

## TEXT CLASS REVIEW

### TEMAS A TRATAR EN EL CUE:


- Ciclo de vida de un proceso.
- Introducción Módulo HTTP de Node.js.
- Persistencia de datos.
- Argumentos de entrada en programa Node.js.
- Motor de plantillas.

### CICLO DE VIDA

Cuando iniciamos nuestro programa con **node** "index.js", el archivo es primero analizado o **parseado** (parsed). En esta etapa nuestro programa no correrá si es que existe algún error de sintaxis, y una vez que el programa ha sido analizado o **parseado**, todas las funciones y variables quedan guardadas en una reserva de memoria. En este punto, correrá una cierta cantidad de veces, infinitamente, hasta que no existan más tareas en el **loop** de eventos.

Existe otra instancia en donde nuestro programa puede mantenerse funcionando infinitamente, salvo le indiquemos lo contrario. Haremos una pequeña introducción a **node** como servidor HTTP.

Utilizaremos el módulo **HTTP**, el cual viene por defecto instalado en **node**, y el método **createServer()**, para crear nuestro primer servidor. Ejecutaremos el programa.

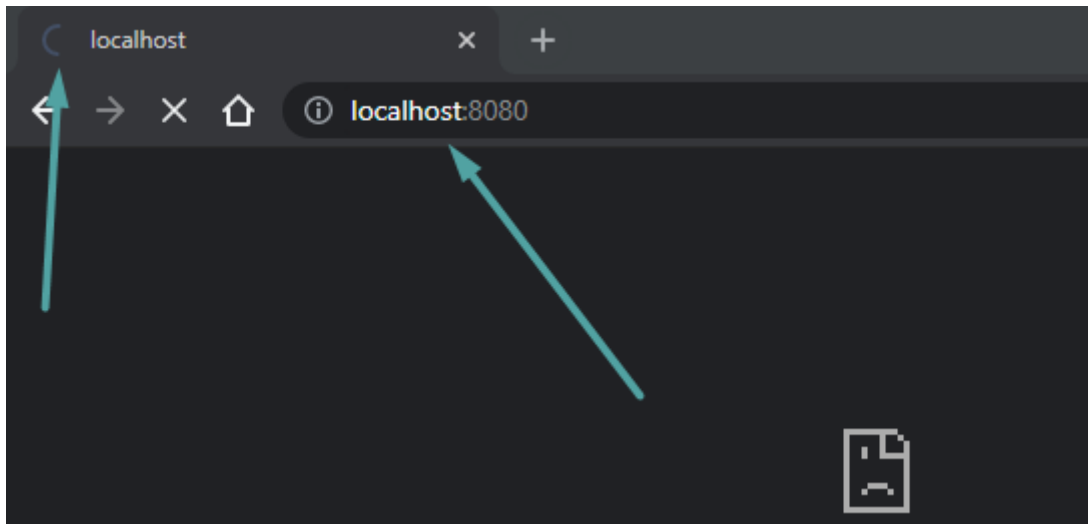


```
JS index.js  X
JS index.js > ...
1  const http = require('http');
2
3  const server = http.createServer((req,res) => {
4    console.log('Servidor escuchando en puerto 8080')
5  })
6
7  server.listen(8080)

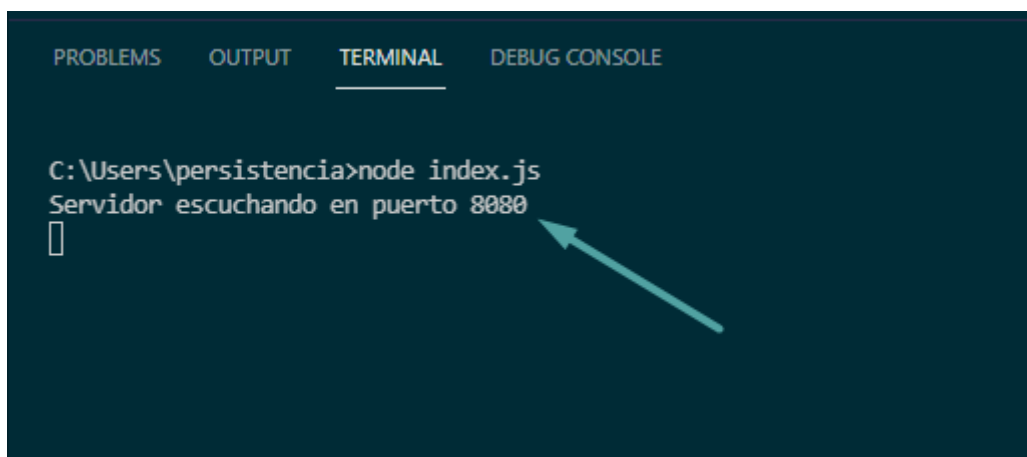
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia>node index.js
█
```

Recuerda que un servidor siempre está escuchando y esperando peticiones, en este caso, la dirección por defecto a consultar es en `"localhost:{puerto}"`, donde el puerto será aquel que especificamos al utilizar el método `listen()`.



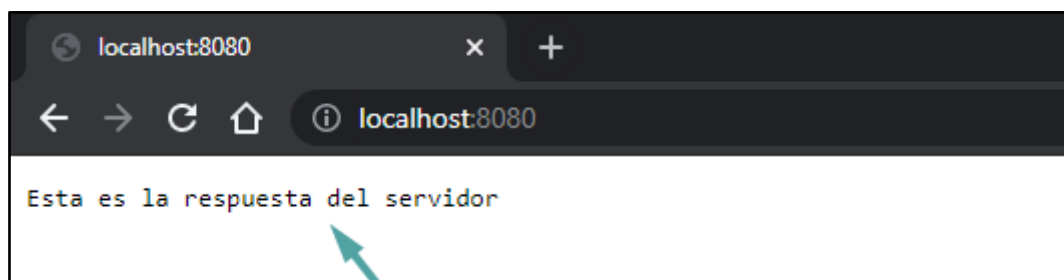
Al hacer la consulta en el navegador, podemos ver cómo se obtiene el mensaje en la consola.



Nuestro navegador se queda cargando, puesto que está esperando una respuesta; para enviar una desde nuestro servidor, utilizamos el método `write` y `end`, contenidos en el objeto de respuesta.

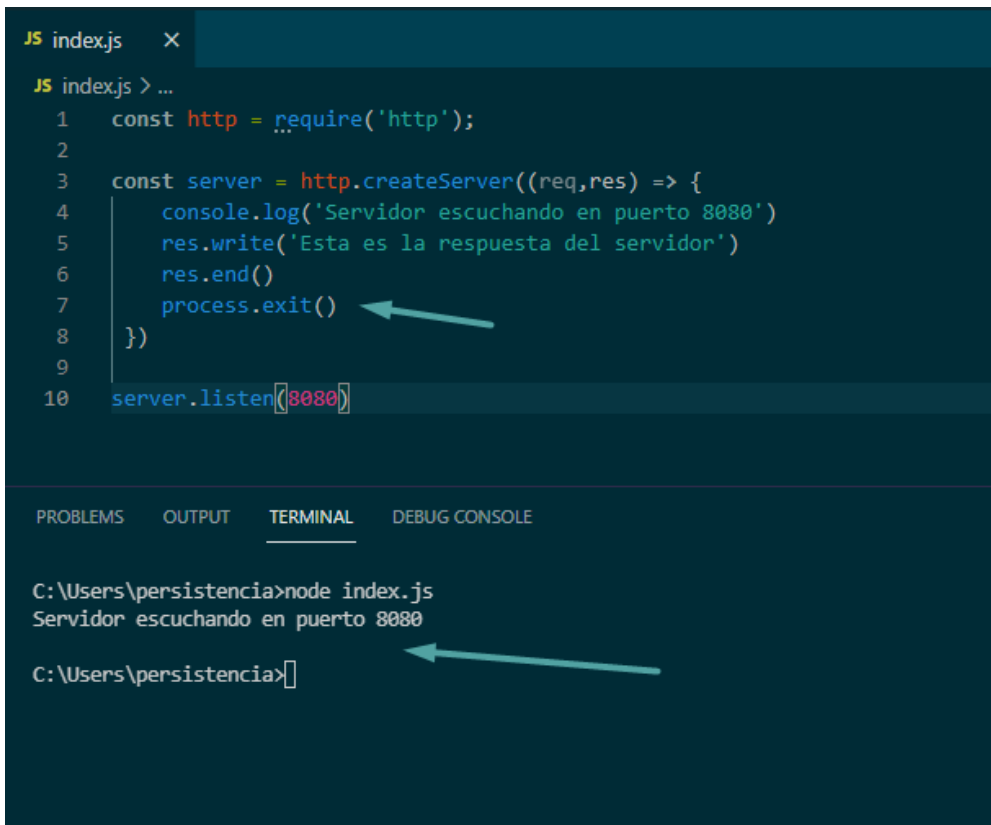
```
JS index.js  X
JS index.js > ...
1  const http = require('http');
2
3  const server = http.createServer((req, res) => {
4    console.log('Servidor escuchando en puerto 8080')
5    res.write('Esta es la respuesta del servidor')
6    res.end()
7  })
8
9  server.listen(8080)
```

Reiniciamos nuestro servidor luego de guardar los cambios (terminando la ejecución por consola, utilizando las teclas `ctrl + c`, y volviendo a ejecutar el programa), y ahora podemos ver que éste obtiene la respuesta que enviamos desde el programa.



Hemos creado un servidor muy simple, el cual puede llegar a ser muy poderoso. Dentro de los próximos CUEs trataremos más sobre `node js` como servidor.

Por ahora, ya sabemos que este servidor seguirá corriendo hasta que nosotros terminemos el programa, ya sea desde la consola, o de manera programática, utilizando el método `process.exit()`. Al momento de hacer una consulta nuevamente en tu navegador, verás como el proceso termina.



```
JS index.js X
JS index.js > ...
1  const http = require('http');
2
3  const server = http.createServer((req,res) => {
4    console.log('Servidor escuchando en puerto 8080')
5    res.write('Esta es la respuesta del servidor')
6    res.end()
7    process.exit()
8  })
9
10 server.listen(8080)
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\persistencia>node index.js
Servidor escuchando en puerto 8080
C:\Users\persistencia>
```

El código anterior se ejecutará una vez, justo al momento de recibir una petición desde el navegador, y luego el proceso se terminará.

Al terminar un proceso o programa en Node, significa que todos los resultados que deriven de éste se reiniciarán, y por lo tanto, no tenemos una sensación de persistencia de datos, sino una re-ejecución para obtener los mismos, considerando que nuestro código no ha sufrido cambios.

Es complicado que las aplicaciones del mundo real puedan funcionar sin una capa de persistencia de datos, y es aquí donde entran en juego las conexiones hacia base de datos. Por ahora, nosotros simularemos esta persistencia de datos utilizando el módulo `fs`, el cual también nos permite escribir información en archivos de nuestro sistema.

Vamos a crear tres archivos: `escribe.js`, `consulta.js`, y `datos.txt`.

En el archivo `datos.txt`, escribiremos "Mensaje inicial"; luego, en el archivo `consulta.js`, importaremos el módulo `fs`; y utilizaremos el método `readFile()` para obtener los datos del archivo de texto.



```
datos.txt
1 Mensaje inicial

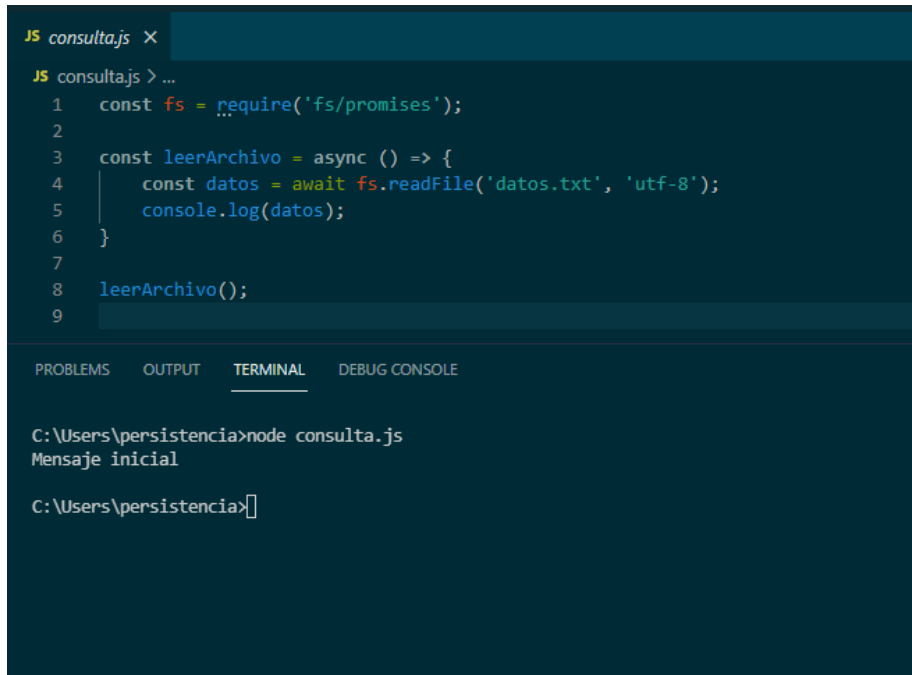
JS consulta.js
1 const fs = require('fs/promises');
2
3 const leerArchivo = async () => {
4   const datos = await fs.readFile('datos.txt', 'utf-8');
5   console.log(datos);
6 }
7
8 leerArchivo();
9
```

Por último, en el archivo `escribe.js`, utilizaremos el método `writeFile()`, el cual es muy similar al uso de `readFile`, con la diferencia que el segundo argumento que recibe el método es el texto que queremos escribir, y el tercer argumento es el tipo de codificación.



```
JS escribe.js
1 const fs = require('fs/promises');
2
3 const escribirArchivo = async () => {
4   try{
5     await fs.writeFile('datos.txt', 'Cambiando informacion de archivo', 'utf-8');
6     console.log("Los datos han sido cambiados exitosamente");
7   } catch (err) {
8     console.log(err);
9   }
10 }
11
12 escribirArchivo();
13
```

En primer lugar, ejecutamos el archivo `consulta.js`, para ver la información actual contenida en el archivo de texto.



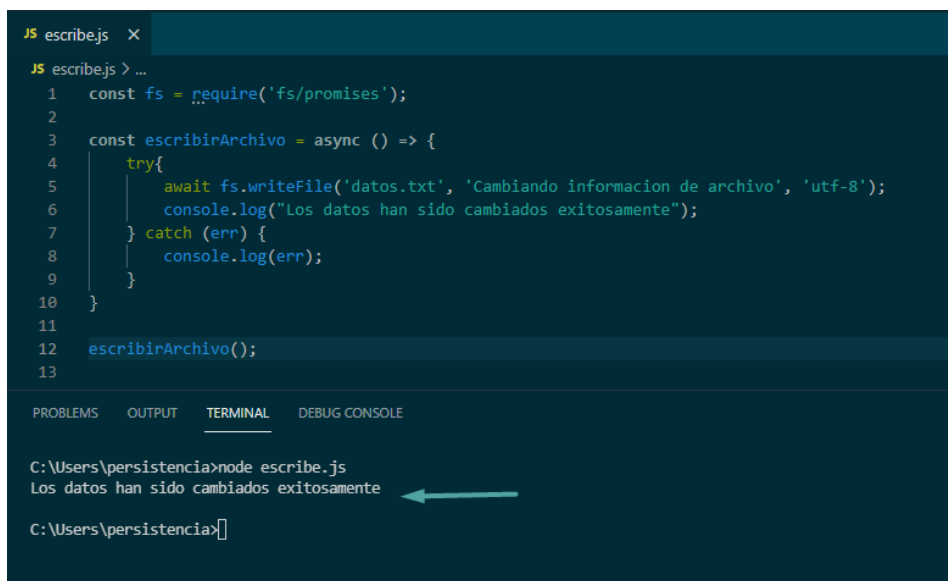
```
JS consulta.js X
JS consulta.js > ...
1  const fs = require('fs/promises');
2
3  const leerArchivo = async () => {
4    const datos = await fs.readFile('datos.txt', 'utf-8');
5    console.log(datos);
6  }
7
8  leerArchivo();
9

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia>node consulta.js
Mensaje inicial

C:\Users\persistencia>
```

Luego, ejecutamos el archivo `escribe.js`, para cambiar los datos dentro del archivo de texto.

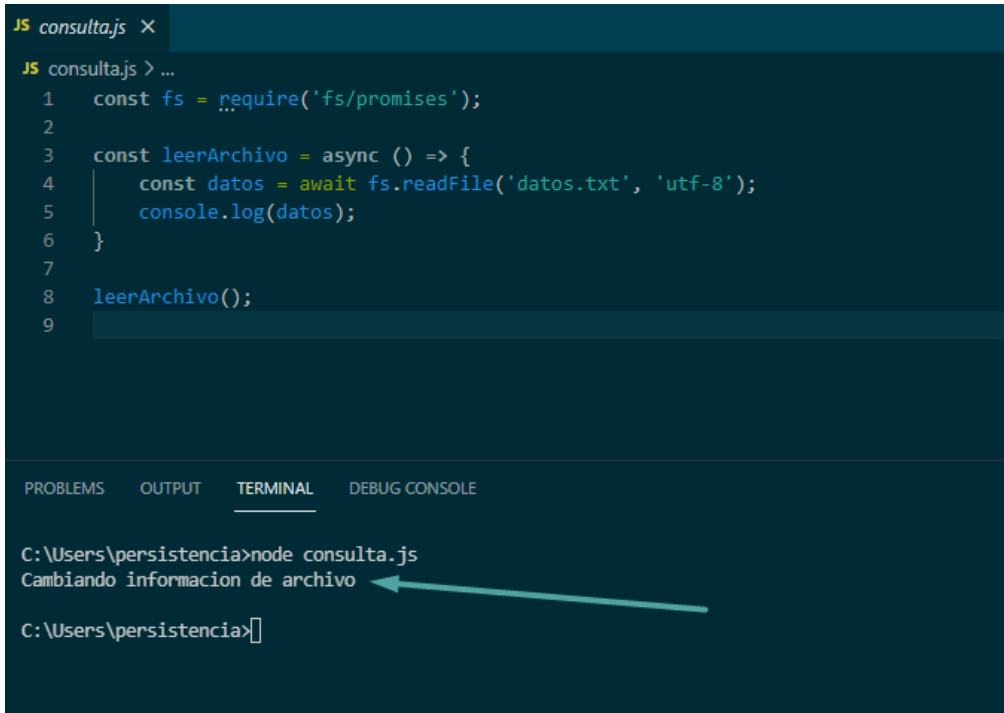


```
JS escribe.js X
JS escribe.js > ...
1  const fs = require('fs/promises');
2
3  const escribirArchivo = async () => {
4    try{
5      await fs.writeFile('datos.txt', 'Cambiando informacion de archivo', 'utf-8');
6      console.log("Los datos han sido cambiados exitosamente");
7    } catch (err) {
8      console.log(err);
9    }
10 }
11
12 escribirArchivo();
13

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia>node escribe.js
Los datos han sido cambiados exitosamente
C:\Users\persistencia>
```

Y volvemos a hacer la consulta.



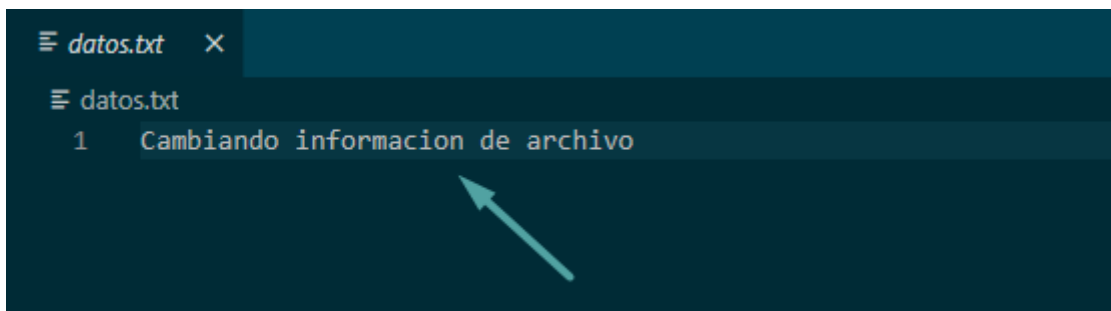
```
JS consulta.js X
JS consulta.js > ...
1  const fs = require('fs/promises');
2
3  const leerArchivo = async () => {
4    const datos = await fs.readFile('datos.txt', 'utf-8');
5    console.log(datos);
6  }
7
8  leerArchivo();
9

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia>node consulta.js
Cambiando informacion de archivo

C:\Users\persistencia>
```

Incluso, podemos mirar el archivo en nuestro editor de código, y ver como se ha cambiado la información contenida.



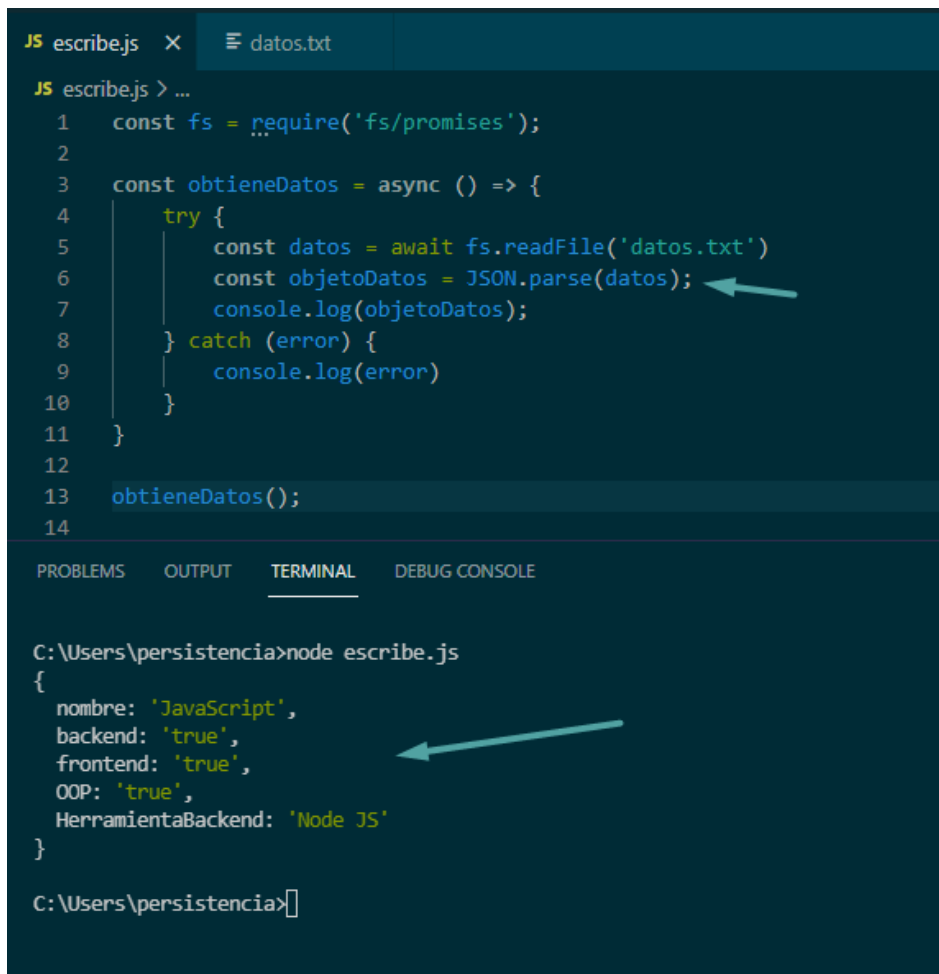
```
datos.txt X
datos.txt
1  Cambiando informacion de archivo
```

Hasta ahora podríamos decir que esto es una persistencia a medias, ya que estamos eliminando el contenido anterior, y reemplazándolo con el nuevo. Generalmente, nuestra data no estará guardada en formato de simple texto, así que haremos pruebas utilizando el formato **json** para guardar los datos en un objeto.

Añadiremos el siguiente juego de datos a nuestro archivo de texto.

```
1 {  
2     "nombre": "JavaScript",  
3     "backend": "true",  
4     "frontend": "true",  
5     "OOP": "true",  
6     "HerramientaBackend": "Node JS"  
7 }
```

Ahora, lo que haremos será modificar nuestro programa `escribe.js` para tomar los datos de tipo **json**, transformarlos a un objeto, y editarlo. Para esto, transformaremos los datos recibidos directamente a formato **json**.



```
JS escribe.js x  datos.txt  
JS escribe.js > ...  
1  const fs = require('fs/promises');  
2  
3  const obtieneDatos = async () => {  
4      try {  
5          const datos = await fs.readFile('datos.txt')  
6          const objetoDatos = JSON.parse(datos);  
7          console.log(objetoDatos);  
8      } catch (error) {  
9          console.log(error)  
10     }  
11 }  
12  
13 obtieneDatos();  
14
```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```
C:\Users\persistencia>node escribe.js  
{  
  nombre: 'JavaScript',  
  backend: 'true',  
  frontend: 'true',  
  OOP: 'true',  
  HerramientaBackend: 'Node JS'  
}  
  
C:\Users\persistencia>
```



Si queremos modificar nuestro objeto para cambiar alguna propiedad, podemos hacer uso del **spread operator**, que permite hacer una copia de nuestro objeto, y modificar la data que queremos.

```
JS escribe.js x datos.txt
JS escribe.js > [0] obtieneDatos > [0] nuevoObjeto
1  const fs = require('fs/promises');
2
3  const obtieneDatos = async () => {
4    try {
5      const datos = await fs.readFile('datos.txt')
6      const objetoDatos = JSON.parse(datos);
7
8      const nuevoObjeto = { ...objetoDatos, nombre: 'python' }
9
10   } catch (error) {
11     console.log(error)
12   }
13 }
14
```

Recuerda que, de la misma forma, también podemos agregar propiedades nuevas a nuestro objeto.

```
JS escribe.js x datos.txt
JS escribe.js > [0] obtieneDatos
1  const fs = require('fs/promises');
2
3  const obtieneDatos = async () => {
4    try {
5      const datos = await fs.readFile('datos.txt')
6      const objetoDatos = JSON.parse(datos);
7
8      const nuevoObjeto = { ...objetoDatos, nombre: 'python', asincrono: true };
9
10   } catch (error) {
11     console.log(error)
12   }
13 }
```

Y ahora, al momento de indicarle a nuestro programa que queremos escribir o modificar el archivo, primero transformaremos nuestro nuevo objeto de vuelta a formato **JSON**. Los argumentos extra usados en el método **JSON.stringify** son solo para darle formato a nuestro **JSON**, y que así el resultado no se encuentre en una sola línea de texto.

```
JS escribe.js > [E] obtieneDatos
1  const fs = require('fs/promises');
2
3  const obtieneDatos = async () => {
4    try {
5      const datos = await fs.readFile('datos.txt')
6      const objetoDatos = JSON.parse(datos);
7
8      const nuevoObjeto = { ...objetoDatos, nombre: 'python', asincrono: true };
9
10     const contenidoArchivo = JSON.stringify(nuevoObjeto, null, 2); ←
11   } catch (error) {
12     console.log(error)
13   }
14 }
```

Cambiaremos el método **escribirArchivo()**, para que reciba como argumento la información a modificar.

```
JS escribe.js x  datos.txt
JS escribe.js > [E] obtieneDatos
1  const fs = require('fs/promises');
2
3  const escribirArchivo = async (contenido) => {
4    try{
5      await fs.writeFile('datos.txt', contenido, 'utf-8');
6      console.log("Los datos han sido cambiados exitosamente");
7    } catch (err) {
8      console.log(err);
9    }
10 }
11
```

Llamamos a la función `escribirArchivo()` dentro de la función `obtieneDatos()`, le pasamos nuestro nuevo objeto en formato `json`, y ejecutamos.

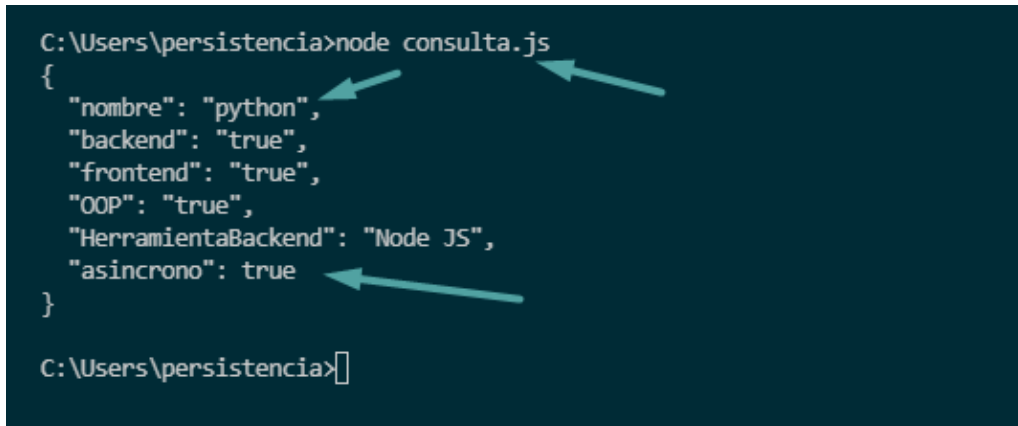


```
JS escribejs x datos.txt
JS escribejs > | obtieneDatos
1  const fs = require('fs/promises');
2
3  const escribirArchivo = async (contenido) => {
4    try{
5      await fs.writeFile('datos.txt', contenido, 'utf-8');
6      console.log("Los datos han sido cambiados exitosamente");
7    } catch (err) {
8      console.log(err);
9    }
10  }
11
12  const obtieneDatos = async () => {
13    try {
14      const datos = await fs.readFile('datos.txt')
15      const objetoDatos = JSON.parse(datos);
16
17      const nuevoObjeto = { ...objetoDatos, nombre: 'python', asincrono: true };
18
19      const contenidoArchivo = JSON.stringify(nuevoObjeto, null, 2);
20
21      escribirArchivo(contenidoArchivo);
22    } catch (error) {
23      console.log(error)
24    }
25  }
26
27
28  obtieneDatos();
29
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\persistencia>node escribe.js
Los datos han sido cambiados exitosamente
C:\Users\persistencia>
```

Nos aseguramos de que nuestro archivo de texto ha sido cambiado exitosamente.



```
C:\Users\persistencia>node consulta.js
{
  "nombre": "python",
  "backend": "true",
  "frontend": "true",
  "OOP": "true",
  "HerramientaBackend": "Node JS",
  "asincrono": true
}
C:\Users\persistencia>
```

Y por último, revisamos nuestro archivo.

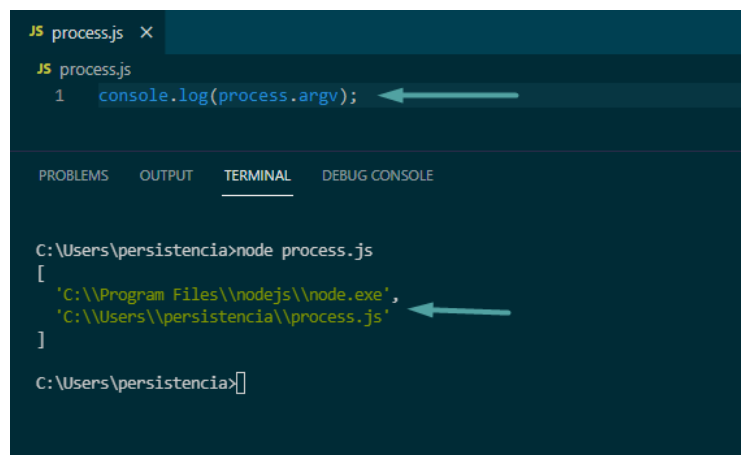


```
datos.txt x
datos.txt
1 {
2   "nombre": "python",
3   "backend": "true",
4   "frontend": "true",
5   "OOP": "true",
6   "HerramientaBackend": "Node JS",
7   "asincrono": true
8 }
```

Por ahora este programa no es tan eficiente, ya que debemos cambiar su contenido para decidir qué información queremos modificar.

Haremos una breve introducción a la ejecución de **node js** con argumentos de entrada.

Cada vez que ejecutamos nuestro programa con **"node archivo.js"**, tenemos acceso a un Array llamado **argv**, que está contenido dentro del objeto process (este es global, y puede accederse desde cualquier parte de tu programa), que por defecto siempre mostrará la ruta desde donde se está ejecutando Node, y la ruta del archivo ejecutado como los primeros dos ítems el Array.



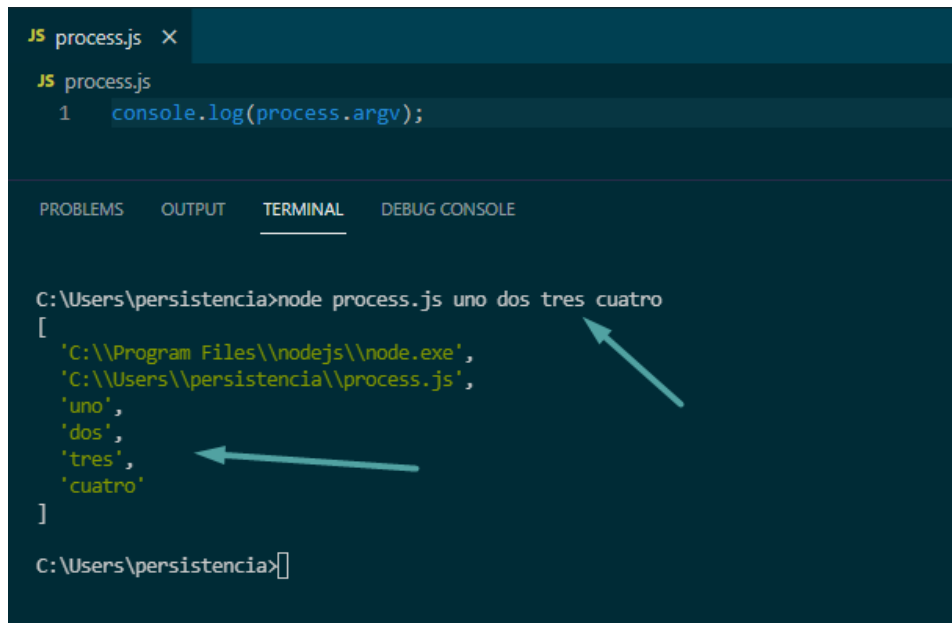
```
JS process.js x
JS process.js
1 console.log(process.argv);

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\persistencia>node process.js
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\persistencia\\process.js'
]

C:\Users\persistencia>
```

¿Qué pasa si agregamos más cadenas de texto en el comando de ejecución de Node?



```
JS process.js X
JS process.js
1 console.log(process.argv);

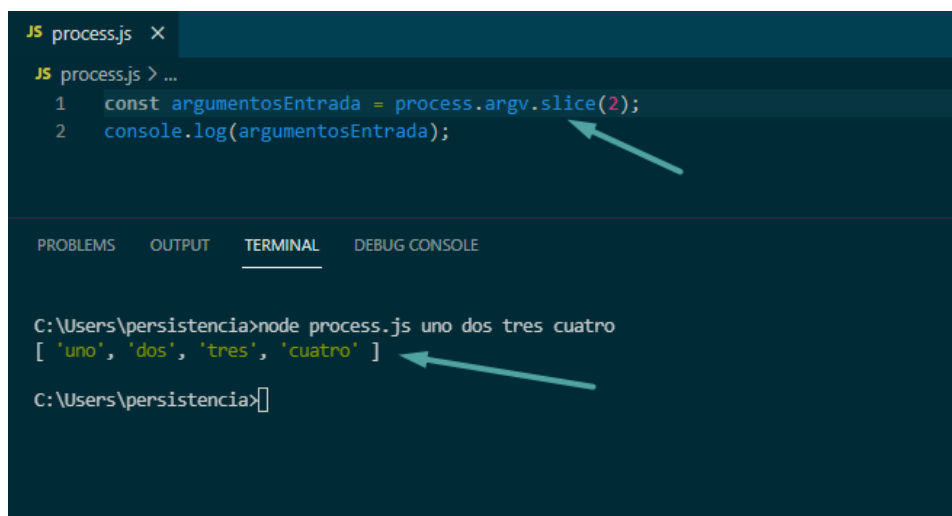
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\persistencia>node process.js uno dos tres cuatro
[
  'C:\\Program Files\\nodejs\\node.exe',
  'C:\\Users\\persistencia\\process.js',
  'uno',
  'dos',
  'tres',
  'cuatro'
]
C:\Users\persistencia>
```

The screenshot shows a VS Code editor with a file named `process.js` containing the line `console.log(process.argv);`. The terminal window below shows the command `node process.js uno dos tres cuatro` being executed. The output is an array of strings: `['C:\\Program Files\\nodejs\\node.exe', 'C:\\Users\\persistencia\\process.js', 'uno', 'dos', 'tres', 'cuatro']`. Two blue arrows point to the third and fourth elements of the array, 'uno' and 'dos', respectively.

Podemos pasar todos los argumentos que queramos desde la línea de comandos, y utilizarlos dentro de nuestro programa.

Empezamos eliminando de entrada los primeros dos ítems del Array, para así poder limpiar la información manteniendo solo los que nos interesan.



```
JS process.js X
JS process.js > ...
1 const argumentosEntrada = process.argv.slice(2);
2 console.log(argumentosEntrada);

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\persistencia>node process.js uno dos tres cuatro
[ 'uno', 'dos', 'tres', 'cuatro' ]
C:\Users\persistencia>
```

The screenshot shows the same VS Code editor with `process.js` updated to `const argumentosEntrada = process.argv.slice(2);` and `console.log(argumentosEntrada);`. The terminal shows the same command `node process.js uno dos tres cuatro`. The output array is `[ 'uno', 'dos', 'tres', 'cuatro' ]`. Two blue arrows point to the second and third elements of the array, 'dos' and 'tres', respectively.

Ahora, para seguir con nuestro programa y modificar nuestro archivo de texto, pasaremos dos argumentos a su ejecución: el primero será el nombre de la propiedad que queremos agregar o modificar; y el segundo será el valor de esa propiedad. Definiremos dos variables para los valores de entrada.

```
JS escribe.js X
JS escribe.js > ...
1  const fs = require('fs/promises');
2
3  const argumentosEntrada = process.argv.slice(2);
4  const propiedad = argumentosEntrada[0];
5  const valor = argumentosEntrada[1];
6
7  const escribirArchivo = async (contenido) => {
8    try{
9      await fs.writeFile('datos.txt', contenido, 'utf-8');
10     console.log("Los datos han sido cambiados exitosamente");
11   } catch (err) {
12     console.log(err);
13   }
14 }
```

Ahora, agregaremos estas variables al momento de definir el nuevo objeto.

```
JS escribe.js X
JS escribe.js > [0] obtieneDatos
1  const fs = require('fs/promises');
2
3  const argumentosEntrada = process.argv.slice(2);
4  const propiedad = argumentosEntrada[0];
5  const valor = argumentosEntrada[1];
6
7  const escribirArchivo = async (contenido) => {
8    try{
9      await fs.writeFile('datos.txt', contenido, 'utf-8');
10     console.log("Los datos han sido cambiados exitosamente");
11   } catch (err) {
12     console.log(err);
13   }
14 }
15
16 const obtieneDatos = async () => {
17   try {
18     const datos = await fs.readFile('datos.txt')
19     const objetoDatos = JSON.parse(datos);
20
21     const nuevoObjeto = { ...objetoDatos, [propiedad]: valor };
22
23     const contenidoArchivo = JSON.stringify(nuevoObjeto, null, 2);
24
25     escribirArchivo(contenidoArchivo);
26
27   } catch (error) {
```

Y ejecutamos nuestro programa `escribe.js`, con dos argumentos de entrada.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia>node escribe.js nuevaPropiedad nuevoValor
Los datos han sido cambiados exitosamente

C:\Users\persistencia>
```

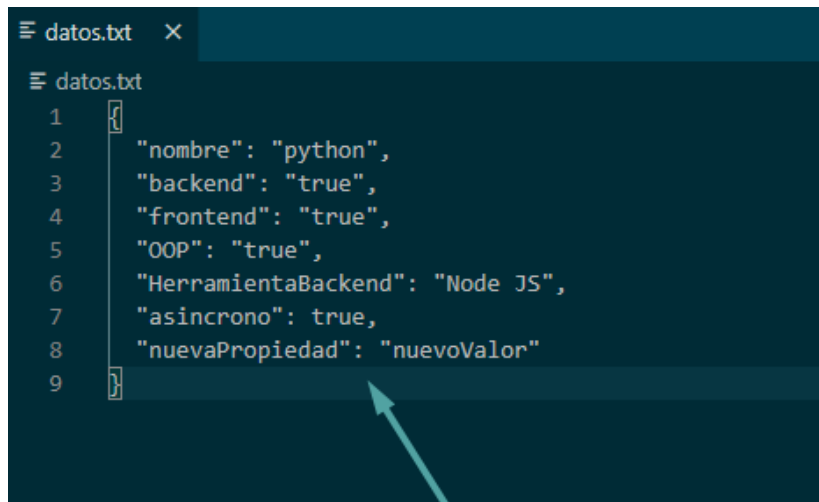
Para revisar el resultado, podemos ejecutar nuestro programa `consulta.js`.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia>node consulta.js
{
  "nombre": "python",
  "backend": "true",
  "frontend": "true",
  "OOP": "true",
  "HerramientaBackend": "Node JS",
  "asincrono": true,
  "nuevaPropiedad": "nuevoValor"
}

C:\Users\persistencia>
```

O bien, revisar nuestro archivo y comprobar que las nuevas propiedades si existen.



```
datos.txt x
datos.txt
1 {
2   "nombre": "python",
3   "backend": "true",
4   "frontend": "true",
5   "OOP": "true",
6   "HerramientaBackend": "Node JS",
7   "asincrono": true,
8   "nuevaPropiedad": "nuevoValor"
9 }
```

## MOTOR DE PLANTILLAS

Una de las características que permite *Express* al crear un servidor con este framework, es la definición de una estructura en forma de documento de salida, mediante el uso de plantillas que puedan generar una vista. Las plantillas permiten utilizar variables que serán cambiadas por datos que se llenan al ser construidas, y comúnmente se utiliza el lenguaje de marcado HTML. Esta herramienta de generación de la estructura y el cambio de variables a datos, se le conoce como: **motor de plantilla** o **motor de vistas**; y, además, *Express* es compatible con diferentes motores, algunos de los más conocidos son: Pug, EJS, Handlebars, entre otros.

Para instalar un motor de plantilla en tu proyecto, debes hacerlo con el comando:

```
npm install nombre_motor_plantilla -save
```

Para que Express pueda representar archivos de plantilla, deben establecerse los siguientes valores de aplicación:

- Views: la carpeta donde se encontrarán los archivos de plantilla.  
Ejemplo: `app.set('views', '/views')`
- View Engine: el motor de plantilla que se utilizará.  
Ejemplo: `app.set('view engine', 'pug')`
- Cada motor de plantilla tiene su propia extensión para creación de las vistas.  
Ejemplo: `index.pug` o `index.ejs`