

TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Contexto de uso Para Node.js.
- JavaScript del lado del servidor.
- Descripción de Node.js, sus características y usos.
- Código asíncrono.
- Loop de eventos.
- Instalación y primer programa.

El concepto de aplicaciones web generalmente se encuentra dividido por lo que llamamos Front End y Backend. A grandes rasgos, el Front End es todo aquello que vemos en nuestro navegador. Mientras que el Back End es todo aquello que se encuentra en el servidor. Siento así, el Front End hace peticiones al Back End, y éste las procesa y devuelve respuestas.

Usualmente, los lenguajes más utilizados para construir el Back End de una aplicación eran Java, PHP, C+, entre otros. Por su parte, JavaScript era un lenguaje que solo estaba diseñado para ser usado en el Front End de una aplicación, haciendo más interactiva una página web.

Con la llegada de **Node.js**, se abre una nueva posibilidad: **el uso de JavaScript del lado del servidor**. Ahora, ya es posible construir aplicaciones Front To Back usando un solo lenguaje de programación, y ahorrando así un poco de tiempo en aprender uno distinto para el Back End.

Actualmente, Node se ha vuelto una herramienta tan poderosa, que incluso empresas como Amazon, Netflix, eBay, Reddit, LinkedIn, Tumblr, y PayPal han decidido hacer uso ella, obteniendo mejoras en sus tiempos de respuesta y rendimiento.

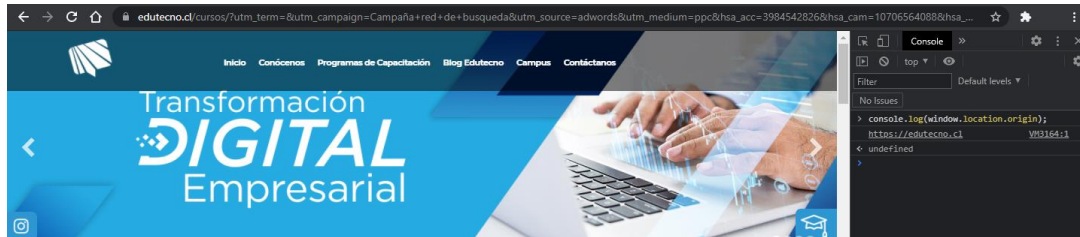
Aprender **JavaScript** y construir aplicaciones usando **Node**, es una aptitud muy requerida al momento de buscar trabajo. Por lo tanto, siempre se tendrá una ventaja competitiva si **Node** está dentro de tu lista de habilidades.

Cuando buscamos en la web una descripción de **Node.js**, la definición que siempre encontrarás, y que se encuentra en la página oficial de **Node** (www.nodejs.org), es la siguiente:

“Node.js es un entorno de ejecución (runtime en inglés) para JavaScript, construido sobre el motor de JavaScript V8 de Chrome”

¿Entorno de ejecución? ¿Motor V8 de Chrome?, ¿a qué nos referimos?

Cuando escribes tu código en un archivo de texto plano, éste debe ser interpretado y ejecutado. Antes, el lenguaje de programación **JavaScript** solo podía ser ejecutado en los navegadores web, por lo tanto, es aquel quien toma tu código, lo lee y lo ejecuta.



La llegada de **Node** permite levantar un entorno de ejecución para tomar tu código, leerlo, y ejecutarlo, ya no solamente en el navegador, sino que también fuera de él.

```
JS index.js  X
JS index.js > ...
1  const animales = {
2    perro: "Doggo",
3    gato: "Catto",
4    Serpiente: "Ssssss"
5  };
6
7  const numero1 = 323;
8
9  const numero2 = 121;
10
11 const suma = numero1 + numero2;
12
13 const multiplicacion = numero1 * numero2;
14
15 const perro = animales.perro;
16
17 console.log(suma);
18 console.log(multiplicacion);
19 console.log(perro);

PROBLEMS  OUTPUT  TERMINAL
$ node index.js
444
39083
Doggo
```

Usemos una analogía para entender mejor el concepto de **runtime**: podemos considerar que el código contenido en tus archivos son los ingredientes para preparar el almuerzo. No podemos llegar y lanzarlos todos a la cocina, esperando que funcione, pues necesitamos de, por lo menos, un sartén o una olla que los contenga para que éstos se puedan cocinar. Entonces, nuestro sartén (runtime) será quien nos provee de un lugar para poder cocinar (ejecutar) nuestros ingredientes (código).

Por otra parte, el **motor V8** de Google es un optimizador para la ejecución de código JavaScript, que le permite al navegador una gran eficiencia al momento de ejecutar **JavaScript**. **Node** utiliza este motor para obtener una alta performance en las aplicaciones construidas sobre este entorno de ejecución. Ahora, sin revisar más detalles, y continuando con la analogía anterior, podemos pensar el motor V8 como el aceite que nos permitiría cocinar eficientemente todos los ingredientes de nuestro sartén.


Una de las particularidades de **Node**, y que le permite ser tan eficiente, es su naturaleza asíncrona. Cada vez que se ejecuta una tarea, ya se espera una respuesta. Tareas como peticiones HTTP, acceder a una base de datos, o al sistema de archivos, toman cierta cantidad de tiempo en ejecutarse; mientras su respuesta no llega, **Node** liberará el hilo de ejecución para el siguiente código. Y una vez se obtiene la respuesta de la primera tarea, envía la señal y retoma la ejecución con el resultado de ésta.

Si parece muy confuso, consideremos lo siguiente:

Vamos a una cafetería y ordenamos nuestro café favorito. Éste primero debe ser preparado, y luego es llevado a nuestra mesa, lo cual puede tomar un tiempo determinado. Durante ese lapso, podemos mirar nuestro celular, sacar fotos, o llamar a un amigo, y así aprovechar el tiempo mientras llega el pedido. Cuando el café está listo, dejamos lo que estábamos haciendo, y lo bebemos. Imagina lo poco eficiente que sería quedarnos detenidos, sin hacer nada mientras llega nuestro café. Esa es la idea detrás de la eficiencia de **Node**.

La forma más común de implementar este concepto de ser notificado cuando la respuesta ha sido recibida, es con las llamadas **funciones de callback**.

La mayoría de los métodos de librerías con los que te encontrarás hacen uso de ellas. Una función de **callback** no es más que una función que se pasa como el argumento de otra, y que contiene el código que se ejecutará una vez que llegue la respuesta, o se emita el evento esperado.



```
FuncionCallback.js

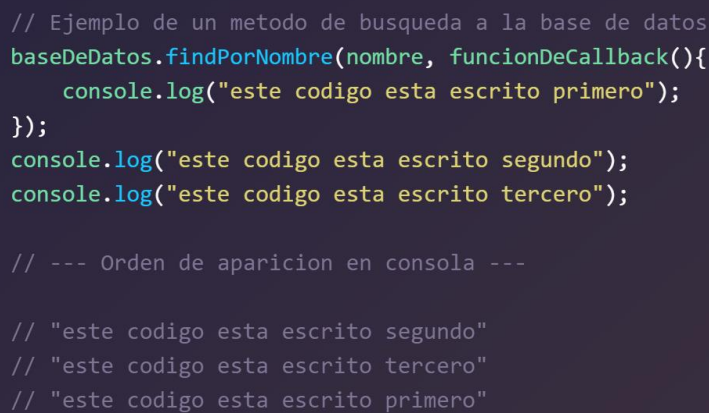
//Ejemplo de sintaxis comun para librerias con que usan funcion de callback
libreria.metodo(argumento, funcionDeCallback() {
  const respuesta = ("me ejecutare solo cuando la respuesta este lista");
  console.log(respuesta);
});
```

Node JS maneja la ejecución de código dentro de un mismo hilo de ejecución, esto significa que **Node** puede hacer solo una cosa a la vez. Cada vez que existe una operación asíncrona, como las descritas anteriormente, pone a la espera la respuesta de esa tarea en un hilo secundario, mientras el hilo primario continúa ejecutándose. Una vez que la operación asíncrona devuelve una respuesta, se genera un evento que es puesto de vuelta en el hilo principal para ejecutar el código de tu **callback**. De ahí también proviene la definición de la arquitectura de **Node** basada en eventos.

Para finalizar con esta introducción, existe otro concepto importante para tener en cuenta. Este hilo de ejecución, mencionado anteriormente, es controlado por el **loop** de eventos de **Node**: es el encargado de orquestar todas las ejecuciones de código, determinando una lista con el orden de ejecución.

Como hemos revisado antes, las funciones de **callback** son enviadas a un hilo secundario, y solo son ejecutadas cuando se dispara el evento con la respuesta de la petición. Por lo tanto, solo cuando el evento ocurra, se añadirá la función a la lista del **loop** de eventos que controla el hilo principal de ejecución.

Este es un concepto muy importante para tener en cuenta, ya que puede ocasionar problemas que no son tan intuitivos a primera vista. Observemos el siguiente ejemplo de código:



```
// Ejemplo de un metodo de busqueda a la base de datos
baseDeDatos.findPorNombre(nombre, funcionDeCallback){
  console.log("este codigo esta escrito primero");
});
console.log("este codigo esta escrito segundo");
console.log("este codigo esta escrito tercero");

// --- Orden de aparicion en consola ---

// "este codigo esta escrito segundo"
// "este codigo esta escrito tercero"
// "este codigo esta escrito primero"
```

¿No te parece un resultado un poco inesperado?

Un método que hace una consulta a la base de datos tomará un tiempo determinado en recibir una respuesta; debido a la naturaleza asíncrona de **Node**, éste continuará con la ejecución del código que viene después, hasta que reciba la respuesta de la base de datos.

Debes tener en cuenta que puede ocurrir que tu código no sea ejecutado en el orden que tú lo escribes, esto puede sonar confuso al principio, pero es una de las fortalezas de **Node**, ya que le permite no bloquear el hilo principal con tareas que toman más tiempo de lo normal, continuando con la ejecución del código, y así “aprovechar el tiempo de espera” que puede tomar una.

Entonces, luego de esta descripción... **¿Qué podemos hacer con Node JS?**

Node js puede ser un servidor web, una aplicación web completa de inicio a fin, podemos conectar con bases de datos, crear programas que interactúen con el sistema de archivos de tu

computador, crear aplicaciones de escritorio, crear múltiples instancias de **Node**, y construir una arquitectura basada en microservicios.

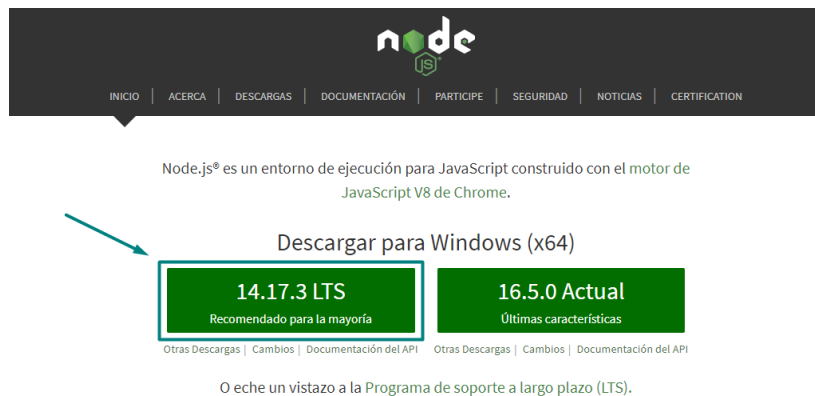
Con todos los conceptos revisados en esta unidad, tendrás una idea del poder de **Node**, para comenzar a crear tus programas y resolver situaciones de la vida real.

INSTALACIÓN DE NODE.JS

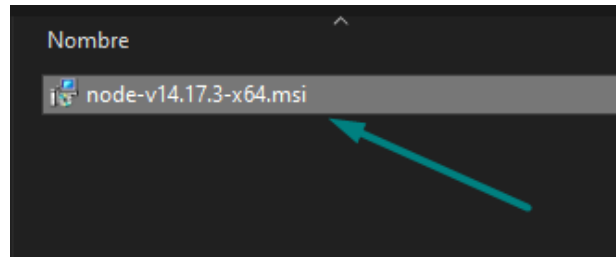
Ahora que ya conoces las características generales de **Node**, comenzaremos con la instalación, y luego escribiremos nuestras primeras líneas de código para ser ejecutadas en él.

En primer lugar, debemos descargar **Node.js** de su página oficial, en la siguiente URL: <https://nodejs.org/>.

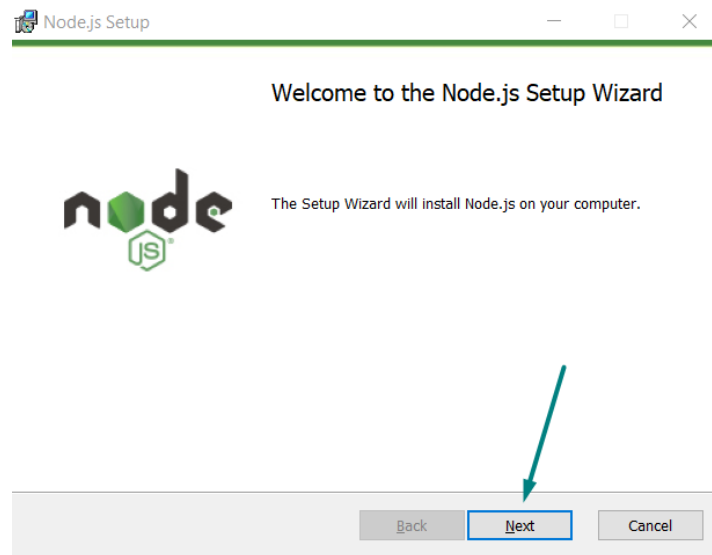
Una vez que entres, lo primero que deberías ver es el botón de descarga. Haz clic en la **versión LTS** que diga “Recomendado para la mayoría”. Esta significa **Long Term Support**, que se traduce como “soporte de largo plazo”, y significa que es la versión que será mantenida durante un periodo extendido de tiempo.



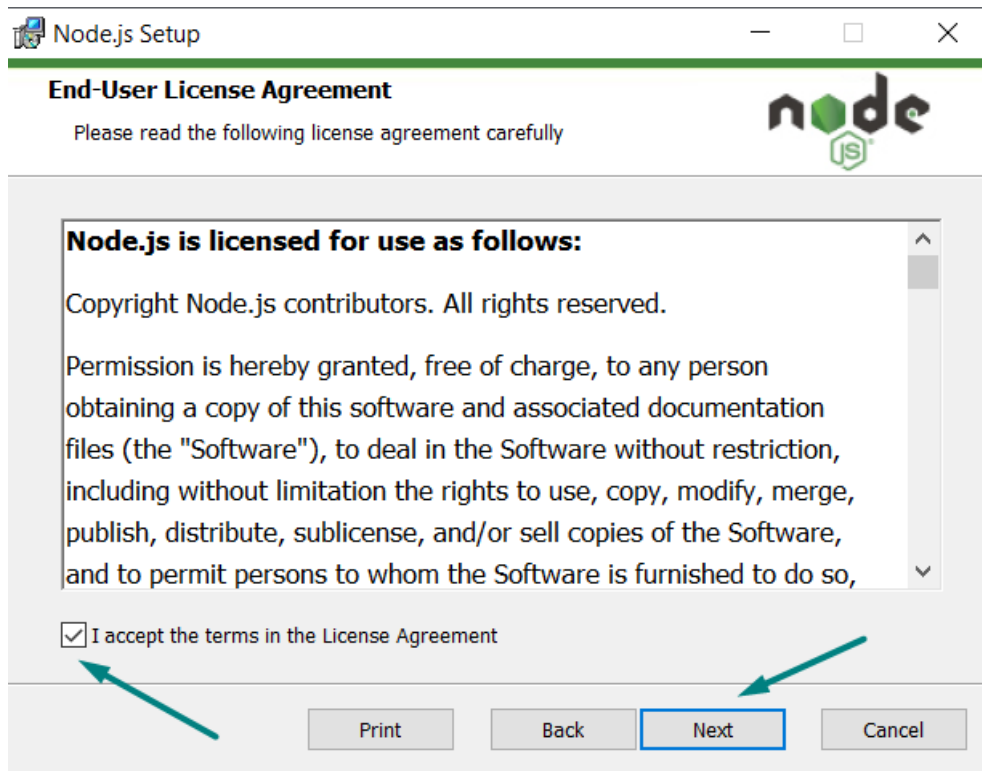
Una vez descargado, hacemos doble clic en el archivo ejecutable.



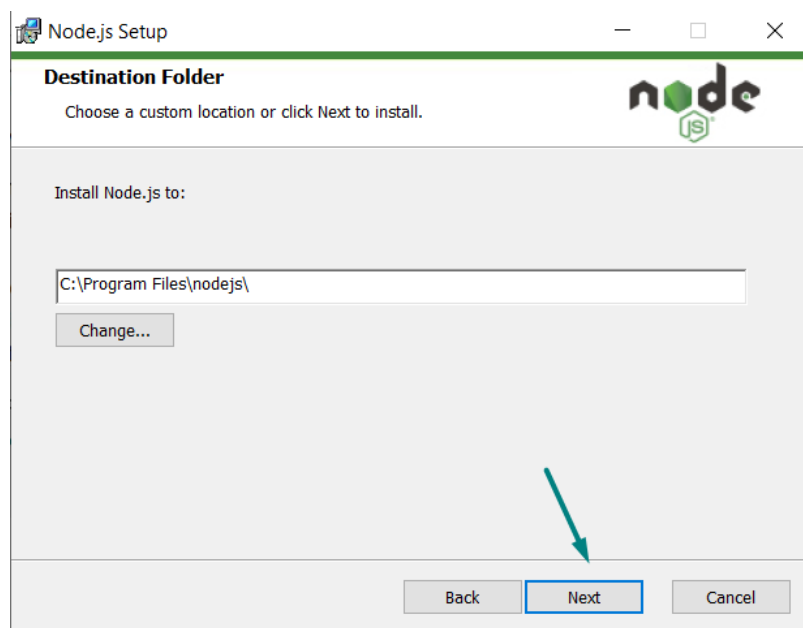
Esperamos unos segundos, y hacemos clic en “Siguiente”.



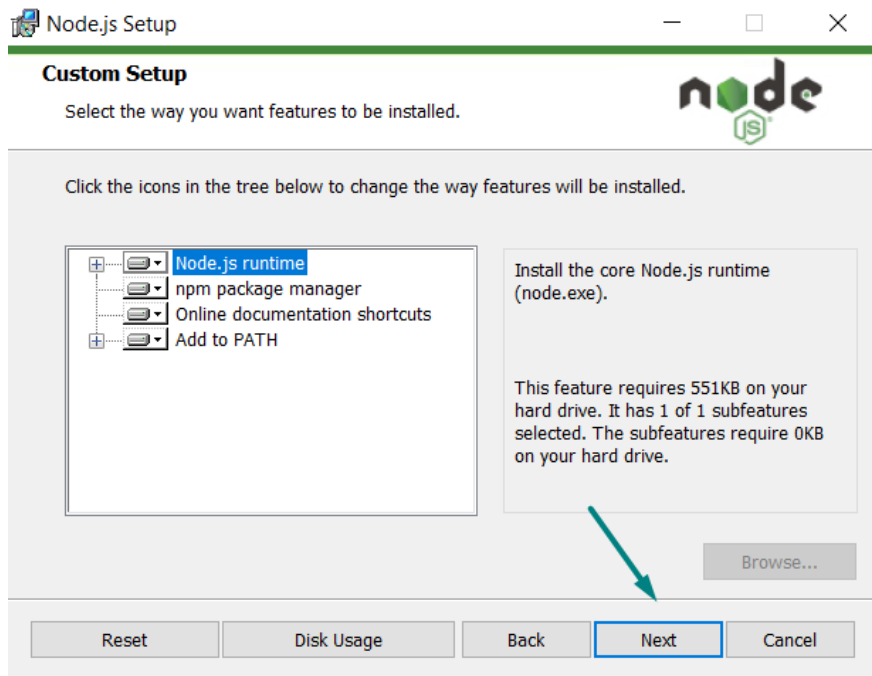
Leemos los términos y condiciones, hacemos clic en la casilla, y luego “Siguiente”.



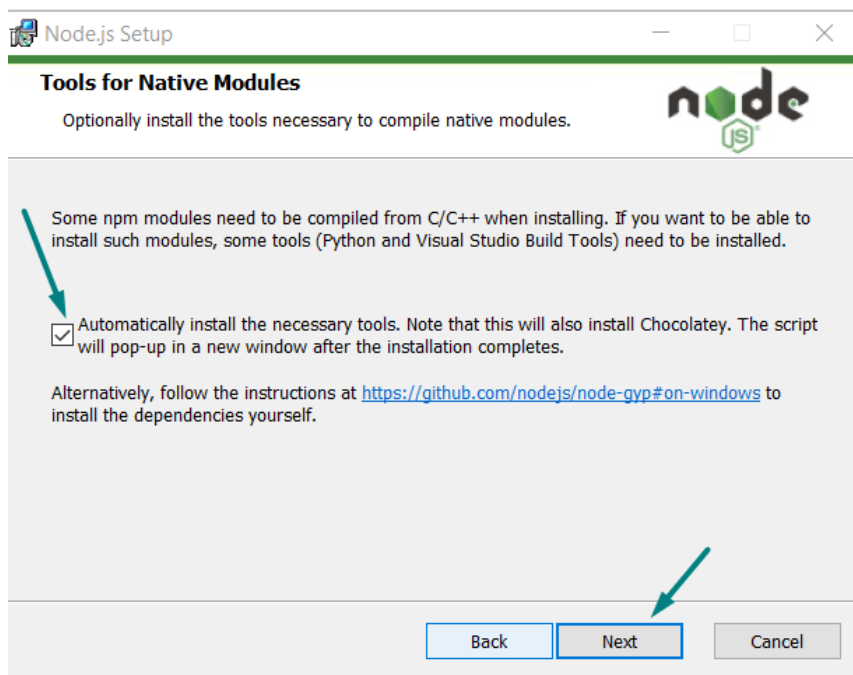
Dejamos la carpeta por defecto, y hacemos clic en "Siguiente".



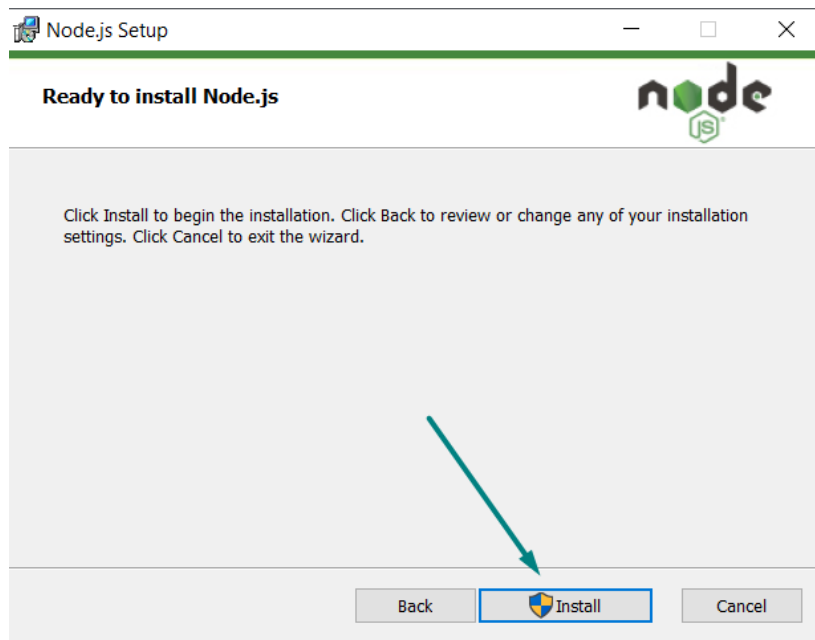
Nuevamente dejamos las opciones por defecto, y hacemos clic en "Siguiente".



Seleccionamos la casilla para instalar las herramientas necesarias, y hacemos clic en “Siguiete”.



Ahora, hacemos clic en “Instalar”, y luego de esto aparecerá un mensaje solicitando permisos de administrador para continuar con la instalación, debes hacer clic en “Aceptar”.



Para verificar que la instalación ha sido exitosa, abriremos la línea de comandos de nuestro sistema operativo, y escribiremos el comando **“node -v”** sin comillas; al presionar enter, tu terminal debería mostrar la versión de **node**.

```
C:\Users>node -v
v14.16.0
C:\Users>
```



Si no logras ver la versión de **node js**, dirígete a la sección de HINTS para revisar las soluciones a los problemas más comunes al momento de la instalación.

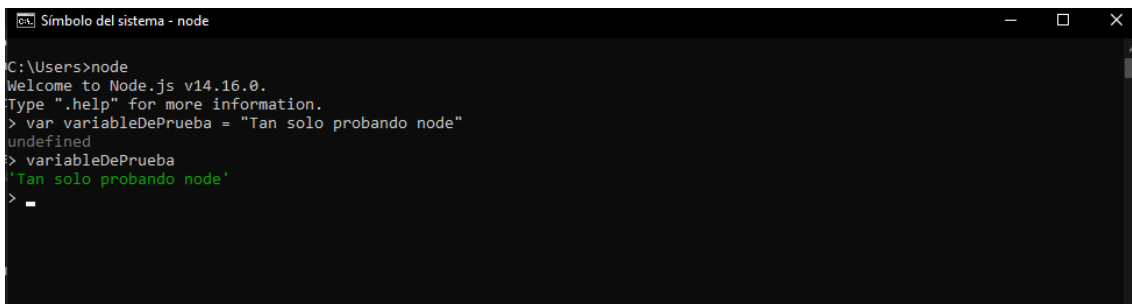
Realicemos el siguiente ejercicio. En tu terminal o línea de comandos, escribe Node, y presiona la tecla enter.



```
Símbolo del sistema - node
C:\Users>node
Welcome to Node.js v14.16.0.
Type ".help" for more information.
>
```

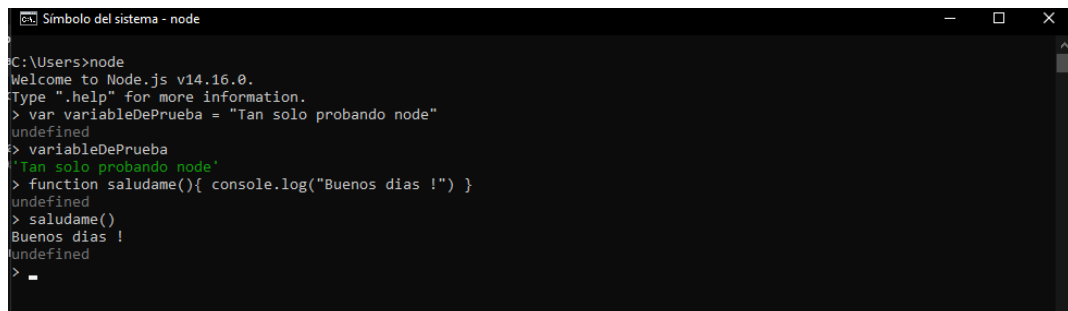
Al hacerlo, **node** se inicia y espera instrucciones. De aquí en adelante, podemos escribir todo el código **JavaScript** que necesitemos.

Podemos definir una variable.



```
Símbolo del sistema - node
C:\Users>node
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> var variableDePrueba = "Tan solo probando node"
undefined
> variableDePrueba
'Tan solo probando node'
>
```

Podemos definir una función, e invocarla.



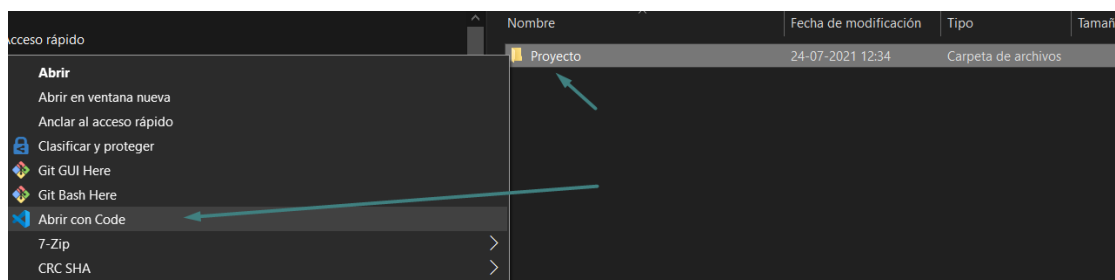
```
Símbolo del sistema - node
C:\Users>node
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> var variableDePrueba = "Tan solo probando node"
undefined
> variableDePrueba
'Tan solo probando node'
> function saludame(){ console.log("Buenos días !") }
undefined
> saludame()
Buenos días !
undefined
>
```

Realizar operaciones aritméticas, y cualquier sintaxis válida de **JavaScript**.

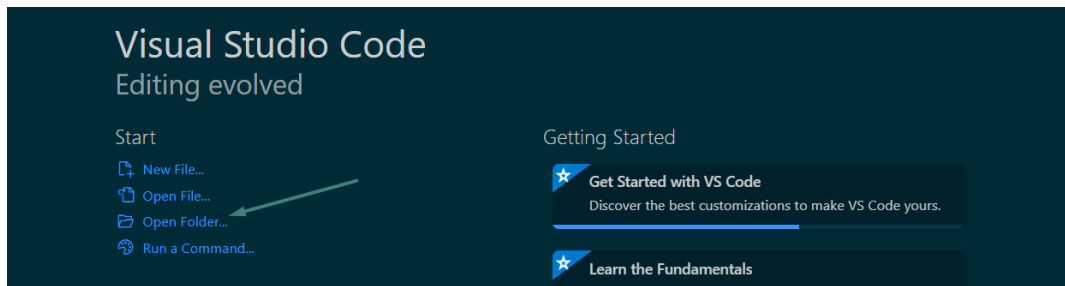
```
Simbolo del sistema - node
Type ".help" for more information.
> var variableDePrueba = "Tan solo probando node"
undefined
> variableDePrueba
'Tan solo probando node'
> function saludame(){ console.log("Buenos días !") }
undefined
> saludame()
Buenos días !
undefined
> var numero1 = 234
undefined
> var numero2 = 567
undefined
> numero1 + numero2
801
> var array = ["hola", "hi", "ciao", "hallo"]
undefined
> array.forEach(item => console.log(item))
hola
hi
ciao
hallo
undefined
```

Es importante destacar que todo el código que hemos escrito hasta ahora en la terminal no será guardado, y una vez que cierres la instancia de **node**, éste se perderá con ella. Para salir de dicha instancia, solo debes presionar las teclas **ctrl + c**. Esto nos llevará al siguiente punto, que es crear nuestro primer programa de Node.

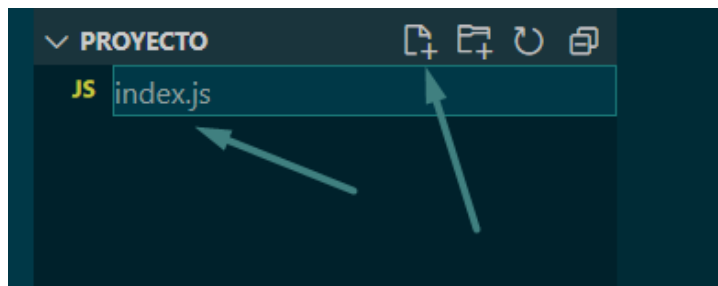
Crearemos una nueva carpeta llamada proyecto, haremos clic derecho, y en las opciones seleccionamos "Abrir con Code", con esto, Visual Studio entenderá que esta es nuestra carpeta de trabajo.



También puedes abrir Visual Studio Code, y hacer clic en la opción Open Folder o Abrir Carpeta, y luego buscar la carpeta recién creada en los archivos de tu sistema operativo.



Una vez abierto nuestro editor de código, crearemos un nuevo archivo llamado index.js.



Ya que estamos más avanzados dentro del curso, ¿qué tal si probamos nuestro primer programa con algo distinto, y no solo con el conocido “Hola mundo”?

Crearemos una función que acepte un argumento de tipo booleano, y dependiendo de su valor, imprimiremos por pantalla los textos “Hola mundo” o “Hasta pronto mundo”, dependiendo sea el caso. El valor de la variable booleana, a su vez, dependerá de si la suma de dos enteros es mayor a un número determinado.

CREANDO NUESTRO PRIMER PROGRAMA

Primero definimos nuestras variables, en este caso crearemos cuatro:

verdaderoFalso: esta variable será definida con el valor true o false, dependiendo del resultado de nuestro programa.

entero1-entero2: contienen los valores de los enteros que queremos sumar.

valorLimite: contienen el valor límite con el cual evaluaremos si la respuesta de nuestro programa es verdadera o falsa.

```
JS index.js X
JS index.js > ...
1 // Definicion de variables
2 var verdaderoFalso;
3
4 var entero1 = 234;
5
6 var entero2 = 456;
7
8 var valorlimite = 100;
```

Luego, creamos nuestra función.

```
// Definicion de funcion
function saludaDespide(valorVerdaderoFalso) {
  // El bloque if evalua valores verdaderos o falso, ya que nuestra variable sera de tipo booleana,
  // no es necesario decir if(valorVerdadero = true)
  if(valorVerdaderoFalso) {
    console.log("Hola Mundo");
  } else {
    console.log("Hasta pronto mundo");
  }
}
```

Como podemos observar, esta es una función muy simple que tiene un parámetro (**valorVerdaderoFalso**), y usa el valor de éste para mostrar por consola el mensaje "Hola Mundo" o "Hasta pronto mundo", mediante un bloque **if**. En este caso, debido a que sabemos que el valor del parámetro será de tipo verdadero o falso (**booleano**), podemos insertar el parámetro directamente dentro de la evaluación del bloque **if**, sin necesidad de hacer una comparación de tipo, igual que (**valorVerdaderoFalso = true**).

Ahora que ya tenemos nuestra función, crearemos el bloque de código que definirá el valor de la variable **verdaderoFalso**. En el que se muestra a continuación, haremos uso de otro bloque **if**, el cual evaluará si la suma de los dos enteros es mayor a **valorLimite**. Además, se puede observar una sintaxis resumida para definir el valor de la variable **verdaderoFalso**. Todo esto para que vayas conociendo y entendiendo las posibilidades que ofrece **JavaScript**.

```
/*  
  Bloque para definir nuestra variable verdaderoFalso  
  Version larga  
*/  
if( (entero1 + entero2) > valorlimite){  
  verdaderoFalso = true;  
} else {  
  verdaderoFalso = false;  
}  
  
/*  
  Las variables en javascript tambien pueden contener evaluaciones condicionales,  
  esto significa que el valor de la variable sera el resultado de la evaluacion logica  
  Version resumida  
*/  
verdaderoFalso = (entero1 + entero2) > valorlimite;
```

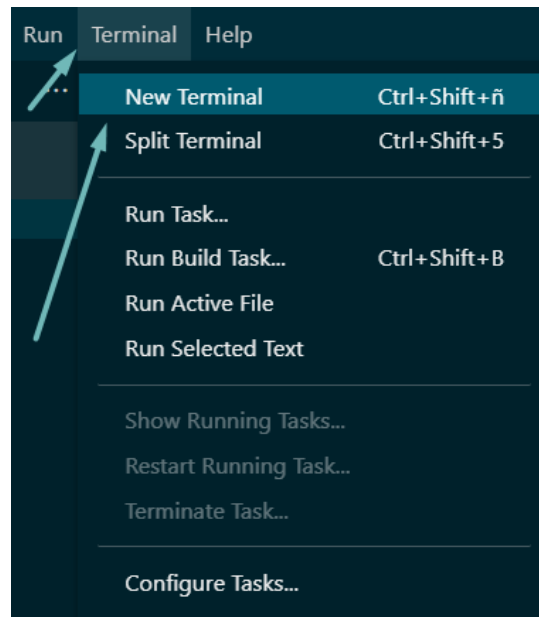
Ahora, ya solo nos queda invocar nuestra función pasando la variable **verdaderoFalso** como argumento.

```
// Invocacion de funcion  
saludaDespide(verdaderoFalso);
```

Si recuerdas lo comentado en el CUE anterior, ahora que tenemos nuestro código escrito, éste se encuentra en nuestro computador y no es más que un archivo de texto plano, es decir, son solo nuestros ingredientes, pero sin un sartén.

Veamos entonces cómo hacer que **node** ejecute nuestro programa.

Usaremos la terminal integrada de Visual Studio Code, en la barra superior hacemos clic en “Terminal”, y luego en “Open new terminal” o “Abrir nueva terminal”.

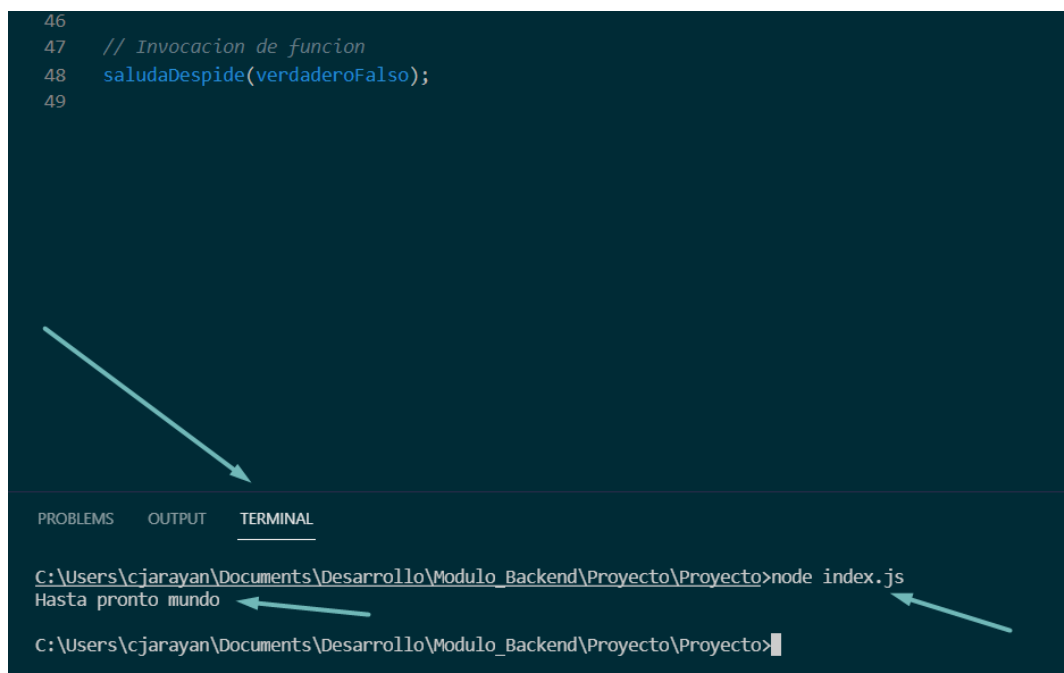


Veremos como aparece nuestro terminal en la parte inferior. Ahora, tan solo basta con escribir en nuestro terminal: Node, y el nombre del archivo, es decir, **“node index.js”** sin comillas. Este comando ejecutará nuestro programa y mostrará por consola el resultado.

```
46
47 // Invocacion de funcion
48 saludaDespide(verdaderoFalso);
49
```

PROBLEMS OUTPUT **TERMINAL**

```
C:\Users\cjarayan\Documents\Desarrollo\Modulo_Backend\Proyecto\Proyecto>node index.js
Hasta pronto mundo
C:\Users\cjarayan\Documents\Desarrollo\Modulo_Backend\Proyecto\Proyecto>
```



A screenshot of the VS Code interface showing the terminal at the bottom. The terminal has tabs for 'PROBLEMS', 'OUTPUT', and 'TERMINAL'. The 'TERMINAL' tab is active, showing the command prompt 'C:\Users\cjarayan\Documents\Desarrollo\Modulo_Backend\Proyecto\Proyecto>' and the command 'node index.js' being executed. The output 'Hasta pronto mundo' is displayed. A green arrow points from the code editor to the terminal, and another green arrow points to the output text.



Podemos ejecutar nuestro archivo cuantas veces queramos. Prueba cambiando los valores de las variables, para alternar el resultado de tu programa (recuerda guardar tu archivo antes de volver a ejecutarlo por consola, pronto conoceremos herramientas que ayudarán con este tipo de tareas repetitivas, tales como guardar un archivo y volver a ejecutarlo manualmente).

¡Felicitaciones! Has creado tu primer Hola Mundo utilizando **Node.js**.