

TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- Tipos de errores en la programación.
- Concepto Debugging.
- Debugging con VS Code.
- Introducción al testing.
- Códigos HTTP.

TIPOS DE ERRORES

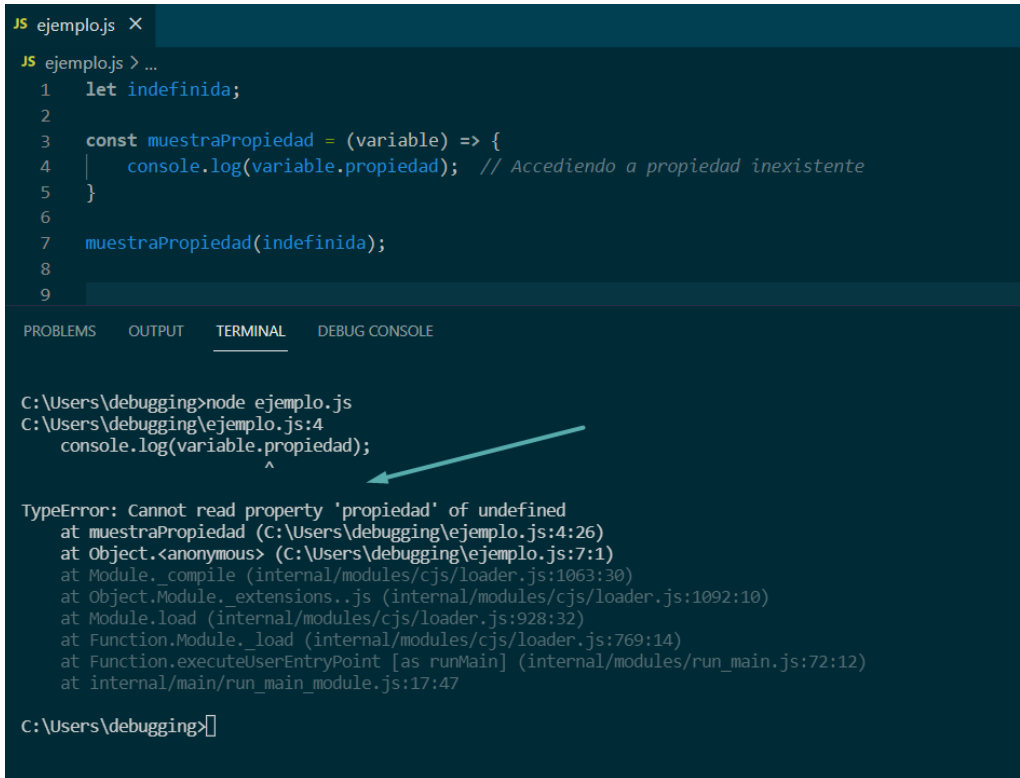
Todos los lenguajes de programación tienen un juego de reglas, las cuales establecen como las palabras y sentencias deben ser estructuradas y escritas. Éstas se conocen generalmente como la sintaxis de un lenguaje de programación; y dictarán la forma de declarar variables, funciones, objetos, entre otros; y con qué palabras claves o palabras reservadas se hará.

Cuando hablamos de un error de sintaxis, nos referimos a uno en el código fuente del programa. Debido a que el lenguaje debe seguir una sintaxis estricta para poder ser compilado, cualquier aspecto del código que no esté escrito conforme a la que fue definida, producirá un error de sintaxis. Éstos generalmente son pequeños errores “gramaticales”, a veces limitado a un solo carácter. Por ejemplo: basta con que solo falte una la llave que cierra una función, para que nuestro programa no sea capaz de compilar.

```
1
2  const devuelveHola = () => {
3    console.log("Hola !")
4  }
```

También existen errores de tiempo de ejecución o **“runtime errors”**, éstos no son detectados al compilar, y son dinámicos, es decir, que ocurrirán a medida que nuestro programa se esté ejecutando. Son un poco más inesperados que los errores de sintaxis, y solo podremos darnos cuenta una vez que nuestro programa llegue a ejecutar la porción de código que producirá el

error. Por ejemplo: existe la posibilidad de que una variable se encuentre indefinida, al momento de ejecutar una función que la reciba como argumento. Si tratamos de acceder a una de sus propiedades, entonces nuestro programa terminará mostrándonos el error.



```
JS ejemplo.js X
JS ejemplo.js > ...
1  let indefinida;
2
3  const muestraPropiedad = (variable) => {
4    |   console.log(variable.propiedad); // Accediendo a propiedad inexistente
5    | }
6
7  muestraPropiedad(indefinida);
8
9

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\debugging>node ejemplo.js
C:\Users\debugging\ejemplo.js:4
  console.log(variable.propiedad);
                        ^
TypeError: Cannot read property 'propiedad' of undefined
    at muestraPropiedad (C:\Users\debugging\ejemplo.js:4:26)
    at Object.<anonymous> (C:\Users\debugging\ejemplo.js:7:1)
    at Module._compile (internal/modules/cjs/loader.js:1063:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
    at Module.load (internal/modules/cjs/loader.js:928:32)
    at Function.Module._load (internal/modules/cjs/loader.js:769:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47

C:\Users\debugging>
```

Por último, podemos hablar de los errores de lógica. Éstos ocurren dentro del código fuente de un programa, y ocasionarán un resultado incorrecto o un comportamiento inesperado. El programa corre correctamente, y no parecieran haber errores de sintaxis, pero aun así tenemos un resultado incorrecto. Son introducidos por el programador, y se deben al mal planteamiento de un problema, o alguna implementación deficiente de algún bloque de código.

Estos errores suelen ser los más difíciles de encontrar, ya que a diferencia de los de sintaxis o de tiempo de ejecución, no estamos recibiendo ningún tipo de alerta o información respecto a donde y como está ocurriendo este error de lógica.

TÉCNICAS DE DEBUGGING

La depuración de un programa o código puede tener distintos enfoques, dependiendo de la necesidad. Anteriormente, ya hemos hablado de la separación del código para utilizar un enfoque más modular, y una de las razones para usarlo es que nos permite buscar errores de manera más simple, ocupándonos de partes específicas del código, y no tanto del programa o código completo. Por lo tanto, una de las primeras técnicas de **debugging**, es ir ejecutando paso a paso los elementos que conforman tu código.

Es recomendable examinar el contenido de tus variables y objetos con el ya conocido **"console.log()"**, en los distintos puntos de tu programa, y ver si es que éstos tienen los valores esperados.

Es importante conocer el orden de ejecución de tu programa; si éste tienes muchas piezas que se van moviendo, debes tener claro cómo interactúan unas con otras. Para ahorrarte tiempo en el futuro, puedes documentar esta ejecución, ya sea mediante diagramas de flujo, o comentando tu código.

DEBUGGING CON VISUAL STUDIO CODE

No es incorrecto usar **console.log()** para ver por consola el contenido de variables y objetos, pero no siempre es lo óptimo, considerando que tenemos otras herramientas que pueden ser bastante más eficientes. Visual Studio Code ofrece una herramienta integrada para realizar depuración de programas.

La herramienta de **debugging** de Visual Studio Code, permite correr nuestro programa hasta determinados puntos, o hacerlo línea por línea, y obtener información respecto a variables y objetos en el momento de la línea que se está ejecutando.

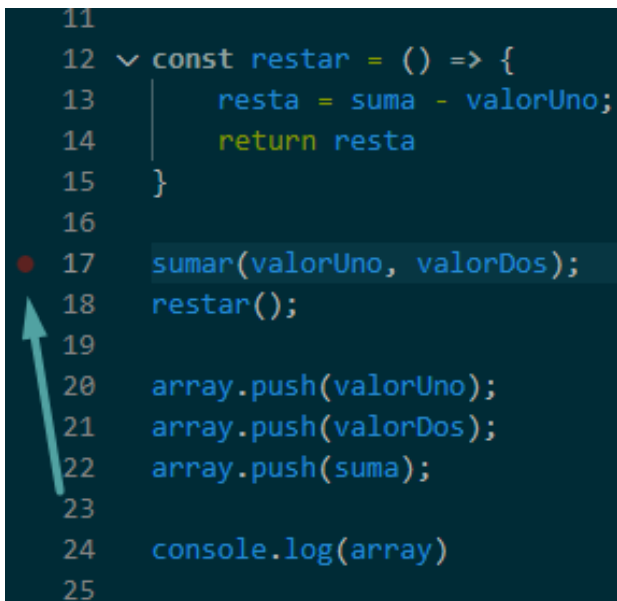
Crearemos un archivo `index.js`, y copiaremos el siguiente código.

```
1 const valorUno = 1;
2 const valorDos = 5;
3 let array = [];
4 let suma;
5 let resta;
6
7 const sumar = (numeroUno, numeroDos) => {
8     suma = numeroUno + numeroDos;
9     return suma;
10 }
11 const restar = () => {
```

```
12     resta = suma - valorUno;
13     return resta
14 }
15
16 sumar(valorUno, valorDos);
17 restar();
18 array.push(valorUno);
19 array.push(valorDos);
20 array.push(suma);
21
22 console.log(array)
```

Dentro de VS Code agregaremos un **breakpoint**. Este es un punto en el código que marca la detención momentánea del programa, y se encuentra al lado izquierdo del número de línea de tu código; debes hacer clic encima para marcarlo.

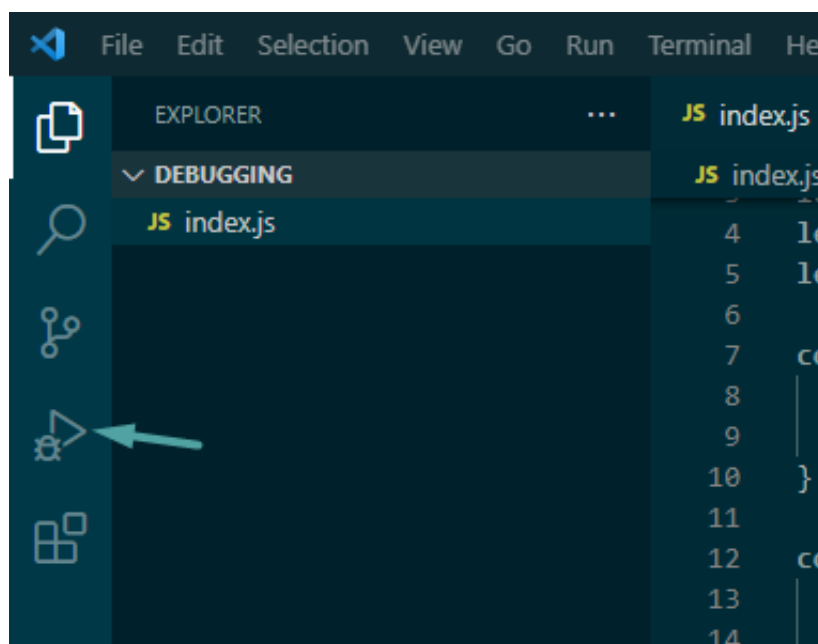
```
11
12 ✓ const restar = () => {
13     resta = suma - valorUno;
14     return resta
15 }
16
17 ● sumar(valorUno, valorDos);
18 restar();
19
20 array.push(valorUno);
21 array.push(valorDos);
22 array.push(suma);
23
24 console.log(array)
25
```



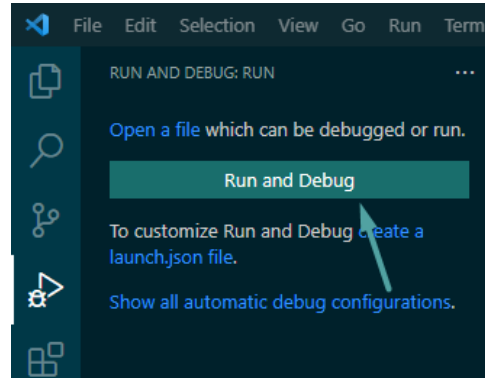
Cuando haces clic en una línea para definir un **breakpoint**, quedará marcada con un punto de color rojo fuerte.

```
15 }  
16  
17 sumar(valorUno, valorDos);  
18 restar();  
19  
20 array.push(valorUno);  
21 array.push(valorDos);  
22 array.push(suma);  
23  
24 console.log(array)  
25
```

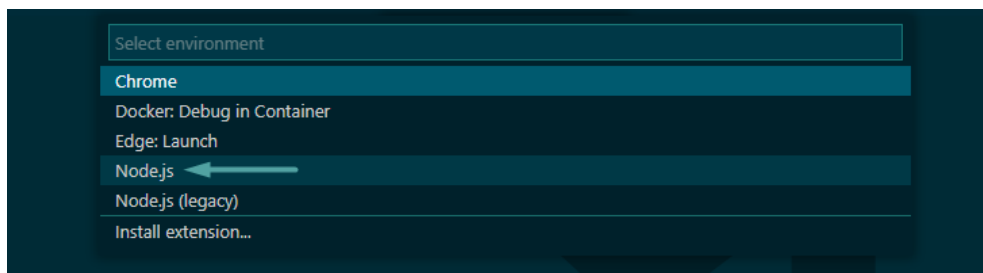
Define el **breakpoint** en la línea de invocación de la función **sumar()**. Ahora, para acceder a la herramienta de depuración de VS Code, haremos clic en el cuarto símbolo (contando de abajo hacia arriba) de la barra lateral izquierda.



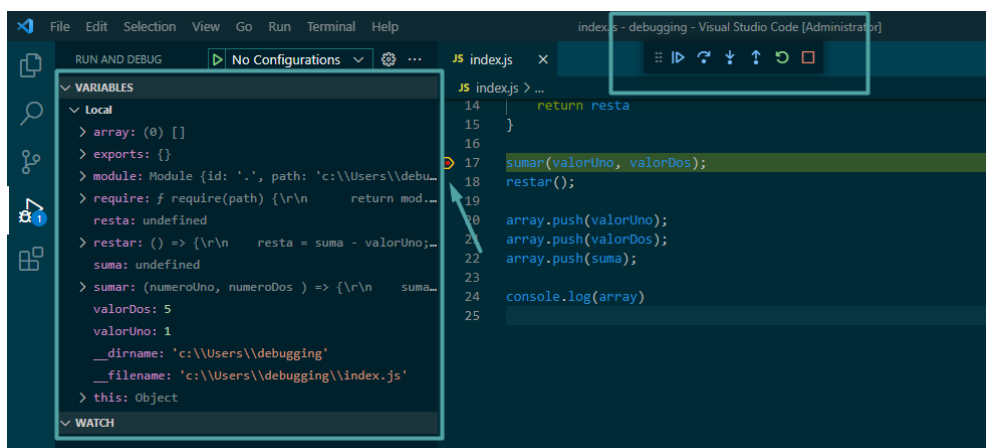
Esto abrirá una pestaña, en la que haremos clic en el botón **“Run and Debug”**.



Al hacerlo, VS Code nos preguntará por nuestro ambiente de trabajo. Debemos dar clic en **Node.js**.



Luego de esto, obtendremos la siguiente vista.



Primero hablaremos del panel izquierdo. En esta sección tenemos toda la información referente a variables, objetos, funciones, entre otros, que existen en nuestro programa, junto a su valor en el momento de la ejecución en donde se encuentre nuestro **breakpoint**.

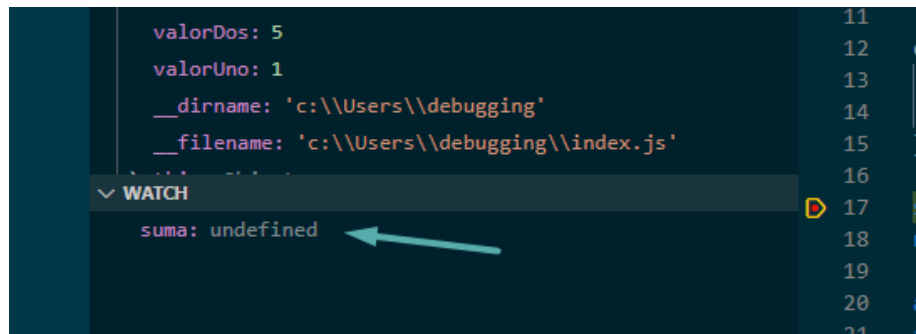
Podemos ver que se encuentra nuestra variable Array, y las variables **valorUno** y **valorDos** están con sus valores definidos, mientras que las variables **suma** y **resta** se encuentran indefinidas al momento de la ejecución, debido a que tenemos detenido nuestro programa.



Puedes también añadir un elemento a la lista **“watch”**, en caso de que quieras observar una variable en particular. Para realizar esta acción, solo debes hacer clic derecho al elemento que deseas agregar, y luego en **“add to Watch”**.



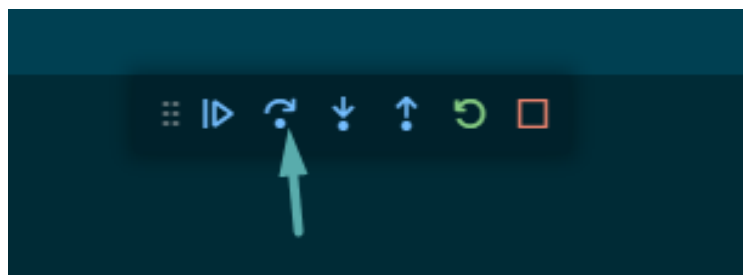
Luego de esto, verás como aparece la variable en la siguiente sección del panel izquierdo.



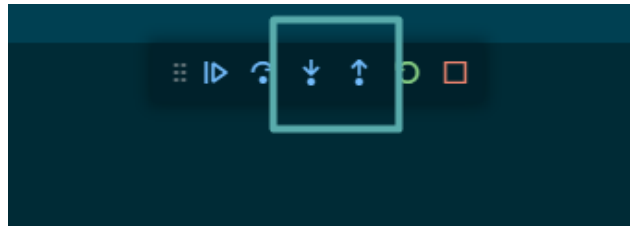
Ahora hablaremos sobre el panel de control de ejecución, y sus funciones. El primer botón "Continue" permite continuar la ejecución del programa, si es que no existe otro **breakpoint**, o entonces la sesión de Debugging solo terminará.



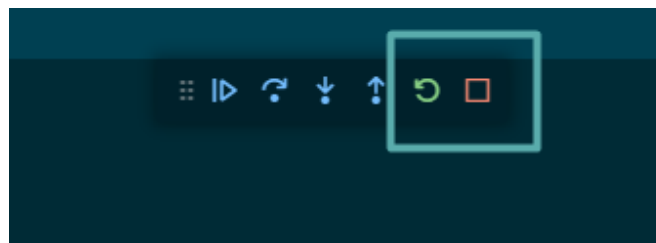
El siguiente botón "**Step Over**", permite avanzar un paso en la ejecución a un nivel macro. Por ejemplo: cuando exista una función, como en nuestro caso, donde el **breakpoint** se encuentra en la línea de invocación de la función **sumar()**, este botón pasará a la línea 18 de ejecución, y no entrará en el detalle de ejecución dentro del cuerpo de la función.



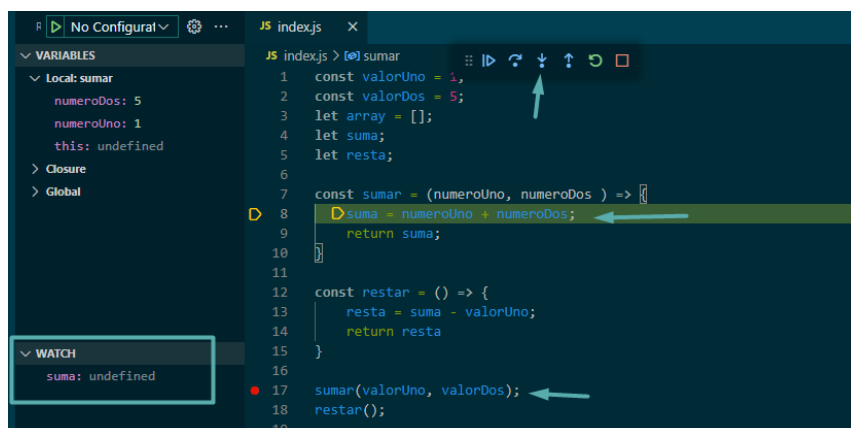
El siguiente botón **“Step Into”**, por su parte, es el que nos permite entrar en el código del cuerpo de una función para ver el detalle de ejecución; por otra parte, el botón que le sigue **“Step Out”**, nos permitirá salirnos del cuerpo de la función, y continuar con el **Debugging** fuera de ésta.



Los dos últimos botones **“Restart”** y **“Stop”**, nos permiten reiniciar el test al último **breakpoint**, y detener la ejecución de nuestra sesión de depuración, respectivamente.



Ahora que ya conocemos nuestros controles, presionaremos el tercer botón **“Step Into”**, para seguir los pasos dentro de nuestra función **sumar()**. Al hacerlo, verás como el programa avanza una línea hacia dentro en la ejecución de la función.



En este punto podemos notar que la variable suma aún no tiene el resultado como valor, y eso pasa porque esta línea de código aún no ha sido ejecutada, estamos sobre la línea, pero **no después de ella**. Por lo tanto, si presionamos el segundo botón **“Step Over”**, veremos como en la variable suma ya tiene su valor definido.

```

JS index.js
1  const valorUno = 1,
2  const valorDos = 5;
3  let array = [];
4  let suma;
5  let resta;
6
7  const sumar = (numeroUno, numeroDos) => {
8    suma = numeroUno + numeroDos;
9    return suma;
10 }
11
12 const restar = () => {
13   resta = suma - valorUno;
14   return resta;
15 }
16
17 sumar(valorUno, valorDos);
18 restar();
19

```

VARIABLES

- Local: sumar
 - numeroDos: 5
 - numeroUno: 1
 - this: undefined
- Closure
- Global

WATCH

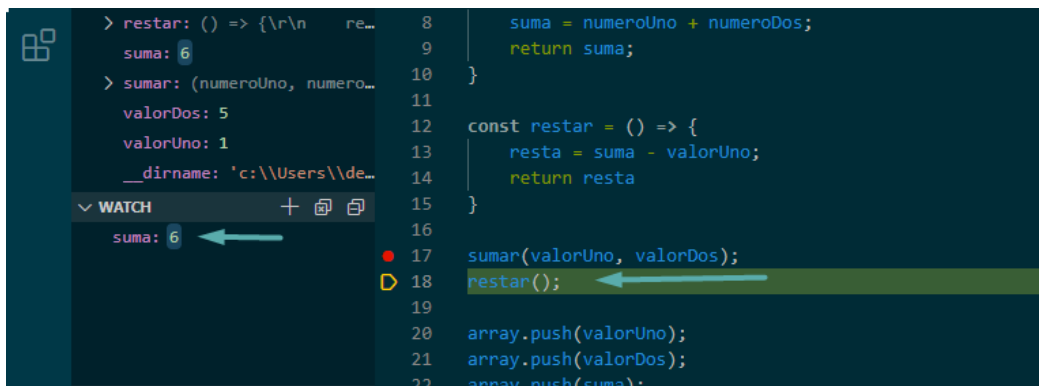
- suma: 6

También podemos notar que, a nivel local, es decir, dentro del alcance de la función, tenemos dos nuevos valores asignados, los cuales corresponden a aquellos que le hemos pasado como argumento a nuestra función.

VARIABLES

- Local: sumar
 - numeroDos: 5
 - numeroUno: 1
 - this: undefined
- Closure
- Global

Reinicia la sesión de depuración con el botón **“restart”**, y esta vez presiona el botón **“Step Over”**,

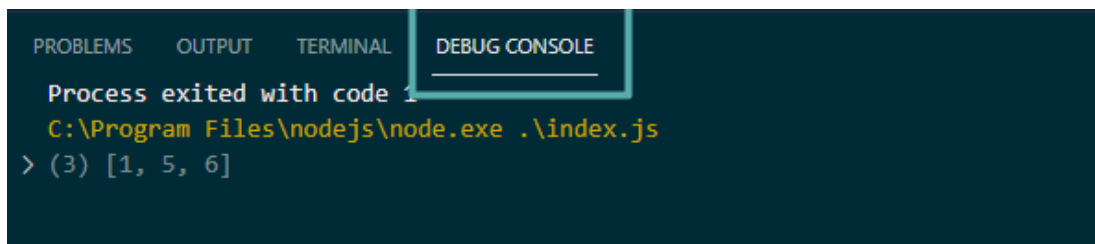


```

> restar: () => {\r\n  re... 8      suma = numeroUno + numeroDos;
  suma: 6                  9      return suma;
> sumar: (numeroUno, numero... 10   }
  valorDos: 5              11
  valorUno: 1              12   const restar = () => {
  __dirname: 'c:\\Users\\de... 13     resta = suma - valorUno;
                                14     return resta
                                15   }
                                16
                                17   sumar(valorUno, valorDos);
                                18   restar();
                                19
                                20   array.push(valorUno);
                                21   array.push(valorDos);
                                22   array.push(suma);
  
```

Podemos observar que esta vez la ejecución ha pasado directamente a la siguiente línea de código, sin entrar a las que se encuentran dentro del cuerpo de la función; también notamos que, en este punto de la ejecución, nuestra variable suma ya tiene asignado su valor.

Reinicia una vez más la sesión de depuración, y presiona el primer botón continúe. ¿Ha ocurrido algo? la sesión de depuración solo continuó hasta el final, ya que no había ningún otro **breakpoint** para detener el programa. Para observar el resultado, haz clic en la consola de depuración.



```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

Process exited with code 1
C:\Program Files\nodejs\node.exe .\index.js
> (3) [1, 5, 6]
  
```

Vuelve a iniciar la sesión de depuración, pero esta vez agrega otro **breakpoint** en la línea 20.

```

15  }
16
17  sumar(valorUno, valorDos);
18  restar();
19
20  array.push(valorUno);
21  array.push(valorDos);
22  array.push(suma);
23
24  console.log(array);
25

```

Ahora inicia la sesión de **Debugging**, una vez que ésta se encuentre en el primer **breakpoint**, presiona el botón "Continue". Con esto hemos continuado efectivamente a la línea 20 de nuestro programa, y en este punto, la variable suma y la variable resta ya tienen sus valores definidos, mientras que la variable array aún se encuentra como uno vacío.

The screenshot shows the VS Code interface during a debugging session. On the left, the 'VARIABLES' panel displays the current state of variables: 'array' is an empty array, 'resta' is 5, and 'suma' is 6. The 'WATCH' panel shows 'suma' as 6. The 'CALL STACK' panel shows the current function call. The code editor on the right shows the same code as the previous image, with a breakpoint at line 20. The execution is paused at that line.

Avanza línea por línea para observar como la variable array va recibiendo uno a uno sus valores. Una vez que esté en la línea final de nuestro programa, podemos notar como ésta tiene sus valores agregados, y que coincide con el resultado del programa al presionar el botón **“Continue”**.

```

1  const valorUno = 1;
2  const valorDos = 5;
3  let array = [];
4  let suma;
5  let resta;
6
7  const sumar = (numeroUno, numeroDos) => {
8    suma = numeroUno + numeroDos;
9    return suma;
10 }
11
12 const restar = () => {
13   resta = suma - valorUno;
14   return resta;
15 }
16
17 sumar(valorUno, valorDos);
18 restar();
19
20 array.push(valorUno);
21 array.push(valorDos);
22 array.push(suma);
23
24 console.log(array);
25

```

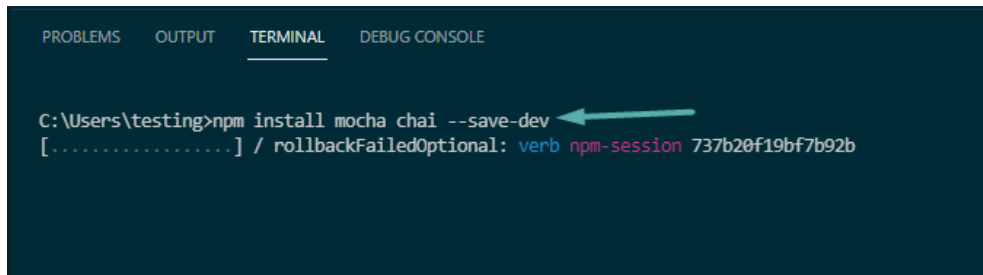
TESTING

Cuando en programación mencionamos **testing**, nos referimos a realizar validaciones a nuestro código, mediante pruebas preparadas exclusivamente para las porciones de código deseadas. Éstas nos darán como resultado una visión respecto a la calidad de nuestro programa. Todas las pruebas y herramientas implementadas nos ayudarán a tener una mejor vista de los errores, y su corrección nos permitirá tener un programa estable y maduro.

Si bien existen varias herramientas de **testing** para aplicaciones de **node.js**, algunas de las más populares y utilizadas actualmente son **Jest** y **Mocha**; este último normalmente se usa junto a **Chai**, una librería de aserción (logra que los test tengan una semántica que permite una fácil lectura).

Para la introducción al testing usaremos **Mocha y Chai**. Mocha es un **“test runner”**, y será el encargado de correr todos nuestros test; mientras que **Chai**, como mencionamos anteriormente, es una **“assertion library”** o librería de aserciones, y nos permitirá definir las características del test.

Crearemos una nueva carpeta, e iniciaremos un proyecto npm utilizando el comando `npm init`. Luego, instalaremos mocha y chai, pero usaremos las opciones `--save-dev` para guardar estos módulos como parte de nuestro ambiente de desarrollo.

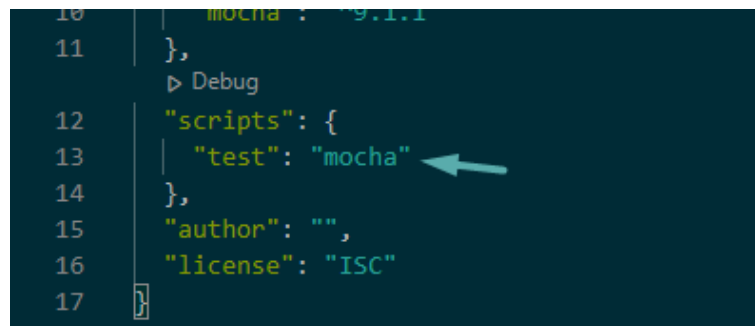


```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\testing>npm install mocha chai --save-dev
[.....] / rollbackFailedOptional: verb npm-session 737b20f19bf7b92b
  
```

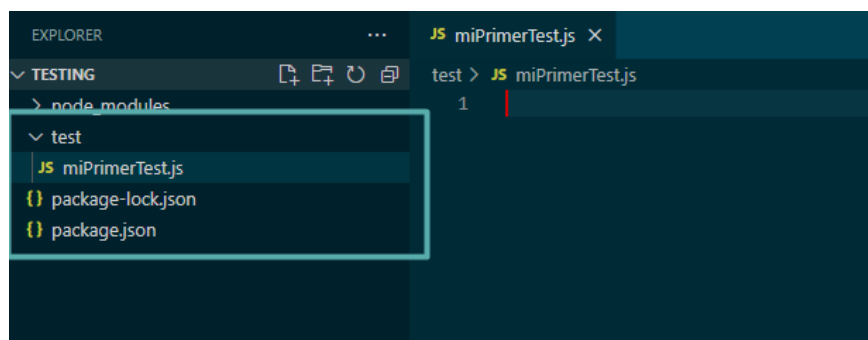
En el archivo `package.json`, editaremos en la sección `scripts` la propiedad `test`, para así habilitar el uso de `Mocha` en nuestro programa.



```

10   "mocha": "9.1.1",
11 },
12   "scripts": {
13     "test": "mocha"
14   },
15   "author": "",
16   "license": "ISC"
17 }
  
```

Mocha buscará nuestros tests en una carpeta llamada `test por defecto`. La crearemos, y dentro de ella, un archivo `miPrimerTest.js`.

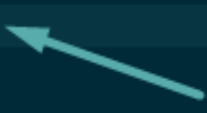


```

EXPLORER
├── TESTING
│   ├── node_modules
│   └── test
│       ├── JS miPrimerTest.js
│       ├── {} package-lock.json
│       └── {} package.json
  
```

Para escribir nuestro primer test, importaremos el paquete chai de la siguiente forma.

```
JS miPrimerTest.js X
test > JS miPrimerTest.js > ...
1  const assert = require('chai').assert;
2
3
```



Esto nos permitirá usar la sintaxis de aserción que provee chai. Ahora utilizaremos el método `describe()` que pertenece a mocha, éste toma como primer argumento un nombre descriptivo respecto al test realizado, y una función que contendrá el test en sí.

```
JS miPrimerTest.js X
test > JS miPrimerTest.js > ...
1  const assert = require('chai').assert;
2
3  describe('Verifica string', () => {
4
5
6  })
7
```

Dentro de la función, escribiremos nuestro test, el cual verificará si determinado valor es de tipo String. Para esto, utilizamos el método `it()`, que nos permite declarar un nuevo test; recibe como primer argumento su descripción, y como segundo una función que contendrá sus condiciones. Dentro del cuerpo de la función, utilizaremos el método `isString()`, el cual recibe como primer parámetro el valor a evaluar, y como segundo parámetro el mensaje en caso de no pasar el test.

```
JS miPrimerTest.js X
test > JS miPrimerTest.js > ...
1  const assert = require('chai').assert;
2
3  describe('Verifica string', () => {
4
5    it('Verifica si valor entregado es string', () => {
6
7      const string = 'Este es un texto de prueba';
8      assert.isString(string, 'No es una cadena');
9    });
10
11  });
12
13
```

En la terminal, haremos la prueba utilizando el comando `npm run test`.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\testing>npm run test

> testing@1.0.0 test C:\Users\testing
> mocha

Verifica string
  ✓ Verifica si valor entregado es string

1 passing (5ms)

C:\Users\testing>
```


Veamos qué sucede si ahora cambiamos el valor entregado al método `isString()`.

```
JS miPrimerTest.js X
test > JS miPrimerTest.js > ...
1  const assert = require('chai').assert;
2
3  describe('Verifica string', () => {
4
5      it('Verifica si valor entregado es string', () => {
6
7          const string = 'Este es un texto de prueba';
8          const number = 7;
9
10         assert.isString(number, 'No es una cadena');
11     })
12 })
13
14
```

Y utilizando nuevamente el comando `npm run test`, vemos como el resultado esta vez es negativo.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\testing>npm run test

> testing@1.0.0 test C:\Users\testing
> mocha

Verifica string
  1) Verifica si valor entregado es string

0 passing (51ms)
1 failing

1) Verifica string
  Verifica si valor entregado es string:
AssertionError: No es una cadena: expected 7 to be a string
    at Context.<anonymous> (test/miPrimerTest.js:10:16)
    at processImmediate (internal/timers.js:461:21)

npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! testing@1.0.0 test: `mocha`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the testing@1.0.0 test script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

npm ERR! A complete log of this run can be found in:
npm ERR! C:\Users\Administrador\AppData\Roaming\npm-cache\_logs\2021-09-08T18_56_23_344Z-debug.log
```

Generalmente, las pruebas se realizarán de forma unitaria, probando distintas porciones del código. Cuando hemos mencionado que un código modular permite que un programa sea más fácil de depurar, es justamente para estos casos, ya que te encontrarás revisando funciones que se encuentran declaradas en archivos distintos, y son importadas y utilizadas por tu programa principal.

Para demostrar un ejemplo de este caso, crearemos un archivo `utils.js`, dentro escribiremos una función que retorne un string, y luego la exportaremos.

```
JS utils.js  X
JS utils.js > ...
1  const devuelveString = () => {
2    |    return "Este es un texto de pruebas"
3  }
4
5  module.exports = { devuelveString : devuelveString };
```

Recomendación: cuando estés declarando un objeto, puedes utilizar una sintaxis corta si es que la llave y el valor de éste se llamarán de la misma forma.

```
JS utils.js  X
JS utils.js > ...
1  const devuelveString = () => {
2    |    return "Este es un texto de pruebas"
3  }
4
5  module.exports = { devuelveString }; ←
6
```

Ahora, importaremos esta función en nuestro archivo `miPrimerTest.js`.

```
JS miPrimerTest.js x
test > JS miPrimerTest.js > describe("Verifica string") callback > it("Verifica si valor entregado es string") callback
1  const assert = require('chai').assert;
2  const { devuelveString } = require('../utils'); ←
3
4  describe('Verifica string', () => {
5
6      it('Verifica si valor entregado es string', () => {
7          ←
8          assert.isString(devuelveString(), 'No es una cadena');
9      })
10
11
12 })
13
```

Y si ejecutamos una vez más el comando `npm run test`, podemos observar que nuestra función pasa el test.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\testing>npm run test

> testing@1.0.0 test C:\Users\testing
> mocha

Verifica string
  ✓ Verifica si valor entregado es string

1 passing (12ms)
```

Chai también tiene un módulo que nos permite hacer pruebas a nuestro servidor, éste se llama **chai-http**, y lo añadiremos a las dependencias de nuestro proyecto utilizando el comando **"npm install chai-http --save-dev"**. Luego, crearemos un pequeño servidor que retornará un texto como respuesta, dentro de un nuevo archivo **index.js** que se debe crear en la misma ruta que el archivo **utils.js**. La única diferencia con nuestros servidores anteriores, es que esta vez exportaremos el servidor utilizando **"module.exports"**.



The screenshot shows the VS Code Explorer on the left with the 'test' folder selected. The main editor shows the content of 'index.js' with the following code:

```

1  const http = require('http');
2
3  const servidor = http.createServer((req, res) => {
4    res.write('Respuesta desde servidor')
5    res.end();
6  });
7
8  servidor.listen(3000, () => {
9    console.log('Servidor funcionando en http://localhost:3000/');
10 });
11
12 module.exports = { servidor };
  
```

A red arrow points to the 'test' folder in the Explorer, and another red arrow points to the 'module.exports' line in the code.

Ahora, crearemos un nuevo test para nuestro servidor. Dentro de la carpeta test, crea un nuevo archivo **.js**. Requeriremos los módulos **chai** y **chai-http**, esta vez sin el método **assert**.



The screenshot shows the VS Code editor with the file 'testServidor.js' open. The code is as follows:

```

1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3
4
  
```

A red box highlights the first two lines of code.

También requeriremos nuestro servidor.



The screenshot shows the VS Code editor with the file 'testServidor.js' open. The code is as follows:

```

1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3  const { servidor } = require('../index');
4
  
```

A red arrow points to the third line of code, which imports the 'servidor' from the 'index' file.

Y para poder utilizar los métodos de chai-http, debemos indicarle al módulo **chai** que haga uso de chai-http, con la siguiente línea de código.

```
JS testServidor.js X
test > JS testServidor.js > ...
1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3  const { servidor } = require('../index');
4
5  chai.use(chaiHttp);
6
7
8
```

Comenzaremos de la misma forma que el test anterior, con los métodos **describe()** e **it()**.

```
JS testServidor.js X
test > JS testServidor.js > ...
1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3  const { servidor } = require('../index');
4
5  chai.use(chaiHttp);
6
7  describe('Probando respuesta de servidor', () => {
8    it('Comprueba que respuesta de servidor es el string "Respuesta desde servidor"', () => {
9
10
11    })
12  })
13
14
```

Ahora, para poder realizar una petición a nuestro servidor, utilizaremos el método **request()**, seguido del método **get()**. Para **request**, pasamos como argumento nuestro servidor, y luego debemos especificar la ruta, pasándola como argumento a **get()**.

```
JS testServidor.js X
test > JS testServidor.js > ...
1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3  const { servidor } = require('../index');
4
5  chai.use(chaiHttp);
6
7  describe('Probando respuesta de servidor', () => {
8    it('Comprueba que respuesta de servidor es el string "Respuesta desde servidor"', () => {
9
10      chai.request(servidor).get('/')
11
12    })
13  })
14
```

Y para especificar que la petición ha terminado, utilizamos el método **end**, el cual tiene como parámetros el error y la respuesta.

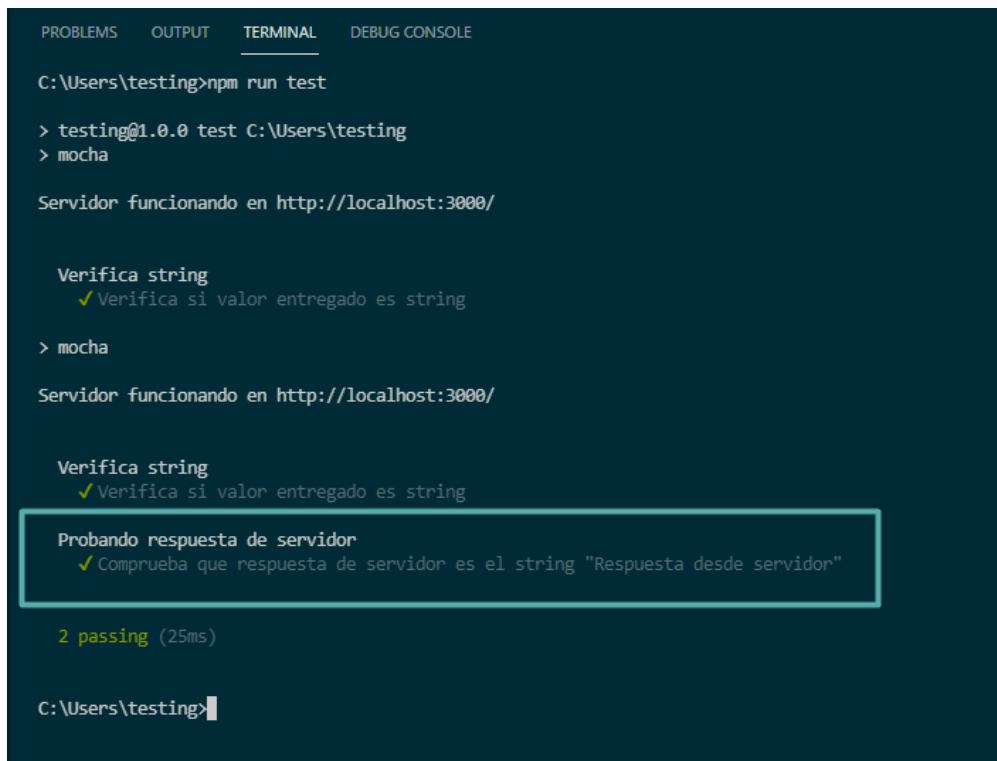
```
JS testServidor.js
test > JS testServidor.js > ...
1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3  const { servidor } = require('../index');
4
5  chai.use(chaiHttp);
6
7  describe('Probando respuesta de servidor', () => {
8    it('Comprueba que respuesta de servidor es el string "Respuesta desde servidor"', () => {
9
10     chai.request(servidor).get('/').end((error, respuesta) => {
11
12     })
13   })
14 })
15 })
16
```

Ya solo nos queda establecer las condiciones del test, esta vez utilizando el método **assert()** y el método **equal()**. **equal()** recibe tres argumentos: el valor actual, el valor esperado, y el mensaje en caso de error. En este ejemplo, comprobaremos que `respuesta.text` contenga el valor que envía el servidor al realizar una consulta de tipo **get**.

```
JS testServidor.js X
test > JS testServidor.js > ...
1  const chai = require('chai')
2  const chaiHttp = require('chai-http');
3  const { servidor } = require('../index');
4
5  chai.use(chaiHttp);
6
7  describe('Probando respuesta de servidor', () => {
8    it('Comprueba que respuesta de servidor es el string "Respuesta desde servidor"', () => {
9
10     chai.request(servidor).get('/').end((error, respuesta) => {
11       chai.assert.equal(respuesta.text, 'Respuesta desde servidor', "La respuesta no ha sido la esperada");
12     })
13   })
14 })
15 })
```



Y al momento de correr nuestro test, podemos observar que primero corre “verifica string”, y luego el test en nuestro servidor, el cual es exitoso.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\testing>npm run test

> testing@1.0.0 test C:\Users\testing
> mocha

Servidor funcionando en http://localhost:3000/

Verifica string
  ✓ Verifica si valor entregado es string

> mocha

Servidor funcionando en http://localhost:3000/

Verifica string
  ✓ Verifica si valor entregado es string

  Probando respuesta de servidor
    ✓ Comprueba que respuesta de servidor es el string "Respuesta desde servidor"

2 passing (25ms)

C:\Users\testing>
```

CÓDIGOS HTTP

El protocolo HTTP define ciertos códigos de respuesta, los cuales permiten conocer el estado de esta misma, o las posibles razones para el fallo de una respuesta.

Estos códigos están agrupados por tipos de respuesta:

RESPUESTAS INFORMATIVAS (1xx)

Corresponden a los códigos dentro del rango de los 100.

RESPUESTAS SATISFACTORIAS (2xx)

Los códigos dentro del rango de los 200, son considerados respuestas procesadas exitosamente por el servidor. Algunos de los más comunes son:

- 200: Ok
- 201: Created
- 202: Accepted
- 204: No content

REDIRECCIONES (3xx)

Los códigos en este rango comprenden redirecciones de URL, donde una petición a cierta URL luego es derivada a otra.

Errores de cliente (4xx)

Los códigos 400 son códigos de error, dentro de los más comunes podemos encontrar:

- 400: Bad Request
- 401: Unauthorized
- 402: Forbidden
- 403: Not found

ERRORES DE SERVIDOR (5xx)

Estos códigos definen respuestas que han sido recibidas correctamente, pero no han podido ser procesadas por el servidor. Los más comunes de este grupo son:

- 500: Internal Server Error
- 501: Not implemented
- 503: Service Unavailable
- 508: Loop detected

Dentro de nuestro servidor, podemos definir el código de respuesta dependiendo del proceso de nuestro programa, y las posibles respuestas que podríamos obtener, ya sean positivas o negativas. Esto también es útil al momento de realizar testing, pues permite crear nuestro test para leer el código de respuesta HTTP.

Definiendo la respuesta desde el servidor:

```
JS index.js > ...
1  const http = require('http');
2
3  const servidor = http.createServer((req, res) => {
4    res.statusCode = 201; ←
5    res.write('Respuesta desde servidor')
6    res.end();
7  });
8
9  servidor.listen(3000, () => {
10   console.log(`Servidor funcionando en http://localhost:3000/`);
11 });
12
13 module.exports = { servidor };
```

Validando código en testing:

```
6
7  describe('Probando respuesta de servidor', () => {
8    it('Comprueba que respuesta de servidor es el string "Respuesta desde servidor"', () => {
9
10     chai.request(servidor).get('/').end((error, respuesta) => {
11       chai.assert.equal(respuesta.status, 200, "La respuesta no ha sido la esperada");
12     })
13   })
14 })
15
16
```