

```

        modifier_ob.modifiers.new("mirror_mirror", "MIRROR")

    # Assign object to mirror_ob
    mirror_mod.mirror_object = mirror_ob

    # Mirror operation
    operation = "MIRROR_X":
        mirror_mod.use_x = True
        mirror_mod.use_y = False
        mirror_mod.use_z = False
    elif_operation == "MIRROR_Y":
        mirror_mod.use_x = False
        mirror_mod.use_y = True
        mirror_mod.use_z = False
    elif_operation == "MIRROR_Z":
        mirror_mod.use_x = False
        mirror_mod.use_y = False
        mirror_mod.use_z = True

    #selection at the end-add back the deselected mirror modifier object
    mirror_ob.select=1
    modifier_ob.select=1
    bpy.context.scene.objects.active = modifier_ob
    print("Selected" + str(modifier_ob)) # modifier ob is the active ob
    #mirror_ob.select = 0
    #one = bpy.context.selected_objects[0]
    #bpy.data.objects[one.name].select = 1
except:
    print("please select exactly two objects, the last one gets the modifier unless its

----- OPERATOR CLASSES -----
Mirror Tool
mirror_ob.select=1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier

MirrorX(bpy.types.Operator):
    """
    This adds an X mirror to the selected object
    """
    name = "object.mirror_mirror_x", context.selected_objects[0]
    name = "Mirror X"
    bpy.data.objects[one.name].select = 1
except:
    print("please select exactly two objects, the

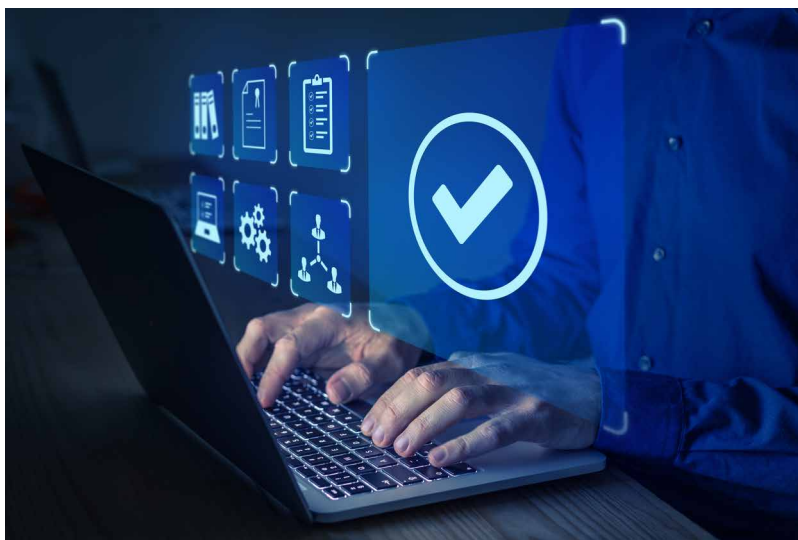
----- OPERATOR CLASSES -----

```

Programación con JavaScript II

Pruebas

Introducción



Parte fundamental del proceso de la construcción e implementación de una aplicación web es la etapa de pruebas. Es aquí donde el desarrollador de software asegura la calidad del producto a entregar, es decir confirma que cumple con las expectativas del usuario o incluso las supera.

Las pruebas permiten descubrir errores o “bugs” que podrían impactar en diferente medida a las organizaciones, desde problemas simples en el proceso de la información hasta pérdidas económicas o incluso humanas.

La detección oportuna de los errores es un proceso arduo que está basado principalmente en la intuición del programador y que depende en gran medida del tiempo que se destine a realizar pruebas durante el proyecto, que por lo general nunca es suficiente.

En este tema abordaremos los conceptos de los tipos de prueba del software que podemos realizar para cualquier proyecto de desarrollo. Conoceremos una herramienta creada por Facebook en 2011 que posteriormente fue convertida en open source para hacer pruebas de alto rendimiento de manera automática y que cuenta con una configuración muy simple llamada JEST con la que podrás analizar la calidad de bloques de código antes de su lanzamiento.

Explicación

Tipos de pruebas

Pruebas unitarias

Este tipo de pruebas son las que frecuentemente se realizan para comprobar secciones o bloques de código JavaScript, ya sea comprobando la salida esperada en Html o en la consola del navegador, o bien suministrando diferentes valores y comprobando que las funciones y clases se comporten como esperamos.

Pruebas de integración

En ciertos proyectos en los que intervienen unidades de desarrolladores independientes, conviene establecer mecanismos de pruebas para que cada componente del sistema esté adecuadamente integrado. Las pruebas de integración ayudan a comprobar que las partes de la aplicación construida se encuentren adecuadamente acopladas y funcionen como un todo. En estas pruebas será necesario verificar la comunicación entre cada componente, las entradas, sus procesos, las funciones y salidas. Aquí conviene establecer con claridad modelos tradicionales como la arquitectura MVC (Modelo – Vista -Controlador) de tal suerte que la integración está dividida entre el almacenamiento de los datos, la interfaz de usuario y los procesos de control.

Pruebas funcionales UAT (User Acceptance Test)

Se espera que el sistema cumpla con las expectativas del usuario final. Para ello es necesario que se creen casos de prueba en los que se documentan la secuencia de pasos y las bifurcaciones deseadas, que garanticen que el sistema o aplicación responda como se espera. Generalmente los casos de prueba son elaborados por un representante del usuario final, y son entregados al equipo de desarrollo previo a que el usuario los realice por su cuenta, lo cual ayuda a que se realicen las pruebas unitarias y de integración necesarias.

Pruebas Beta.

Las pruebas beta son bastante útiles para garantizar la calidad del software, ya que se pone a la disposición de ciertos usuarios el sistema para que por algún tiempo puedan probarlo previo al lanzamiento oficial. Estas pruebas piloto ayudan a detectar errores o bugs que no se aparecieron en las pruebas unitarias, de integración o pruebas UAT. Es importante la participación del usuario y de un grupo de desarrolladores que estén documentando sus resultados, para que se vayan realizando los ajustes necesarios o bien detener la prueba beta y regresar a etapas de construcción.

Herramientas para realizar pruebas:

En la actualidad existen muchas plataformas y paquetes para realizar pruebas de aplicaciones construidas en JavaScript. Zaidman (2018) propone la siguiente clasificación de herramientas para realizar pruebas:

1. Las que proveen una infraestructura de pruebas:
 - a. **Mocha** (<https://mochajs.org>)
 - b. **Jasmine** (<https://jasmine.github.io/>)
 - c. **Jest** (<https://jestjs.io/>)
 - d. **Cucumber** (<https://cucumber.io/docs/installation/javascript/>)

2. Las que permiten generar y comparar “instantáneas” de componentes y de estructuras de datos:
 - a. **Jest** (<https://jestjs.io/>)
 - b. **AVA** (<https://github.com/avajs>)
3. Las que proveen de maquetas para pruebas unitarias:
 - a. **Sinon** (<https://sinonjs.org/>)
 - b. **Jasmine** (<https://jasmine.github.io/>)
 - c. **enzyme** (<https://enzymejs.github.io/enzyme/>)
 - d. **Jest** (<https://jestjs.io/>)
 - e. **Cucumber** (<https://cucumber.io/docs/installation/javascript/>)
 - f. **Testdouble** <https://github.com/testdouble/testdouble.js#readme>
4. Las que permiten generar reportes de cobertura:
 - a. **Istanbul** (<https://gotwarlost.github.io/istanbul/>)
 - b. **Jest** (<https://jasmine.github.io/>)
5. Las que consisten en pruebas end-to-end:
 - a. **Nightwatch** (<https://nightwatchjs.org>)
 - b. **Phantom** (<https://phantomjs.org/>),

JEST

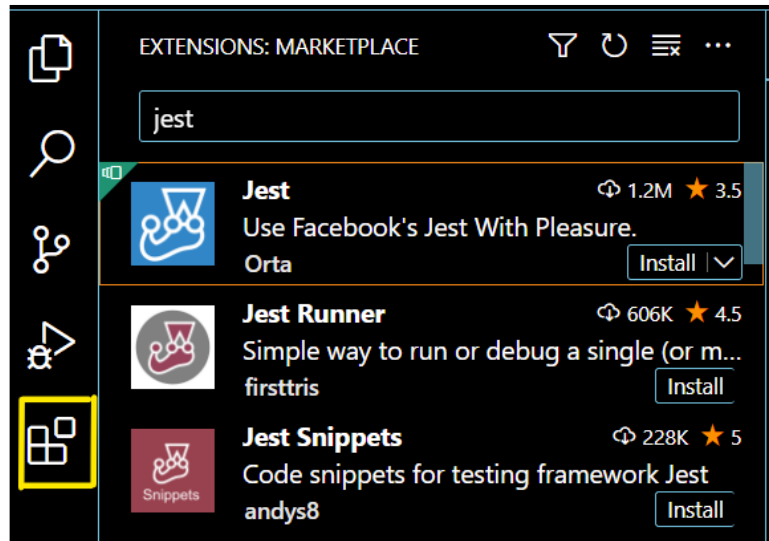
Según Joshi (2020), **Jest** es un marco de trabajo (framework) de prueba muy popular, simple de instalar, creado por la compañía Facebook, que permite realizar pruebas unitarias de JavaScript. Sus características distintivas son:

- Es compatible con muchos otros frameworks populares como NodeJS, Angular, React.
- Incluye la biblioteca completa para trabajar con código JavaScript, por lo que es fácil de configurar.
- Es compatible con TypeScript y puede hacer pruebas con la interfaz del usuario, lo que garantiza estabilidad entre diferentes versiones.
- Se considera una herramienta de prueba muy ágil y versátil, ya que puede manipular el tiempo de ejecución con funciones como `setTimeout()`, `setInterval()`, `clearTimeout()`, `clearInterval()`.

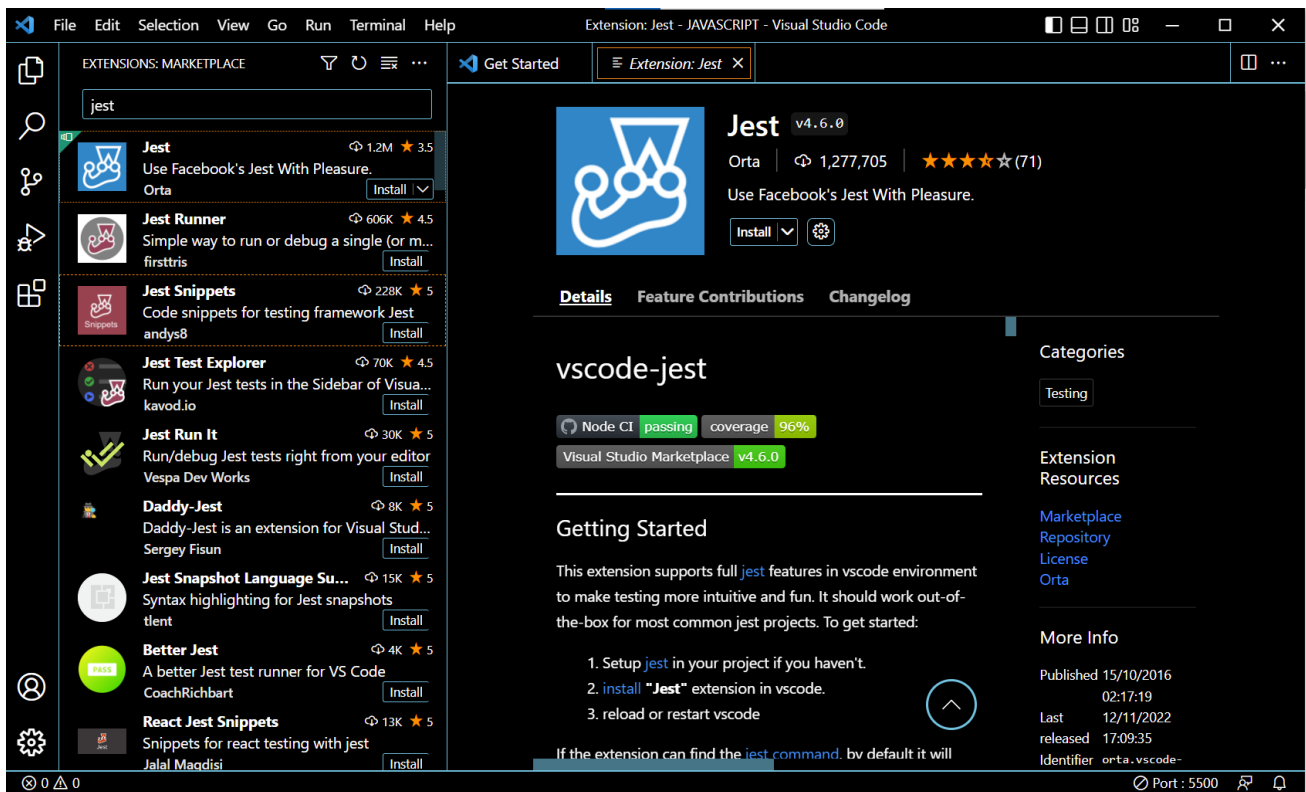
Pasos para instalar Jest:

Requisito previo: antes de instalar **JEST** en Visual studio Code, es necesario incluir Node.js, ya que sobre esta plataforma operan sus las funciones de prueba. Podemos descargarlo desde: <https://nodejs.org/es/> una vez realizado esto, proceder con los pasos siguientes:

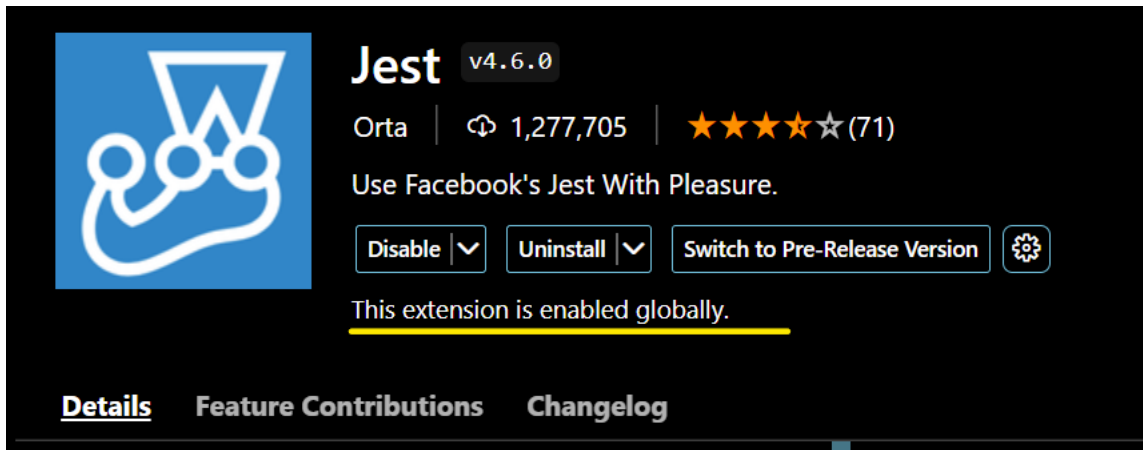
Paso 1. Accede a las extensiones de Visual Studio Code donde encontrarás JEST. Las siguientes imágenes son tomadas de la misma aplicación de escritorio.



Paso 2. Busca Jest e instala la extensión haciendo clic en “install” como se muestra en la figura siguiente.



Paso 3. Asegura que aparezca la leyenda “**This extensión is enabled globally**” como puedes observar en la imagen siguiente que muestra el estatus de la extensión JEST en Visual Studio Code.



Paso 4. Inicia **JEST** desde la **TERMINAL** con la siguiente instrucción:

```
[Path local] > npm init
```

La siguiente imagen muestra el resultado de inicializar la librería JEST

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\fgarc\Documents\JavaScript> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (javascript) |
```

Paso 5. Instala las librerías de **JEST** con la siguiente instrucción en la TERMINAL.

```
[Path local] > npm install --save-dev jest
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

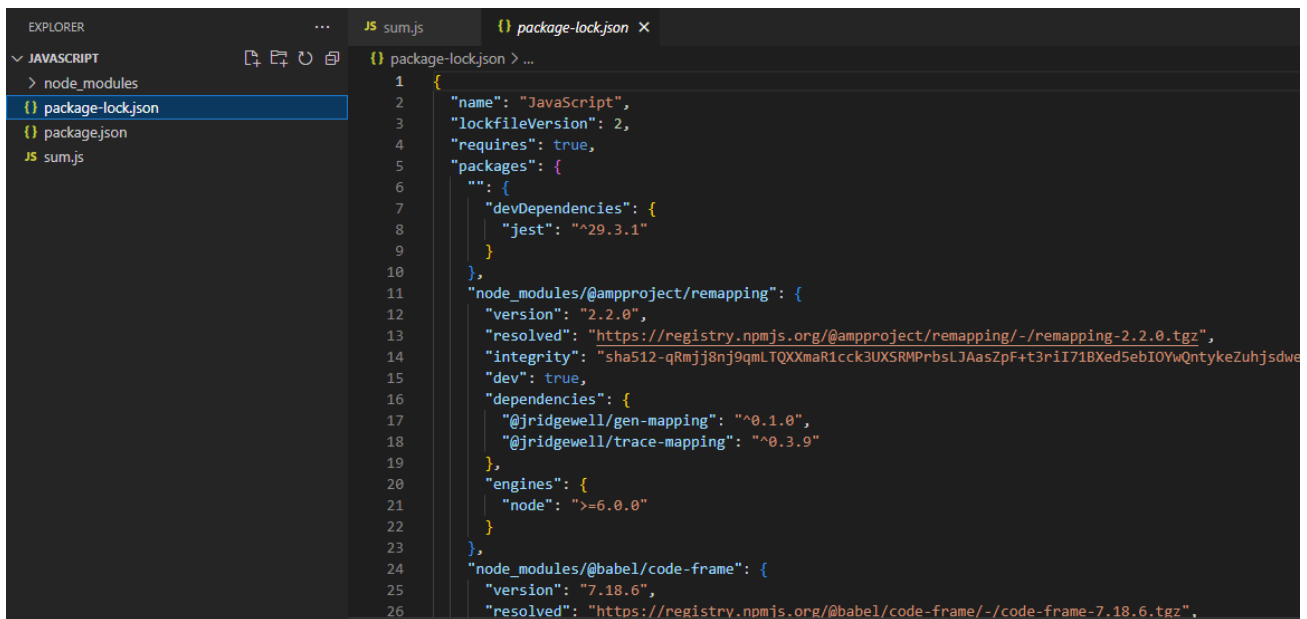
PS C:\Users\fgarc\Documents\JavaScript> npm install --save-dev jest

added 275 packages, and audited 276 packages in 8m

29 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\fgarc\Documents\JavaScript>
```

El resultado del paso anterior, donde se ejecuta **JEST** es la creación de dos archivos **package-lock.json** y **package.json**. La siguiente imagen en la que aparecen ambos archivos muestra el contenido del archivo **package-lock.json**.



```
EXPLORER  ...  JS sum.js  {} package-lock.json X

JAVASCRIPT
  > node_modules
    {} package-lock.json
    {} package.json
    JS sum.js

{} package-lock.json > ...
1  {
2    "name": "JavaScript",
3    "lockfileVersion": 2,
4    "requires": true,
5    "packages": {
6      "": {
7        "devDependencies": {
8          "jest": "^29.3.1"
9        },
10     },
11     "node_modules/@ampproject/remapping": {
12       "version": "2.2.0",
13       "resolved": "https://registry.npmjs.org/@ampproject/remapping/-/remapping-2.2.0.tgz",
14       "integrity": "sha512-qRmjj8nj9qmlTQXXmaR1cck3UXSRMPPrbsLJAasZpF+t3riI718Xed5ebIOYwQntykeZuhjsdwe
15     "dev": true,
16     "dependencies": {
17       "@jridgewell/gen-mapping": "^0.1.0",
18       "@jridgewell/trace-mapping": "^0.3.9"
19     },
20     "engines": {
21       "node": ">=6.0.0"
22     }
23   },
24   "node_modules/@babel/code-frame": {
25     "version": "7.18.6",
26     "resolved": "https://registry.npmjs.org/@babel/code-frame/-/code-frame-7.18.6.tgz",
```

Proceso para iniciar pruebas unitarias con JEST:

Una vez que se haya instalado **JEST** se estará listo para realizar pruebas del código JavaScript. Para ello, será necesario asegurarse que en la librería **package.json** el valor de **"test"** sea **"jest"**. Por lo que debemos abrir el archivo y cambiar el valor por defecto al valor **"jest"**.


```
"scripts": {  
  "test": "jest"  
},
```

En **JEST** es necesario que creamos un archivo **.test.js** con las funciones de prueba.

A continuación, se realizan algunas pruebas básicas al código de una *suma.js*

```
function sum(a,b){  
  return a + b  
}  
module.exports =sum
```

Paso 1. Se crea un archivo llamado **sum.test.js**

```
1. const sum=require('./sum')  
2. test('Suma 1+2 es igual a 3',() => {  
3.   expect(sum(1,2)).toBe(3)  
4. });
```

Analicemos línea por línea el código del archivo **sum.test.js**:

1. Tomará el archivo JavaScript llamado *sum.js* (la extensión se puede omitir) y lo guardará en la constante "sum".
2. Utilizará `test('String',())` como una función arrow.
3. Evaluará el resultado de la función *sum* a través del método `expect()`, incluyendo como parámetros los valores 1 y 2, utilizando el *matcher* "toBe" esperando que sea igual a 3.

Observa que **JEST** utiliza funciones para comparar valores llamados "**matchers**". En el ejemplo anterior el **matcher** utilizado es **toBe()**.

Ahora en la **TERMINAL** se escribe la instrucción **npm test**. Al ejecutarse se observa que la prueba corrió evaluando la función en 5 ms de tiempo, colocando la leyenda "PASS". En la siguiente imagen se observa el reporte de prueba resultado de ejecutar la instrucción `npm test` en la terminal del archivo *suma.test.js*


```
JS sum.test.js > ...
1  const sum = require('./sum');
2
3  test('Suma 1+2 es igual a 3', () => {
4    expect(sum(1, 2)).toBe(3);
5  });
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\fgarc\Documents\JavaScript> npm test
Debugger attached.

> javascript@1.0.0 test
> jest

Debugger attached.
PASS ./sum.test.js (5.923 s)
  ✓ adds 1 + 2 to equal 3 (5 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.836 s
Ran all test suites.
Waiting for the debugger to disconnect...
Waiting for the debugger to disconnect...
PS C:\Users\fgarc\Documents\JavaScript> |
```

Matchers

JEST utiliza algunas funciones llamadas “matchers” para comparar los resultados del código con los valores esperados. Los más comunes son `toBe` y `toMatch`. Sin embargo, existen otros que pueden ayudar a evaluar valores con diferentes condiciones. A continuación, se explican algunos:

Las funciones para valores **numéricos** son las siguientes y por su significado en inglés su explicación no es necesaria:

- `toBeGreaterThan(valor)`
- `toBeGreaterThan(valor)`
- `toBeLessThan(valor)`
- `toBeLessThanOrEqual(valor)`
- `toBe(valor)`
- `toEqual(valor)`

Ejemplos de **matchers** para valores numéricos:

```
test('Probar dos más dos', () => {  
  const value = 2 + 2;  
  expect(value).toBeGreaterThan(3);  
  expect(value).toBeGreaterThanOrEqual(3.5);  
  expect(value).toBeLessThan(5);  
  expect(value).toBeLessThanOrEqual(4.5);  
  
  // toBe y toEqual son equivalentes al usar números  
  expect(value).toBe(4);  
  expect(value).toEqual(4);  
});
```

Matchers para valores del tipo string:

- `not.toMatch(string)`
- `toMatch(string)`

Ejemplos de **matchers** para valores numéricos:

```
test('Existe una a en la palabra Equipo', () => {  
  expect('equipo').not.toMatch(/a/);  
});  
  
test('Aparece la palabra "stop" en el nombre Christoph', () => {  
  expect('Christoph').toMatch(/stop/);  
});
```

Facebook Open Source (2022), ofrece una documentación completa de otras funciones que **JEST** utiliza para evaluar el código en la siguiente página: <https://jestjs.io/docs/expect>

Explicación

Reportes de cobertura del código:

JEST permite generar un reporte de las funciones que abarcó durante la ejecución del código llamado reporte de cobertura. Para generar este reporte es necesario abrir el archivo `package.json` y modificar el valor "test" a "jest --coverage"

```
"scripts": {
  "test": "jest --coverage"
},
```

Al ejecutar de nuevo una prueba desde la terminal mediante la instrucción `npm test`, se genera un resumen sobre las funciones que se han probado. Incluso crea una carpeta llamada `coverage` > `lcov-report` en la que podrás encontrar el mismo reporte en formato html (`index.html`)

En la siguiente imagen se puede observar el contenido del reporte **coverage** en la **TERMINAL**:

```
JS sum.js > ...
1  function sum(a,b){
2      return a + b;
3  }
4  module.exports =sum;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\fgarc\Documents\JavaScript> npm test
Debugger attached.

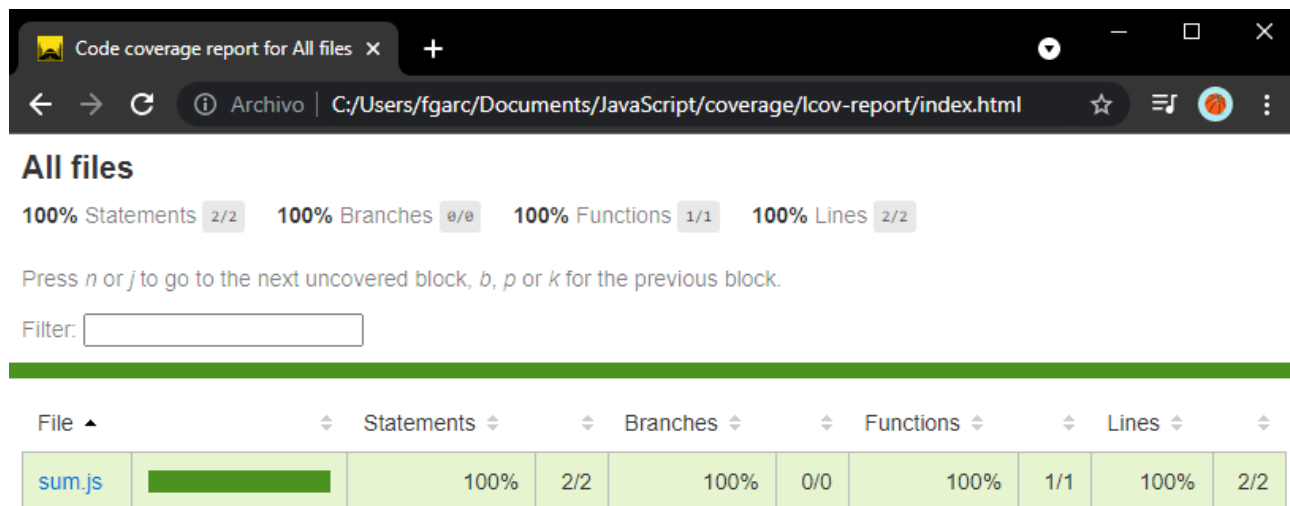
> javascript@1.0.0 test
> jest --coverage

Debugger attached.
PASS ./sum.test.js
  ✓ Suma 1+2 es igual a 3 (4 ms)

-----|-----|-----|-----|-----|-----
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files |    100   |    100   |    100   |    100   |
sum.js    |    100   |    100   |    100   |    100   |
-----|-----|-----|-----|-----|-----

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.925 s
Ran all test suites.
Waiting for the debugger to disconnect...
Waiting for the debugger to disconnect...
PS C:\Users\fgarc\Documents\JavaScript>
```

En la siguiente imagen se puede observar el **Reporte Code Coverage** generado a través de index.html.



La inmersión de software en la vida cotidiana es cada vez más común, desde aparatos electrodomésticos hasta los automóviles dependen en gran medida del poder de procesamiento de cómputo basado en líneas de código, lo cual nos hace pensar en la importancia de una programación libre de errores.

En la actualidad tanto las organizaciones multimillonarias dedicadas al desarrollo de aplicaciones, como el desarrollo *open source*, tienen a su disposición una serie de herramientas de apoyo durante el proceso de *testing*.

En este tema conocimos **JEST** una herramienta que nos permite realizar pruebas a las funciones del código que no requiere de configuraciones complejas. Su ejecución y sintaxis es bastante simple y al mismo tiempo poderosa. Ahora tendrás una ayuda extra para asegurar la calidad de las aplicaciones que generes.



Referencias bibliográficas

- Zaidman, V. (2018). *An overview of JavaScript Testing in 2018*. Recuperado de: <https://medium.com/welldone-software/an-overview-of-javascript-testing-in-2018-f68950900bc3>
- Joshi, V. (2020). *Jest como un marco de prueba de JavaScript popular*. Recuperado de: <https://cynoteck.com/es/blog-post/jest-as-a-popular-javascript-testing-framework/>
- Facebook Open Source. (2022). *Using Matchers*. Recuperado de: <https://jestjs.io/docs/using-matchers>

Para saber más

Lecturas

Para conocer más acerca de **pruebas**, te sugerimos leer lo siguiente:

- Facebook Open Source. (2022). *Documentación JEST*. Recuperado de <https://devdocs.io/jest/>
- Martinez, M. (2021). *Introducción al unit testing en JavaScript con JEST*. Recuperado de <https://40q.agency/introduccion-al-unit-testing-en-javascript-con-jest/>

Videos

Para conocer más acerca de **pruebas**, te sugerimos revisar lo siguiente:

- Raúl Palacios. (2021-07-18). *01 JEST Introducción* [Archivo de video]. Recuperado de <https://www.youtube.com/watch?v=IPAE1NOGsFM>
- Dev Done. (2022-03-11). *Jest VSCode Extension | How to code Tutorial* [Archivo de video]. Recuperado de <https://www.youtube.com/watch?v=zWYmDO5UzWQ>

Checkpoints

Asegúrate de:

- Conocer los diferentes tipos de pruebas para aplicar diferentes herramientas de software.
- Usar la herramienta JEST desde Visual Studio Code para realizar pruebas al código de JavaScript.
- Reconocer diferentes funciones matchers de JEST para aplicar la adecuada según el escenario de prueba.

Requerimientos técnicos

- Uso de la herramienta Terminal de Visual Studio Code
- Herramienta JEST
- Instalación de Node.JS
- Acceso a internet

Prework

- Deberás revisar el tema 8 Pruebas.
- Descarga Node.js desde la página oficial de Nodejs.org
- Instalar la extensión *JEST* en Visual Studio Code.

Tecmilenio no guarda relación alguna con las marcas mencionadas como ejemplo. Las marcas son propiedad de sus titulares conforme a la legislación aplicable, se utilizan con fines académicos y didácticos, por lo que no existen fines de lucro, relación publicitaria o de patrocinio.

La obra presentada es propiedad de ENSEÑANZA E INVESTIGACIÓN SUPERIOR A.C. (UNIVERSIDAD TECMILENIO), protegida por la Ley Federal de Derecho de Autor; la alteración o deformación de una obra, así como su reproducción, exhibición o ejecución pública sin el consentimiento de su autor y titular de los derechos correspondientes es constitutivo de un delito tipificado en la Ley Federal de Derechos de Autor, así como en las Leyes Internacionales de Derecho de Autor.

El uso de imágenes, fragmentos de videos, fragmentos de eventos culturales, programas y demás material que sea objeto de protección de los derechos de autor, es exclusivamente para fines educativos e informativos, y cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por UNIVERSIDAD TECMILENIO.

Queda prohibido copiar, reproducir, distribuir, publicar, transmitir, difundir, o en cualquier modo explotar cualquier parte de esta obra sin la autorización previa por escrito de UNIVERSIDAD TECMILENIO. Sin embargo, usted podrá bajar material a su computadora personal para uso exclusivamente personal o educacional y no comercial limitado a una copia por página. No se podrá remover o alterar de la copia ninguna leyenda de Derechos de Autor o la que manifieste la autoría del material.