo con *readFileSync* y muestra su contenido por consola.
2. Exporta ambas funciones.
3. Crea un archivo **index.js** que:
 - Importe las funciones de *operaciones.js*
 - Pruebe ambas funciones importadas

Argumentos por línea de comando

Con Node podemos capturar los argumentos que se escriben en la terminal. Para esto es necesario utilizar un objeto global llamado *process* y acceder a un atributo *argv*.

Este atributo nos devolverá un arreglo con todos los argumentos escritos por la terminal al ejecutar un script.

Veamos un ejemplo creando un nuevo script con el siguiente código:

```
const argumentos = process.argv.slice(2)

argumentos.forEach(argumento => {
```



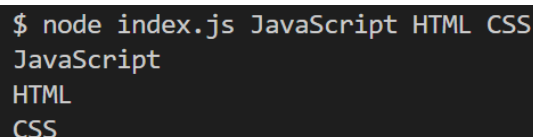
```
    console.log(argumento)  
  })
```

Usaremos el método `slice` para recortar el arreglo omitiendo los 2 primeros argumentos, ya que estos serán la 'node' y el 'nombre del archivo'.

Ahora ejecutemos el script anterior utilizando la siguiente línea de comando:

```
node index.js JavaScript HTML CSS
```

Y veremos lo siguiente en nuestra terminal:



```
$ node index.js JavaScript HTML CSS  
JavaScript  
HTML  
CSS
```

Imagen 14. Terminal con script ejecutado.
Fuente: Desafío Latam.

De esta manera entonces podemos pasarle argumentos a nuestros scripts al momento de ejecutarlos.

Veamos otro ejemplo, en donde utilicemos el archivo *modales.js* para ejecutar sus funciones en nuestro archivo *index.js* pasándole como argumentos el nombre de la persona que queremos saludar y a quien queremos agradecer.

```
const { saludar, darLasGracias } = require('./modales.js')  
  
const argumentos = process.argv.slice(2)  
  
const nombreParaSaludar = argumentos[0]  
const nombreParaAgradecer = argumentos[1]  
  
saludar(nombreParaSaludar)  
  
darLasGracias(nombreParaAgradecer)
```

Ejecutemos el script anterior con la siguiente línea de comando:

```
node index.js Claudia Alejandro
```

Y veremos lo siguiente:

```
Brian@LAPTOP-7I8PV10A MINGW64 ~/Desktop/Node y Express/Unidad 1/clase
$ node index.js Claudia Alejandro
Hola Claudia, ¿cómo estás? 😊
Muchas gracias, Alejandro
```

Imagen 15. Terminal con script ejecutado.
Fuente: Desafío Latam.



¡Manos a la obra! - Operaciones

Basado en el ejercicio anterior, ejecuta las operaciones para crear y leer archivos especificando el nombre y el contenido por línea de comando.

⚠ Si quieres incluir frases como argumentos deberás utilizar las comillas simples (' ')

Por ejemplo:

```
node index.js tareas.txt 'Lavar la ropa'
```

En dónde *tareas.txt* es el nombre del archivo y 'Lavar la ropa' es su contenido.

Realicemos un check de los aprendizajes claves.

1. ¿Qué hace `JSON.stringify` en el siguiente código?

```
const tareas = [
  { text: 'Ir al gimnasio' },
  { text: 'Buscar al niño al colegio' },
  { text: 'Pagar los gastos comunes' },
  { text: 'Cargar bencina' }
]

fs.writeFileSync('tareas.json', JSON.stringify(tareas) )
```

2. ¿Cuál es el problema con el siguiente código?

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

const tareas = fs.readFileSync('tareas.json', 'utf8')
tareas.forEach( (tarea) => {
  console.log(tarea)
})
```

¿Cómo se soluciona?

3. ¿Cómo se deben exportar las siguientes funciones para poder importarlas de esta forma?
- ¿En qué archivo deberían estar?

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

- ¿Puedo utilizar alert o prompt en Node?
- ¿Puedo utilizar el módulo fs en el navegador?