



**GUIAS DE TESTING
PARA
JAVASCRIPT Y
REACT**

Índice

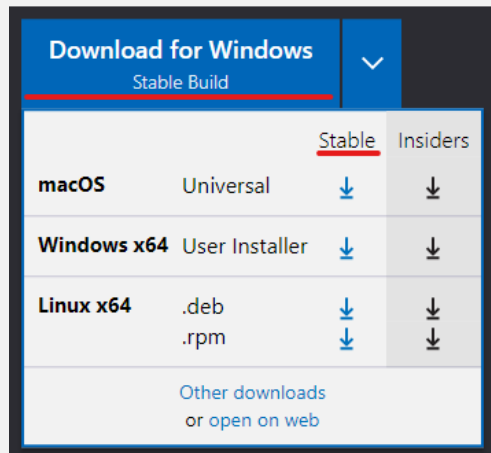
1. VSCode.....	2
2. Node JS.....	4
3. Git.....	4
4. Jest JS.....	6
5. Jest React.....	15
6. Cypress.....	25
7. Sinon JS.....	41
8. Chai JS.....	46

Visual Studio Code

Instalación

Esta guía se trabajó con VSCode debido a la versatilidad del editor y el poder trabajar con todas y cada una de las librerías de testing que vamos a probar a lo largo de esta guía, por lo que comenzaremos con la instalación de este.

1. Primero que nada, vamos a acceder a la página web de Visual Studio, donde seleccionaras tu sistema operativo: <https://code.visualstudio.com/>
























Puedes dar directamente en el botón azul que detecta de manera automática tu sistema operativo o en la pestaña seleccionar esto de manera manual, espera a que termine de descargar y lo instalas sin mayores complicaciones.

Extensiones

Una de las bondades de VSCode es que cuenta con un gran número de extensiones que nos ayudarán a ser más productivos o facilitarnos algunas tareas, ya sean oficiales de Microsoft o creadas por la comunidad de VSCode, aquí te dejo un listado de las extensiones que nos serán muy útiles tanto para las pruebas de testing y algunas otras cosas extras que te serán de utilidad:

- | | | | |
|----|---|----|---|
| 1. |  Auto Rename Tag
Auto rename paired HTML/XML tag
Jun Han | 4. |  Code Runner
Run C, C++, Java, JS, PHP, Python, Perl, Ruby, Go
Jun Han |
| 2. |  Babel JavaScript
VSCode syntax highlighting for today's JavaScript
Michael McDermott | 5. |  Cypress Snippets
Cypress snippets
Andrew Smith |
| 3. |  Chai snippets
Chai snippets in both assert and expect varieties
Nick Hatt | | |

6.  **Error Lens**
Improve highlighting of errors, warnings &...
Alexander
7.  **ES7+ React/Redux/React-Native snippets**
Extensions for React, React-Native and Redux in J...
dsznajder
8.  **HTML Boilerplate**
A basic HTML5 boilerplate snippet generator.
sidthesloth
9.  **HTML CSS Support**
CSS Intellisense for HTML
ecmel
10.  **HTML Snippets**
Full HTML tags including HTML5 Snippets
geyao
11.  **IntelliSense for CSS class names in HTML**
CSS class name completion for the HTML
Zignd
12.  **JavaScript (ES6) code snippets**
Code snippets for JavaScript in ES...
charalampos karypidis
13.  **JavaScript Snippet Pack**
A snippet pack to make you more produ...
Mahmoud Ali
14.  **Jest Runner**
Simple way to run or debug a single (or...
firsttris
15.  **Jest Snippets**
Code snippets for testing framework Jes...
andys8
16.  **Live Server**
Launch a development local Server with li...
Ritwick Dey
17.  **Lorem ipsum**
Generates and inserts lorem ipsum tes...
 **Daniel Imms**
18.  **Prettier - Code formatter**
Code formatter using prettier
 **Prettier**
19.  **Simple React Snippets**
Dead simple React snippets you will actually u...
 **Burke Holland**
20.  **Spanish Language Pack for Visual Studio Code**
Español
 **Microsoft**
21.  **Start git-bash**
Adds a bash command to VSCode that allows y...
McCarter
22.  **Icons**
Icons for Visual Studio Code.
Mhammed Talhaouy

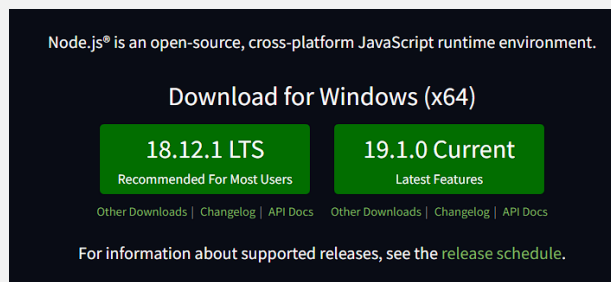
Solo debes ir a la pestaña de extensiones, buscar las extensiones fijándote que sean las mismas e instala, algunas requerirán que reinicies el editor, con esto terminado no deberías tener problemas con lo que estudiaremos a lo largo de esta guía.

Node JS

Instalación

Node JS es algo super importante a instalar ya que sin este no seremos capaces de instalar prácticamente nada de las librerías de testing que veremos más adelante, por lo que debemos instalarlo. Para ello iremos al siguiente enlace:

[Node.js \(nodejs.org\)](https://nodejs.org)

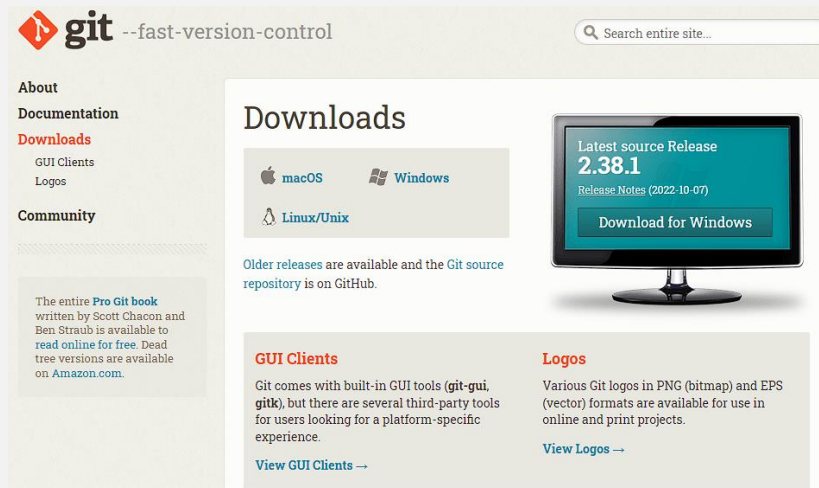


Nos dará a escoger la versión a instalar, selecciona la opción de la izquierda, la recomendada para la mayoría de los usuarios y después instala sin mayores complicaciones.

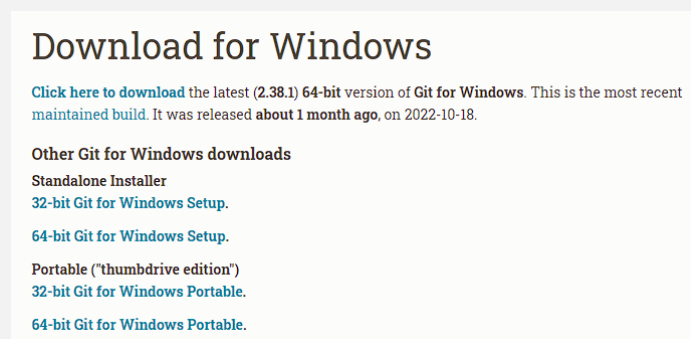
Git

Instalación

Nos será de mucha utilidad hacer uso de la terminal de git bash en nuestro VSCode para la instalación de las librerías de testing por lo que procederemos a instalarlo: **[Git - Downloads \(git-scm.com\)](https://git-scm.com)**



Estando en su página web seleccionaremos el apartado de Downloads y procede a escoger tu sistema operativo, en el caso de Windows nos puede salir lo siguiente:

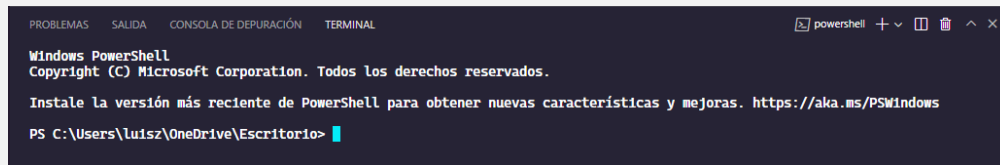


Descarga la versión instalable, si tienes dudas al respecto a la instalación ya que esta puede ser un poco más despistante que la de VSCode y Node JS, puedes seguir el siguiente video tutorial: [**Instalación de Git en Windows paso a paso | \[2021 | 2022\] - YouTube**](#)

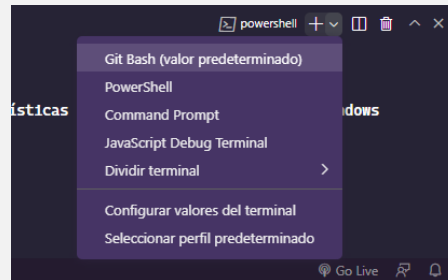
Finalmente, solo nos queda revisar que la terminal de git bash aparezca en nuestro VSCode, abre visual studio y selecciona el apartado de terminal:



Seleccionaremos la opción de nueva terminal (o podemos usar el comando `ctrl+ñ`) nos aparecerá una terminal de PowerShell la cual es la que esta por defecto para VSCode, podremos ver un botón con forma de signo + y junto a él una flecha en dirección hacia abajo:



Simplemente le damos clic y debería aparecer entre las opciones “Git Bash”:



De ser así por fin estamos listos para comenzar a trabajar con las librerías de testing, en caso contrario no te preocupes, busca algún video tutorial que te ayude a resolver el problema y poder agregar git bash a VSCode.

Jest JS

Introducción

Jest es un framework de pruebas de JavaScript que se utiliza para realizar pruebas de código de forma rápida y fácil. Fue desarrollado por Facebook y está diseñado para ser fácil de usar y configurar, lo que lo hace ideal para pruebas unitarias, de integración y de extremo a extremo.

Jest es compatible con una variedad de tecnologías de JavaScript, incluidas React, Vue, Angular y Node.js. Ofrece una amplia gama de funciones, como la ejecución de pruebas de forma paralela, la cobertura de código, la generación de informes detallados de pruebas y la simulación de eventos de usuario.

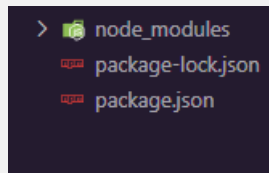
En resumen, Jest es una herramienta poderosa para los desarrolladores que desean realizar pruebas de código de manera rápida y sencilla, lo que les permite garantizar que su código esté funcionando correctamente y cumpliendo con los requisitos de calidad antes de ser implementado en producción.

Instalación y primeras pruebas

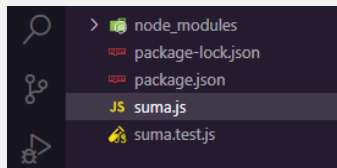
Crea una carpeta con el nombre que desees y ábrela en VSCode, abre una nueva terminal, de preferencia git bash, y agrega el siguiente comando:

npm install --save-dev jest

Una vez que finalice la descarga e instalación del framework veras que aparecen 3 archivos nuevos en nuestra carpeta:



Con esto listo podremos comenzar a trabajar con Jest. Comenzaremos haciendo pruebas sencillas a una función de operaciones básicas, para ello vamos a crear 2 archivos nuevos, **suma.js** y **suma.test.js**:



Dentro de suma.js vamos a agregar el siguiente código:

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

Mientras que en suma.test.js agregamos lo siguiente:

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

Vemos que se manda a llamar la función suma, para iniciar las pruebas se escribe como se ve en el código como si se tratase de una función flecha:

test ('nombre del test', () => {función})

En el ejemplo usamos expect y toBe, en resumen, le estamos indicando que esperamos que la suma de los valores 1 y 2 nos regrese un 3 como resultado (para guardar los archivos y no tener problemas con las pruebas usa el comando *ctrl+s*).

Si instalamos correctamente nuestra extensión de Jest runner se debería de ver un pequeño botón de run y debug sobre nuestra función:

```
Run | Debug
3 test('sumar 1 + 2 es igual a 3', () => {
4   expect(suma(1, 2)).toBe(3);
5 });
```

Daremos clic en **run** para que se corra la prueba en la consola y debería de verse algo como lo siguiente:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

luisz@HP-Luis MINGW64 ~/Downloads/Documentos-Alter/Borrar
$ cd "c:\Users\Luisz\Downloads\Documentos-Alter/Borrar"

luisz@HP-Luis MINGW64 ~/Downloads/Documentos-Alter/Borrar
$ node "node_modules/jest/bin/jest.js" "c:/Users/Luisz/Downloads/Documentos-Alter/Borrar/suma.test.js" -t "sumar 1 + 2 es igual a 3"
PASS ./suma.test.js
  sumar 1 + 2 es igual a 3 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.895 s, estimated 1 s
Ran all test suites matching /c:\Users\Luisz\Downloads\Documentos-Alter\Borrar\suma.test.js/ with test
s matching "sumar 1 + 2 es igual a 3".

luisz@HP-Luis MINGW64 ~/Downloads/Documentos-Alter/Borrar
$
```

Vemos que marca como PASS/passed la prueba. También podemos correr la prueba desde consola, para ello entra al archivo de nombre package.json y agrega lo siguiente (no modifies lo que ya este agregado):

```
JS suma.js suma.test.js package.json x
package.json > ...
1 {
2   "devDependencies": {
3     "jest": "^29.3.1"
4   },
5   "scripts": {
6     "test": "jest"
7   }
8 }
```

Ahora abre una nueva consola y coloca el siguiente comando:

npm test

Deberías ver como se vuelve a correr la prueba replicando el resultado antes visto, también habrás notado que apareció un pequeño botón de nombre “depurar”, si lo presionas lo que hará es que se compilaren todas las pruebas que tengamos en ese momento, puedes hacer la prueba haciendo clic en dicho botón, pero de momento no le tomaremos demasiada importancia.

Ahora las siguientes pruebas que haremos requerirán de que instalemos Babel que nos ofrece una sintaxis para JS más moderna, para ello en la terminal agregaremos el siguiente comando:

npm install --save-dev babel-jest @babel/core @babel/preset-env

Una vez finalizada la instalación crearemos un archivo de nombre “.babelrc” y le agregaremos el siguiente código:



Guarda el archivo y crearemos un nuevo archivo de nombre index.js donde colocaremos el siguiente código:

```
export const calculadora = {
  sum(a, b) {
    return a + b;
  },
  rest(a, b) {
    return a - b;
  },
  multi(a, b) {
    return a * b;
  },
  div(a, b) {
    return a / b;
  }
}
```

Mientras que en suma.test.js comentaremos lo que teníamos antes y agregaremos lo siguiente:

```
import { calculadora } from ".";

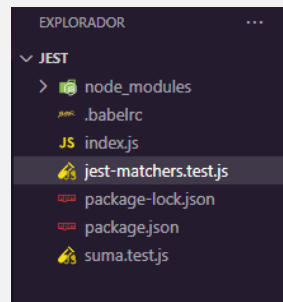
test('sum calculator', () => {
  const result = calculadora.sum(1, 2);
  expect(result).toBe(3);
});
```

Como vemos podemos enviar el objeto completo y simplemente dentro de la prueba podemos llamar a cualquiera de las funciones que nos interese realizarles pruebas, como se ve en el código que estamos almacenando en una variable de nombre “result” lo que nos retorne el mandar a la función de suma los valores de 1 y 2, para que finalmente le digamos que esperamos que el valor de “result” sea 3.

Véase también que al momento de importar nuestro objeto con las funciones colocamos que venga de ".", es porque cuando tenemos de nombre nuestro archivo como index y está en la misma carpeta que nuestro test podemos simplemente colocar el punto, ya que JavaScript/Jest sabrá que no estamos refiriendo a dicho archivo index.

Matchers

Con Jest podemos hacer uso de algo llamado matchers, en realidad ya vimos uno, el "toBe", pero hay muchos más que ese, donde podemos comprobar veracidades, números, cadenas de texto, vectores e iteraciones, excepciones, etc. Para poder ver un poco como funcionan estos vamos a crear un nuevo archivo al cual le pondremos por nombre **jest-matchers.test.js**:



Ahora dentro de este vamos a probar varios de estos matchers, empezando por ver que no necesariamente debemos pasar una función para poder probarlos:

```
test('asignación de un objeto', () => {  
  const datos = { uno: 1 };  
  datos['dos'] = 2;  
  expect(datos).toEqual({ uno: 1, dos: 2 });  
});
```

Da clic en el botón de run para verificar que la prueba paso con éxito. Bueno empezaremos a ver un listado de algunos tipos de matchers

Veracidad

```
test('null', () => {  
  const n = null;  
  expect(n).toBeNull(); // espera que "n" sea igual a null  
  expect(n).toBeDefined(); // espera que "n" este definida  
  expect(n).not.toBeUndefined(); // espera que "n" no sea undefined  
  expect(n).not.toBeTruthy(); // espera que "n" no sea true  
  expect(n).toBeFalsy(); // espera que "n" no sea false  
});
```

```
test('cero', () => {
  const z = 0;
  expect(z).not.toBeNull(); // que "z" no sea null
  expect(z).toBeDefined(); // que "z" este definida
  expect(z).not.toBeUndefined(); // que "z" no sea undefined
  expect(z).not.toBeTruthy(); // que "z" no sea true
  expect(z).toBeFalsy(); // que "z" sea false (0 se considera false)
});
```

Números

```
test('dos mas dos', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3); // que "value" sea mayor que...
  expect(value).toBeGreaterThanOrEqual(3.5); // que "value" sea
mayor o igual que...
  expect(value).toBeLessThan(5); // que "value" sea menor que...
  expect(value).toBeLessThanOrEqual(4.5); // que "value" sea menor o
igual que...

  // toBe y toEqual son equivalentes para números
  expect(value).toBe(4);
  expect(value).toEqual(4);
});

test('agregando números de punto flotante', () => {
  const value = 0.1 + 0.2;
  //expect(value).toBe(0.3); Esto no funcionará debido al error de
redondeo
  expect(value).toBeCloseTo(0.3); // Esto funciona.
});
```

Cadenas de texto:

```
test('no hay I en Team', () => {
  expect('team').not.toMatch(/I/); // la cadena no haga "match" con
la expresión regular
});

test('hay "stop" en Christoph', () => {
  expect('Christoph').toMatch(/stop/); // la cadena haga "match"
con la expresión regular
});

//también podemos hacer uso del toBe con cadenas:
test('string', () => {
  expect('team').toBe('team');
});
```

Vectores e iterables:

```
const listaDeCompras = [
  'pañales',
```

```

    'pañuelos',
    'bolsas de basura',
    'toallas de papel',
    'leche',
  ];

test('la leche se encuentra en la lista de compras', () => {
  expect(listaDeCompras).toContain('leche'); // espera que el
  arreglo contenga el elemento 'leche'
  expect(new Set(listaDeCompras)).toContain('leche'); // lo mismo
  con set
});

```

Excepciones

```

function compilarCódigoAndroid() {
  throw new ConfigError('Usted está usando el código incorrecto');
}

test('La compilacion de android va por buen camino', () => {
  expect(() => compilarCódigoAndroid()).toThrow();
  expect(() => compilarCódigoAndroid()).toThrow(ConfigError);

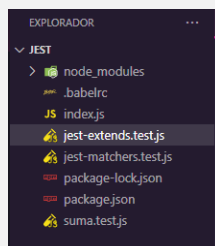
  // Puede usar también el mensaje de error exacto o una expresión
  regular
  expect(() => compilarCódigoAndroid()).toThrow('Usted está usando
  el código incorrecto');
  expect(() => compilarCódigoAndroid()).toThrow(/JDK/);
});

```

Como puedes notar, hay una gran variedad de estos, pero son fáciles de entender, ya que son muy literales al momento de leerlos. Pero estos solo son unos cuantos, si quieres ver la lista completa, revisa el siguiente enlace: <https://jestjs.io/es-ES/docs/expect>

Matchers Personalizados

Otra de las cosas que podemos realizar con Jest es crear nuestros propios matchers, esto nos servirá para cuando necesitemos hacer algún tipo de prueba específica, pero que a la vez sepamos que vamos a realizar más de una vez. Haremos un ejemplo simple de esto, para ello vamos a crear un archivo nuevo llamado "jest-extends.test.js" quedando de la siguiente manera:



Procederemos a colocar el siguiente código:

```
expect.extend({
  toBeEqualTwo(received) {

    if (received !== 2) { // si el valor recibido no es 2
      return {
        pass: false, // pasamos un falso
        message: () => `Expected ${received} to be number 2`
      }; // mensaje de error personalizado
    }

    return {
      pass: true // simplemente pasamos true en caso de
    }; // cumplirse el test
  }
});
```

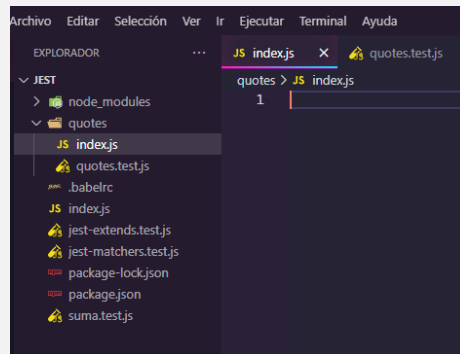
Para indicar que estamos haciendo un matcher personalizado empezamos colocando "expect.extend", seguido el nombre de nuestro matcher que en este caso es "toBeEqualTwo" y finalmente la lógica de nuestro matcher. Debajo de dicho código colocaremos lo siguiente:

```
test("number 2", () => {
  expect(2).toBeEqualTwo(); // llamaremos a nuestro matcher
  // personalizado como toBeEqualTwo()
});
```

Que es donde simplemente lo probamos y nos debería mandar el mensaje de PASSED en consola. La comunidad ha creado y compartido múltiples matchers personalizados, por lo que aquí tienes un enlace a estos: <https://github.com/jest-community/jest-extended>

Tests asíncronos

Terminaremos esta parte de la guía de Jest viendo cómo podemos hacer test a funciones asíncronas con promesas, ya que son algo con lo que nos podemos topar, para ello vamos a crear una carpeta llamada "quotes" y dentro de esta tendremos 2 archivos de nombres: "index.js" y "quotes.test.js":



Dentro de index vamos a agregar la siguiente función que tendrá una promesa y un temporizador:

```
export const getQuote = () => {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve("No creas todo lo que lees en  
internet, por Benjamin Franklin"), 1000;  
  })  
}
```

Pasando al archivo test si intentamos realizar la prueba como lo hemos venido haciendo desde el principio de esta guía nos dará un error, ya que no estaremos resolviendo la promesa como se ve en este ejemplo:

```
import { getQuote } from ".";  
  
test('getQuote', () => {  
  const quote = getQuote();  
  
  expect(quote).toBe(  
    "No creas todo lo que lees en internet, por Benjamin Franklin"  
  );  
});
```

Si corremos esa prueba nos mandara error, para evitar eso debemos hacer lo siguiente:

```
import { getQuote } from ".";  
  
test('getQuote', async () => {  
  const quote = await getQuote();  
  
  expect(quote).toBe("No creas todo lo que lees en internet, por  
Benjamin Franklin");  
});
```

Volvemos asíncrona nuestra prueba haciendo uso del `async await` para resolver la promesa y con esto al momento de correr la prueba esta será exitosa. Si quieres profundizar más en Jest JS revisa más a fondo su documentación [Getting Started · Jest \(jestjs.io\)](#).

Jest React

Introducción

Jest es una herramienta muy popular para realizar pruebas de React. Con Jest, puedes realizar pruebas unitarias y de integración para tus componentes de React, asegurándote de que estén funcionando como se espera y cumpliendo con los requisitos de calidad.

Una de las características clave de Jest es su integración con React. Jest viene con un conjunto de utilidades predefinidas que te permiten realizar pruebas para componentes de React, y también es compatible con otras bibliotecas comunes de React.

Jest te permite realizar pruebas en tiempo real, lo que significa que puedes ver los resultados de las pruebas a medida que las escribes. También te permite generar informes detallados de las pruebas y ofrece una amplia gama de opciones de configuración.

En resumen, Jest es una herramienta muy útil para realizar pruebas de React, y es ampliamente utilizada por los desarrolladores de React para garantizar que sus componentes funcionen correctamente y cumplan con los requisitos de calidad.

Instalación

Al ser un proyecto de react vamos a necesitar hacer unas cuantas instalaciones extras, empieza por crear una nueva carpeta para tu proyecto, abre dicha carpeta en VSCode, abre una nueva terminal y coloca el siguiente comando:

`npm create vite@latest`

Como estamos haciendo uso de Vite se nos irán solicitando seleccionar ciertas opciones a lo largo de la instalación, en principio se te solicitara que escribas el nombre del proyecto, seguido de esto te solicitara seleccionar el framework sobre el cual se trabajara, selecciona React y finalmente te pedirá el lenguaje base para eso, selecciona JavaScript:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  JUPYTER

Need to install the following packages:
  create-vite@4.0.0
Ok to proceed? (y) y
✓ Project name: ... jest-react
? Select a framework: » - Use arrow-keys. Return to submit.
>  Vanilla
  Vue
  React
  Preact
  Lit
  Svelte
  Others
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  JUPYTER

Need to install the following packages:
  create-vite@4.0.0
Ok to proceed? (y) y
✓ Project name: ... jest-react
✓ Select a framework: » React
? Select a variant: » - Use arrow-keys. Return to submit.
>  JavaScript
  TypeScript
  JavaScript + SWC
  TypeScript + SWC
```

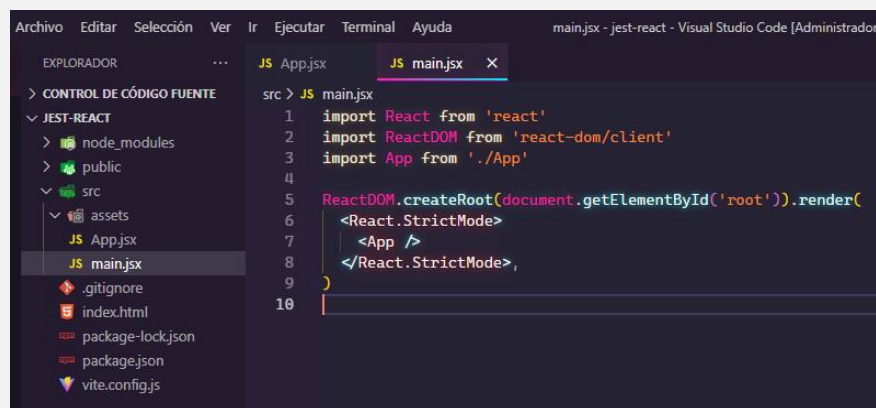
Una vez terminado este proceso entramos a la carpeta que se nos acaba de crear y una vez dentro en consola aplicamos el comando:

npm install

Para poder correr el proyecto y verlo en el navegador hacemos uso del comando:

npm run dev

Esto nos creará un servidor local y se nos dará el enlace en la consola para poder abrirlo en el navegador. Por mientras borraremos los archivos `app.css` e `index.css`, así como en el archivo `main.jsx` borraremos la línea que importa el `index.css`:

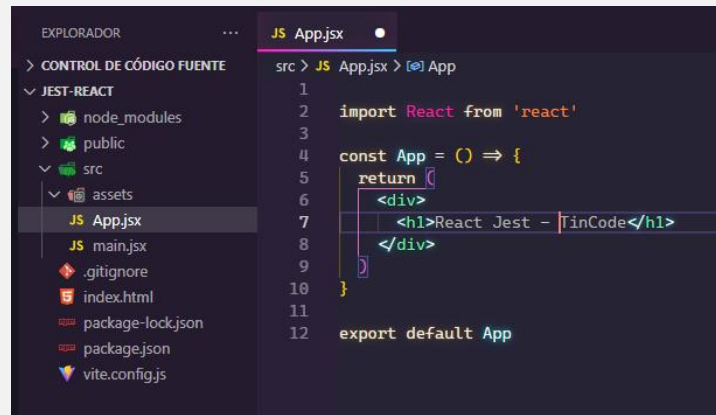


```
Archivo  Editar  Selección  Ver  Ir  Ejecutar  Terminal  Ayuda  main.jsx - jest-react - Visual Studio Code [Administrador]

EXPLORADOR  ...
> CONTROL DE CÓDIGO FUENTE
  JEST-REACT
  > node_modules
  > public
  > src
    > assets
    JS App.jsx
    JS main.jsx
    .gitignore
    index.html
    package-lock.json
    package.json
    vite.config.js

src > JS main.jsx
1  import React from 'react'
2  import ReactDOM from 'react-dom/client'
3  import App from './App'
4
5  ReactDOM.createRoot(document.getElementById('root')).render(
6    <React.StrictMode>
7      <App />
8    </React.StrictMode>,
9  )
10
```

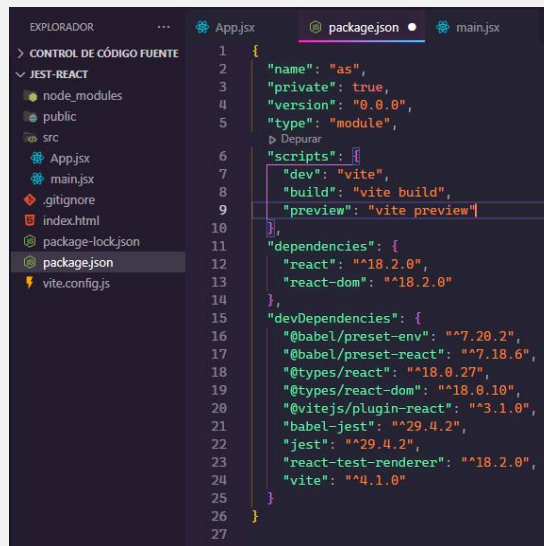
Ahora dentro de `App.jsx` borraremos todo lo que contenga y empezaremos desde 0 con el siguiente código:



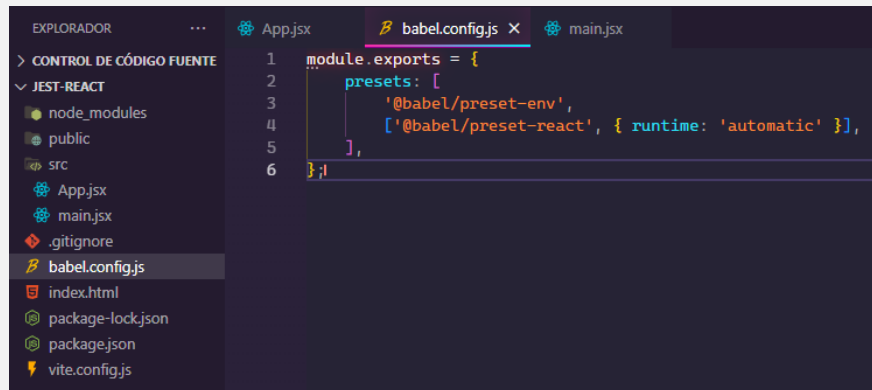
Abrimos la terminal y colocaremos el comando que se nos proporciona dentro de la documentación de Jest el cual es el siguiente:

npm install --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-renderer

Una vez finalizada la instalación procede a abrir el archivo “package.json”:



Para terminar, crearemos un archivo en la carpeta raíz de nombre babel.config.js y le colocaremos el siguiente código:



```
1 module.exports = {  
2   presets: [  
3     '@babel/preset-env',  
4     ['@babel/preset-react', { runtime: 'automatic' }]],  
5   ],  
6 };
```

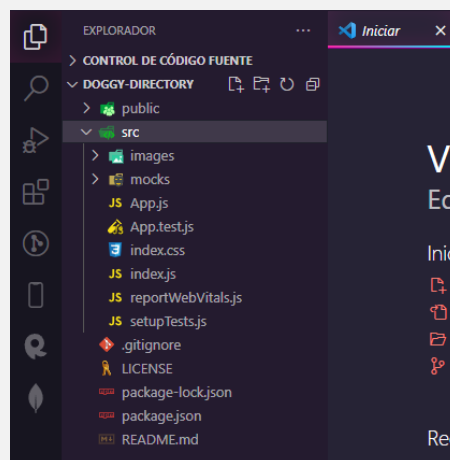
Con esto ya podrás comenzar a hacer pruebas a tus proyectos de React con Jest, pero por motivos de que quede más claro el cómo se hace uso de Jest con React vamos a hacer uso de una web ya creada para dichos fines.

Pruebas Jest en React en Doggy Directory

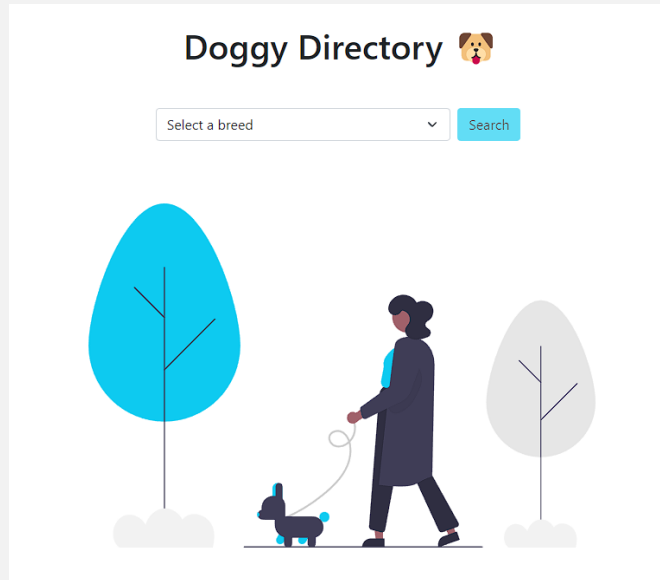
Crea una carpeta nueva, ábrela en VSCode, abre una nueva terminal y coloca el siguiente comando:

git clone <https://github.com/do-community/doggy-directory>

Esto creará una copia de todos los archivos de dicho repositorio en nuestra carpeta. Cuando termine la descarga nos aparecerá una carpeta de nombre “doggy-directory” entra a dicha carpeta ya sea con el comando de cd desde la terminal o abriéndola directamente con el explorador de archivos de VSCode (recomiendo lo segundo).



Ahora vuelve a colocar en la terminal el comando de **npm install** y una vez finalice la descarga podremos empezar a trabajar. Cuando termine la instalación de npm abre el proyecto en el navegador con el comando **npm start**, deberíamos ver la siguiente aplicación en el navegador:



Ahora necesitamos poner a Jest en modo Watch, ósea que cada vez que hagamos un cambio en nuestro código y lo guardemos, se ejecutará en automático la prueba, para ello simplemente solo necesitamos colocar en consola el comando **npm test** y deberá aparecer lo siguiente:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  GITLENS  JUPYTER
No tests found related to files changed since last commit.
Press 'a' to run all tests, or run Jest with '--watchAll'.

Watch Usage
> Press a to run all tests.
> Press f to run only failed tests.
> Press q to quit watch mode.
> Press p to filter by a filename regex pattern.
> Press t to filter by a test name regex pattern.
> Press Enter to trigger a test run.

```

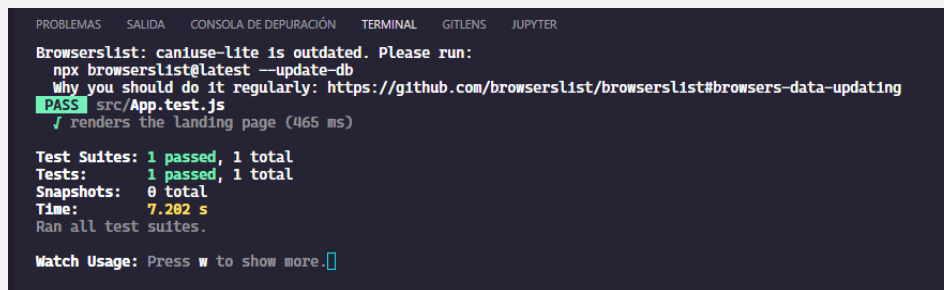
Es un menú para ciertas funcionalidades de la prueba para Jest, bastante descriptivas.

Ahora dentro de la carpeta **src** se encuentra un archivo de nombre **App.test.js**, dentro de esta hay un pequeño código que solo verifica que el componente `<App/>` se renderice correctamente, pero vamos a eliminar dicha prueba y colocaremos el siguiente código (no borres los imports):

```
test("renders the landing page", () => {
  render(<App />);
  expect(screen.getByRole("heading")).toHaveTextContent(/Doggy
Directory/);
  expect(screen.getByRole("combobox")).toHaveDisplayValue("Select a
breed");
  expect(screen.getByRole("button", { name: "Search"
})).toBeDisabled();
  expect(screen.getByRole("img")).toBeInTheDocument();
});
```

Si te das cuenta es muy descriptivo el código con los test que se quieren realizar, siendo el primero que revise que el elemento que tiene el rol de **“heading”** tenga el texto **“Doggy Directory”**, que el valor mostrado en el select sea **“Select a breed”**, que el boton de nombre **“Search”** este deshabilitado y finalmente que el elemento con el rol **“img”** este presente en el documento (para que sea más claro lo de los roles y nombres de los elementos inspeccionar la página y revisar los elementos de esta).

Una vez revisado lo anterior guardamos y abrimos la terminal, como anteriormente usamos el comando `npm test` este quedo en modo de watch, esto significa que con solo guardar el archivo se realizara el test de manera automática, por lo que debemos visualizar lo siguiente:



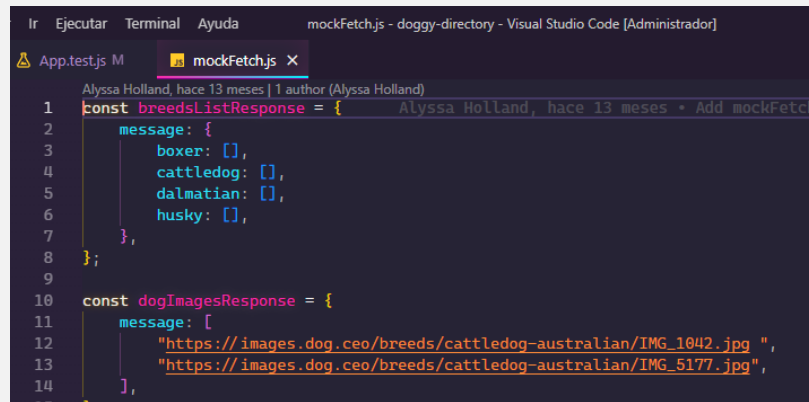
```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  GITLENS  JUPYTER
Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating
PASS src/App.test.js
  ✓ renders the landing page (465 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.262 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Test de Fetch

Como sabemos muchas veces se necesitan hacer peticiones fetch a apis o bdd por lo que debemos ser capaces de realizar tests para ello, para esto nos iremos a la carpeta de mocks y abriremos el archivo que contiene `“mockFetch.js”`:



```
1 const breedsListResponse = {
2   message: {
3     boxer: [],
4     cattledog: [],
5     dalmatian: [],
6     husky: [],
7   },
8 };
9
10 const dogImagesResponse = {
11   message: [
12     "https://images.dog.ceo/breeds/cattledog-australian/IMG_1042.jpg",
13     "https://images.dog.ceo/breeds/cattledog-australian/IMG_5177.jpg",
14   ],
15 }
```

Volviendo al archivo App.test.js vamos a agregar un nuevo código debajo de la anterior prueba, el cual es el siguiente (escríbelo manualmente para que se hagan las importaciones necesarias de manera automática):

```
beforeEach(() => {
  jest.spyOn(window, "fetch").mockImplementation(mockFetch);
})

afterEach(() => {
  jest.resetAllMocks();
})
```

En la parte de los imports debemos agregar lo siguiente:

```
import mockFetch from "../mocks/mockFetch";
```

Antes de continuar una breve explicación de todo lo anterior, el método mockFetch devuelve un objeto que se parece mucho a la estructura de lo que devolvería una llamada de búsqueda en respuesta a las llamadas API dentro de la aplicación. El método mockFetch es necesario para probar la funcionalidad asíncrona en dos áreas de la aplicación Doggy Directory: el menú desplegable de selección que completa la lista de razas y la llamada API para recuperar imágenes de perros cuando se realiza una búsqueda.

Debes saber que `jest.spyOn(window, "fetch");` lo que hace es que crea una función simulada que rastreará las llamadas al método de búsqueda adjunto a la variable de ventana global en el DOM. Mientras que `.mockImplementation(mockFetch);` acepta una función que se utilizará para implementar el método simulado. Debido a que este comando anula la implementación original de búsqueda, se ejecutará cada vez que se llame a búsqueda dentro del código de la aplicación.

Guardamos el archivo y revisamos que en la consola nos manda el siguiente error:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  GITLENS  JUPYTER

This ensures that you're testing the behavior the user would see in the browser. Learn more at https://reactjs.org/link/wrap-tests-with-act
    at App (C:\Users\Luisz\OneDrive\Documentos\Escuela\Servicio Social\Guía de Tests\Jest React Inst\Jest-React-Doggy\doggy-directory\src\App.js:5:31)

18 |   |
19 |   |   .then((json) => {
> 20 |     |     setBreeds(Object.keys(json.message));
    |     |     ^
21 |   |   });
22 |   |   }, []);
23 |   |

at printWarning (node_modules/react-dom/cjs/react-dom.development.js:67:30)
at error (node_modules/react-dom/cjs/react-dom.development.js:43:5)
at warnIfNotCurrentlyActingUpdatesInDEV (node_modules/react-dom/cjs/react-dom.development.js:24864:9)
at setBreeds (node_modules/react-dom/cjs/react-dom.development.js:16135:9)
at src/App.js:28:9

You, hace 15 minutos  Lín. 18, col. 24  Espacios: 2  UTF-8  CRLF  {} Babel JavaScript
```

La advertencia le indica que se produjo una actualización de estado cuando no se esperaba. Sin embargo, el resultado también indica que las pruebas han simulado con éxito el método de obtención.

En este paso, se burló del método de búsqueda e incorporó ese método en un conjunto de pruebas. Aunque la prueba está pasando, aún debe abordar la advertencia.

La advertencia de acto se produce porque se ha burlado del método de obtención y, cuando se monta el componente, realiza una llamada a la API para obtener la lista de razas. La lista de razas se almacena en una variable de estado que completa el elemento de opción dentro de la entrada de selección.

Podemos ver la lista de las razas con solo dar clic en el select que hay en la página. La advertencia se lanza porque el estado se establece después de que el bloque de prueba termine de representar el componente.

Vamos a arreglar dicho error, para ello vamos a App.test.js y modificaremos un poco el bloque de prueba que teníamos anteriormente de la siguiente manera:

```
test("renders the landing page", async () => {
  render(<App />);
  expect(screen.getByRole("heading")).toHaveTextContent(/Doggy
Directory/);
  expect(screen.getByRole("combobox")).toHaveDisplayValue("Select a
breed");
  expect(await screen.findByRole("option", { name: "husky"
})).toBeInTheDocument();
  expect(screen.getByRole("button", { name: "Search"
})).toBeDisabled();
  expect(screen.getByRole("img")).toBeInTheDocument();
});
```

Volvimos la función de la prueba asíncrona y agregamos un await. La palabra clave async le dice a Jest que el código asíncrono se ejecuta como resultado de la llamada a la API que se produce cuando se monta el componente.

Una nueva aserción con la consulta `findBy` verifica que el documento contiene una opción con el valor de `husky`. Las consultas `findBy` se usan cuando necesita probar código asíncrono que depende de que algo esté en el DOM después de un período de tiempo. Debido a que la consulta `findBy` devuelve una promesa que se resuelve cuando el elemento solicitado se encuentra en el DOM, la palabra clave `await` se usa dentro del método `expect`.

Guardamos el archivo y revisamos que en consola ya no hay ningún tipo de error en la prueba.

Ahora vamos a hacer tests sobre la función de búsqueda de la aplicación, para esto vamos a agregar un par de importaciones dentro del archivo de `App.test.js`:

```
import { render, screen, waitForElementToBeRemoved } from "@testing-library/react";
import userEvent from '@testing-library/user-event';
```

Luego de esto crearemos un nuevo bloque de tests después de todo el código que hemos hecho con anterioridad:

```
test("should be able to search and display dog image results", async
() => {
  render(<App />);
})
```

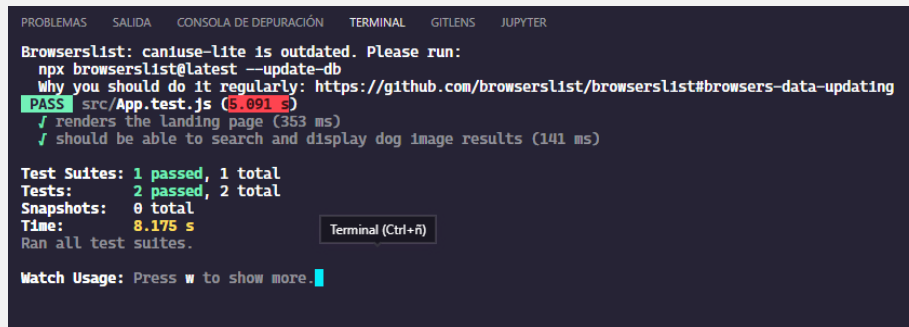
Ahora dentro de este escribiremos una nueva serie de tests quedando de la siguiente manera:

```
test("should be able to search and display dog image results", async
() => {
  render(<App />);

  //Simular la selección de una opción y verificar su valor
  const select = screen.getByRole("combobox");
  expect(await screen.findByRole("option", { name:
"cattledog"})).toBeInTheDocument();
  userEvent.selectOptions(select, "cattledog");
  expect(select).toHaveValue("cattledog");
})
```

Como se indica en el texto comentado estamos simulando la selección de una opción del select, siendo primero que estamos almacenando en una variable de nombre **“select”** dicho elemento, seguido de esto verificamos que la opción **“cattledog”** este presente en dicho select, hacemos uso de un `userEvent` para que simule que el usuario dio click en dicha opción y finalmente verificamos que

nuestro **“select”** tenga el valor de **“cattledog”**. Guardamos el archivo y revisamos que la terminal muestre lo siguiente:



```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  GITLENS  JUPYTER

Browserslist: caniuse-lite is outdated. Please run:
  npx browserslist@latest --update-db
  Why you should do it regularly: https://github.com/browserslist/browserslist#browsers-data-updating
PASS src/App.test.js (5.091 s)
  ✓ renders the landing page (353 ms)
  ✓ should be able to search and display dog image results (141 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        8.175 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Ahora agregaremos más líneas de código a nuestro test siendo las siguientes:

```
//Simular el inicio de la solicitud de búsqueda
const searchBtn = screen.getByRole("button", { name: "Search" });
expect(searchBtn).not.toBeDisabled();
userEvent.click(searchBtn);

//El estado de carga se muestra y se elimina una vez que se
muestran los resultados
await waitForElementToBeRemoved(() =>
screen.queryByText(/Loading/i));
```

En resumen, ahora almacenamos en una variable “searchBtn” el botón de búsqueda y seguido de esto verificamos que no esté deshabilitado y haciendo uso del userEvent realizamos un clic en este. La búsqueda debería hacer que aparezca un elemento de carga y posteriormente sea removido al finalizar dicha carga, como lo indica de manera muy explícita la función de waitForElementToBeRemoved. Guardamos el archivo y revisamos que en la consola indique que todos los test fueron realizados con éxito.

Ahora haremos una prueba sobre las imágenes que resultan de la búsqueda, con lo cual haremos agregando el siguiente código:

```
//Verifica el despliegado de las imágenes y el conteo de estas
const dogImages = screen.getAllByRole("img");
expect(dogImages).toHaveLength(2);
expect(screen.getByText(/2 Results/i)).toBeInTheDocument();
expect(dogImages[0]).toHaveAccessibleName("cattledog 1 of 2");
expect(dogImages[1]).toHaveAccessibleName("cattledog 2 of 2");
```

Después de escribir lo anterior guardamos el archivo y vemos que en la terminal la prueba paso con éxito. La consulta getAllByRole seleccionará todas las imágenes de perros y las asignará a la variable dogImages. La variante “AllBy” de la

consulta devuelve una matriz que contiene varios elementos que coinciden con el rol especificado. La variante "AllBy" difiere de la variante ByRole, que solo puede devolver un único elemento.

La implementación de búsqueda simulada contenía dos URL de imagen dentro de la respuesta. Con el comparador toHaveLength de Jest, puede verificar que se muestran dos imágenes.

La consulta getByText comprobará que el recuento de resultados adecuado aparece en la esquina derecha.

Dos aserciones que usan los comparadores toHaveAccessibleName verifican que el texto alternativo apropiado esté asociado con imágenes individuales.

Con esto terminamos de ver de manera introductoria el cómo se puede usar Jest para hacer test del DOM en react, si quieres profundizar más en esto revisa la documentación de Jest para React: [Testing React Apps · Jest \(jestjs.io\)](https://jestjs.io/docs/react)

Cypress

Introducción

Cypress es una herramienta de automatización de pruebas de extremo a extremo para aplicaciones web modernas. Es de código abierto y se ejecuta en un navegador, lo que significa que puede interactuar con su aplicación como lo hace un usuario, pero de manera automatizada.

Cypress es conocido por su arquitectura única y su facilidad de uso. Permite escribir pruebas en JavaScript, lo que lo hace accesible para aquellos con conocimientos básicos de programación. Además, proporciona una amplia gama de funciones integradas que simplifican el proceso de escribir, ejecutar y depurar pruebas.

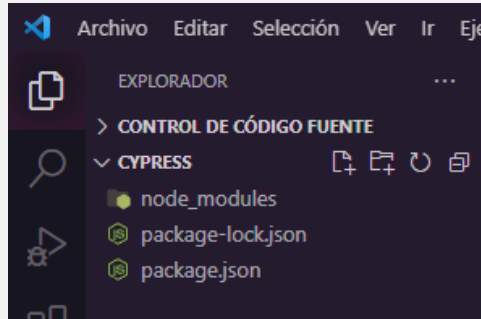
Cypress también es conocido por su capacidad para ejecutar pruebas de manera rápida y confiable, lo que lo hace ideal para la integración y la entrega continua de aplicaciones web. En general, Cypress es una herramienta poderosa para mejorar la calidad y la eficiencia del proceso de desarrollo de aplicaciones web.

Instalación

Para esto vamos a crear una nueva carpeta donde almacenaremos nuestro proyecto, ponle el nombre que gustes y dentro de esta abriremos nuestro VSCode y comenzaremos abriendo la terminal y escribiendo el comando:

npm i cypress --save-dev

Una vez finalizada la instalación de los archivos veremos que se agregaron los siguientes archivos:



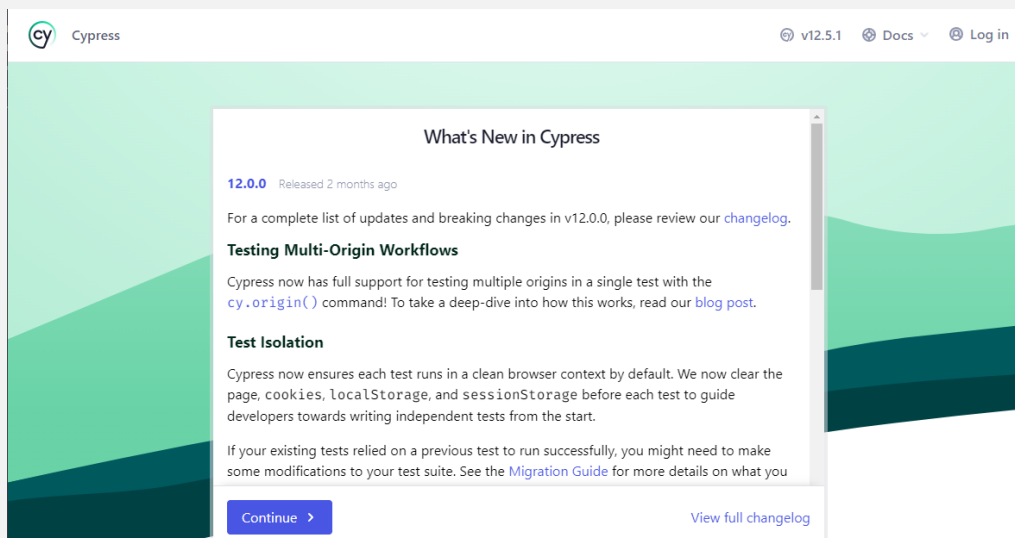
Lo cual indica que ya estamos más que listos para empezar a trabajar con Cypress.

Primeros pasos

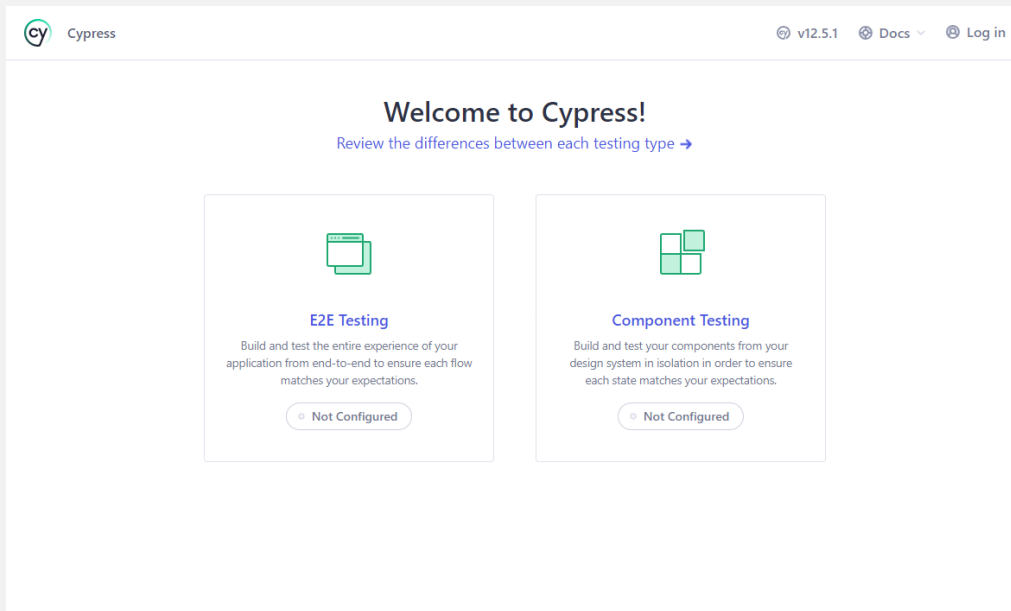
Para comenzar a hacer uso de este primero deberemos abrirlo, para ello en la consola colocaremos el siguiente comando:

npm cypress open

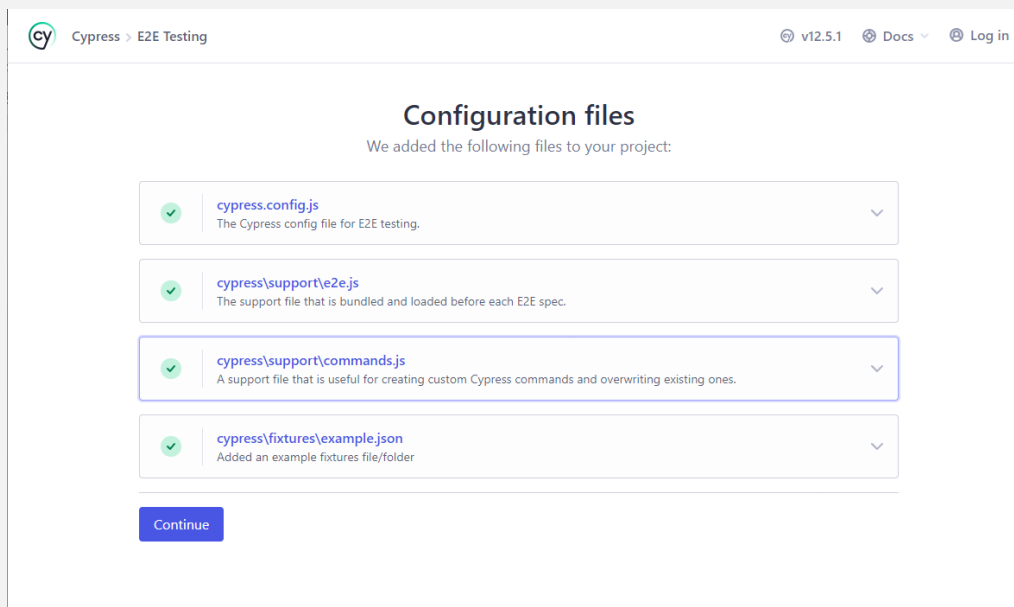
Puede que no salga esta ventana, así que solo hay que dar en continuar:



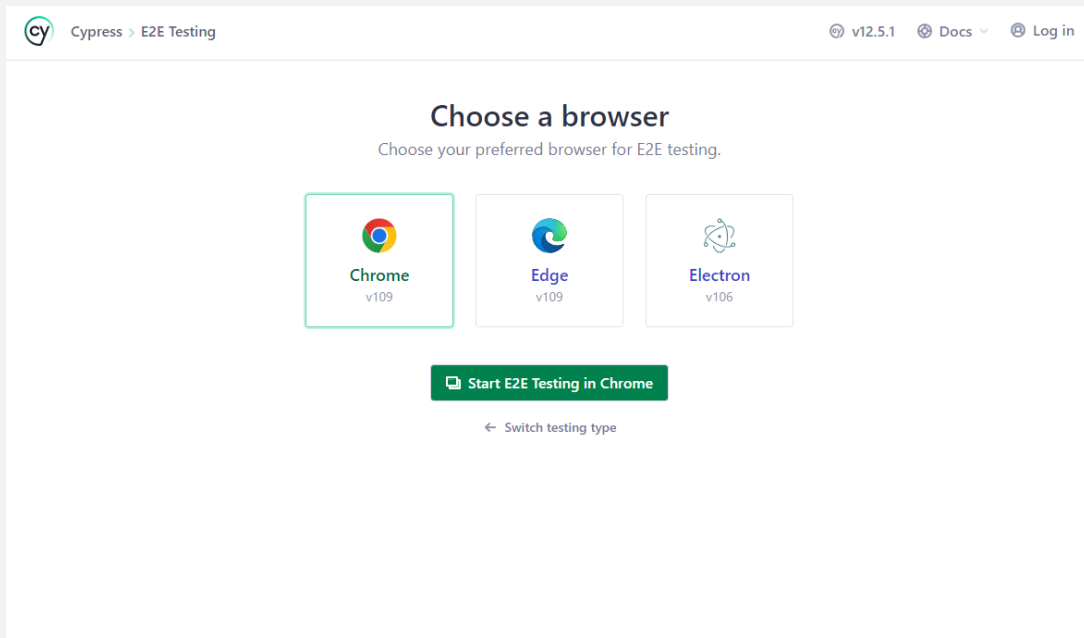
A lo que nos llevara a lo importante:



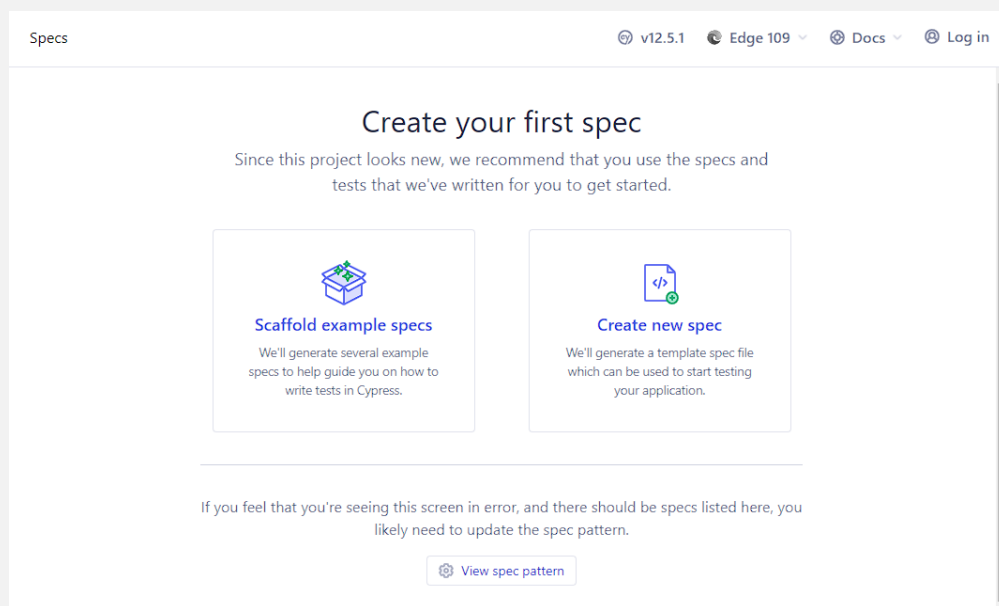
Como puedes ver tiene 2 opciones principales, lo que es el E2E Testing y el Component Testing, de momento nos enfocaremos primero en el E2E Testing, daremos clic en dicha opción y nos abrirá una ventana de configuración para este:



En este caso no necesitamos mover nada así que solo le damos en continuar y ahora nos saldrá una ventana que nos dará a escoger el navegador en el cual se simularan los test:



Selecciona el de tu preferencia y presiona el botón verde de Start esto abrirá un “navegador” de pruebas de Cypress:



Vamos a comenzar seleccionando la opción de Create new spec y nos aparecerá la siguiente ventana, solo debemos presionar el botón Create spec:

Enter the path for your new spec

cypress\e2e\spec.cy.js

Create specBack

Saldrá la siguiente ventana:

Great! The spec was successfully added

✓ cypress\e2e\spec.cy.js

```
1 describe('template spec', () => {
2   it('passes', () => {
3     cy.visit('https://example.cypress.io')
4   })
5 })
```

✖ Okay, run the spec+ Create another spec

Simplemente continúa presionando el botón de Okay, run the spec y finalmente se abrirá una web de pruebas creada por Cypress para poder ver las funcionalidades que se ofrecen de una manera muy visual y dinámica:

Specs

spec.cy.js 00:05

template spec

✓ passes

TEST BODY

1 visit https://example.cypress.i

o

https://example.cypress.io/

Edge 109 1000x660 (84%)

cypress.io

Commands Utilities Cypress API

GitHub

Kitchen Sink

This is an example app used to showcase Cypress.io testing. For a full reference of our documentation, go to docs.cypress.io

Commands

Commands drive your tests in the browser like a real user would. They let you perform actions like typing, clicking, xhr requests, and can also assert things like "my button should be disabled".

Querying

get

contains

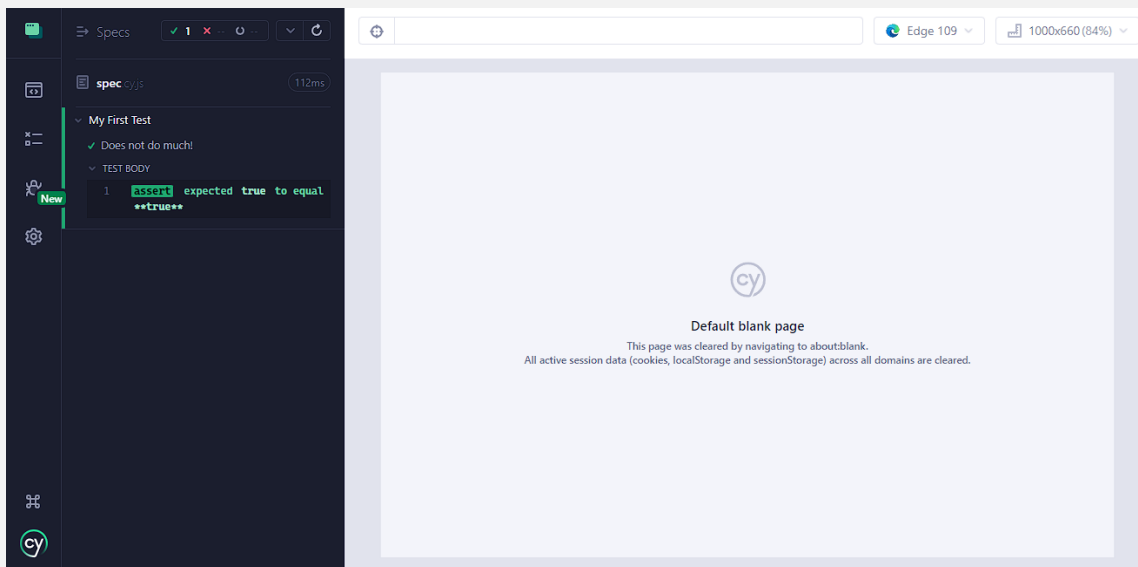
within

root

De momento vamos a escribir nuestro primer test, vamos a ir a nuestro VSCode, dentro de la carpeta cypress estará otra de nombre e2e y dentro de esta un archivo de nombre **“spec.cy.js”**, borraremos el contenido de dicho archivo y colocaremos el siguiente código en su lugar:

```
describe('My First Test', () => {  
  it('Does not do much!', () => {  
    expect(true).to.equal(true)  
  })  
})
```

Guardamos el archivo y revisaremos el navegador de Cypress el cual mostrara lo siguiente:

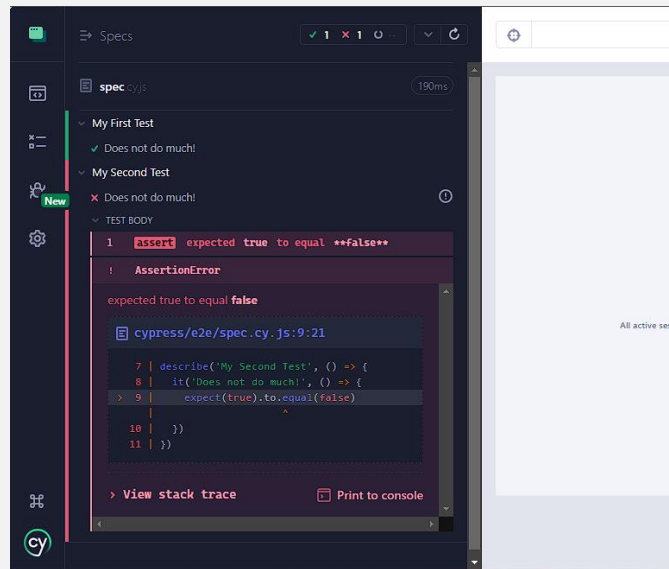


Vemos que en la parte izquierda hay una especie de consola que nos muestra las pruebas realizadas y si pasaron con éxito o no, en este caso muestra el test que escribimos con anterioridad como exitoso.

Ahora intentemos agregar una prueba fallida con el siguiente código:

```
describe('My Second Test', () => {  
  it('Does not do much!', () => {  
    expect(true).to.equal(false)  
  })  
})
```

En el momento en el que guardes el archivo Cypress se actualizará automáticamente realizando de nuevo las pruebas y ahora deberíamos tener lo siguiente:



Como podemos ver nos muestra de manera muy explícita el error que nos lanza Cypress.

Ahora volveremos a la página web de pruebas que nos proporciona Cypress y que vimos antes de escribir nuestras primeras pruebas, para ello vamos a borrar todo el código que hemos escrito con anterioridad y en su lugar colocaremos lo siguiente:

```
describe('My First Test', () => {  
  it('Visits the Kitchen Sink', () => {  
    cy.visit('https://example.cypress.io')  
  })  
})
```

Al guardar notaremos que el comando “visit” nos llevó a la url proporcionada, en este caso la web de pruebas de Cypress.

ADVERTENCIA ANTES DE CONTINUAR: NO INTENTES ACCEDER Y REALIZAR TESTS A OTRAS PÁGINAS WEB POR MEDIO DE CYPRESS (ALGO QUE EN TEORÍA PODEMOS HACER) COMO YOUTUBE O FACEBOOK, YA QUE ESTAS LO PODRÍAN TOMAR COMO UN ATAQUE Y PUEDAN TOMAR ACCIONES EN CONTRA TUYA.

Ahora vamos a intentar algo sencillo como dar clic a un enlace por medio de Cypress, para ello primero debemos cerciorarnos de que dicho enlace exista, para ello colocaremos el siguiente código:

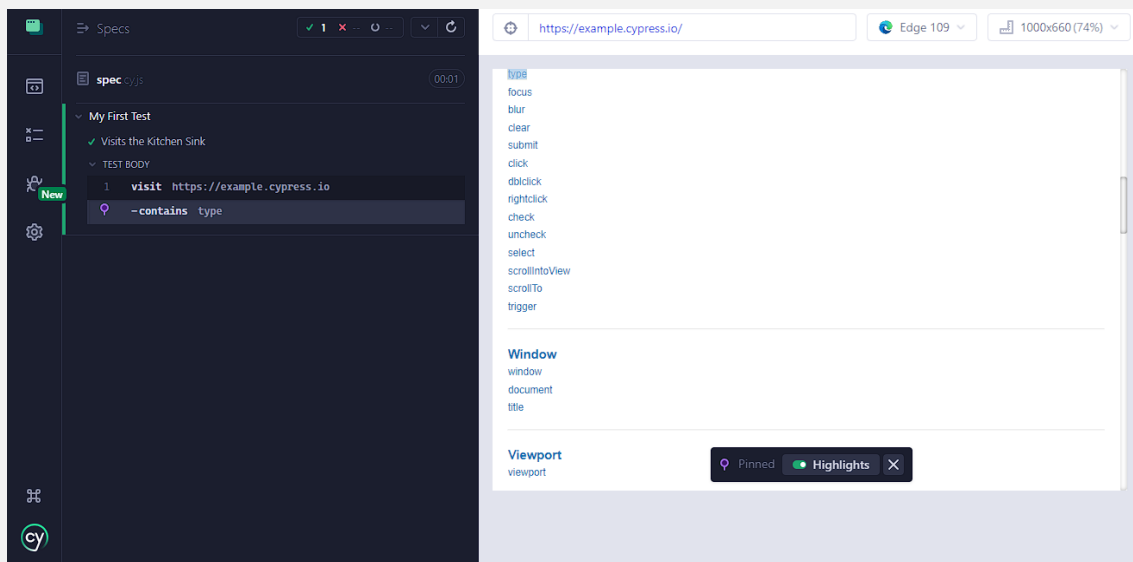
```
describe('My First Test', () => {  
  it('Visits the Kitchen Sink', () => {  
    cy.visit('https://example.cypress.io')
```

```

    cy.contains('type')
  })
})

```

con **'contains'** hacemos una búsqueda de un elemento que este contenido en la página, en este caso buscamos que este **'type'**, si guardamos y revisamos el cypress notaremos que la prueba paso con éxito, ahora da clic en ese renglón de la consola de la prueba de Cypress, eso nos llevara a exactamente donde se encuentra el contenido **'type'** como se puede ver en la imagen (puedes volver a dar clic en dicho renglón de la prueba para desmarcarlo):



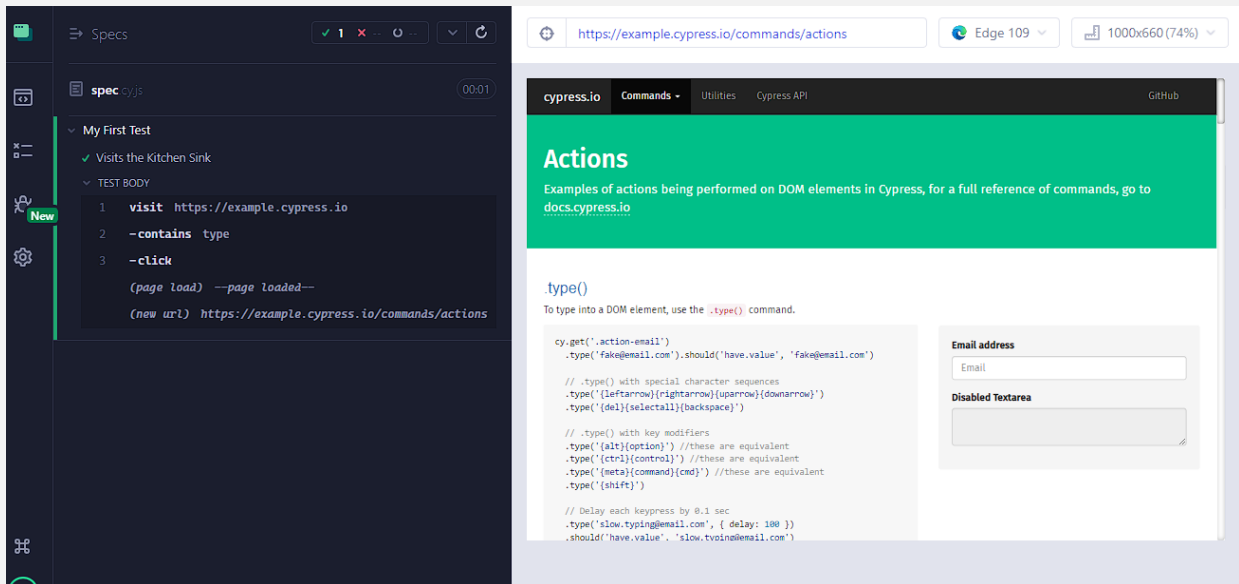
En caso de que no encuentre algún elemento que estemos buscando, simplemente nos mandara un error. Ahora que ya encontramos el elemento podemos darle clic agregando lo siguiente:

```

cy.contains('type').click()

```

Al guardar y revisar el navegador de pruebas notaremos que efectivamente se dio clic en el enlace **'type'** abriendo la siguiente ventana:



Hagamos una afirmación sobre algo en la nueva página en la que hicimos clic. Quizás nos gustaría asegurarnos de que la nueva URL sea la URL esperada. Podemos hacerlo mediante: buscar la URL y encadenar una aserción con `.should()`.

```
describe('My First Test', () => {
  it('clicking "type" navigates to a new url', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()

    // Debe haber una nueva URL que incluya '/commands/actions'
    cy.url().should('include', '/commands/actions')
  })
})
```

Y podemos seguir haciendo más que eso, por ejemplo, podemos seguir agregando lo siguiente:

```
describe('My First Test', () => {
  it('clicking "type" navigates to a new url', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()

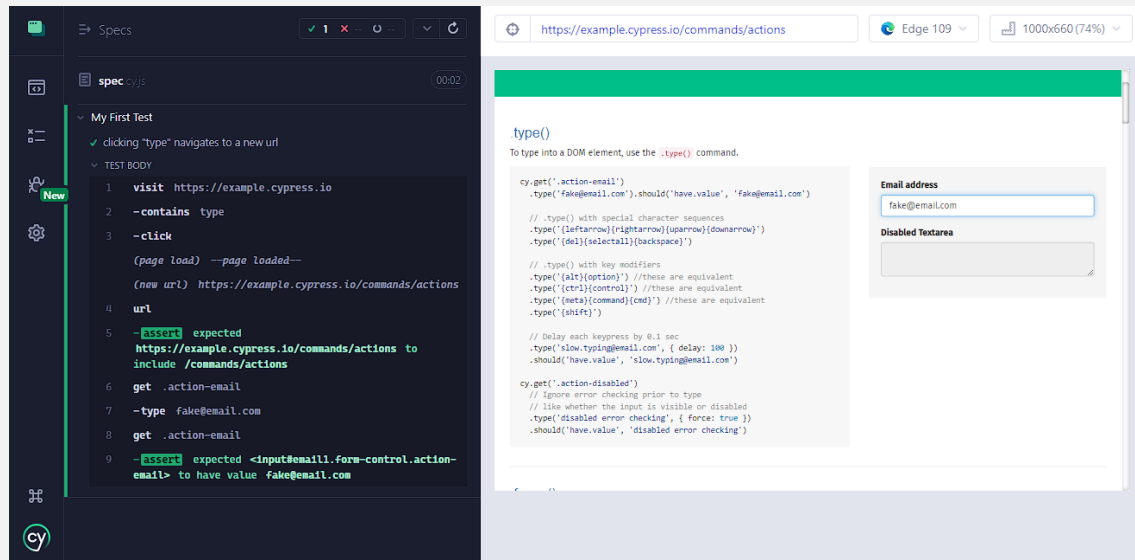
    // Debe haber una nueva URL que incluya '/commands/actions'
    cy.url().should('include', '/commands/actions')

    // Obtener un input, escribir en el input
    cy.get('.action-email').type('fake@email.com')

    // Verificar que el valor ha sido actualizado
    cy.get('.action-email').should('have.value', 'fake@email.com')
```

```
})  
})
```

Lo que resulta en la siguiente ventana:



Como te darás cuenta con Cypress podemos hacer pruebas del DOM muy explícitas donde en esencia simularemos ser un usuario que da clics, navega, escribe o modifica cosas dentro de la web. Si prestas atención dentro de la web de pruebas de cypress notarás que contiene ejemplos con códigos ya escritos como los siguientes:

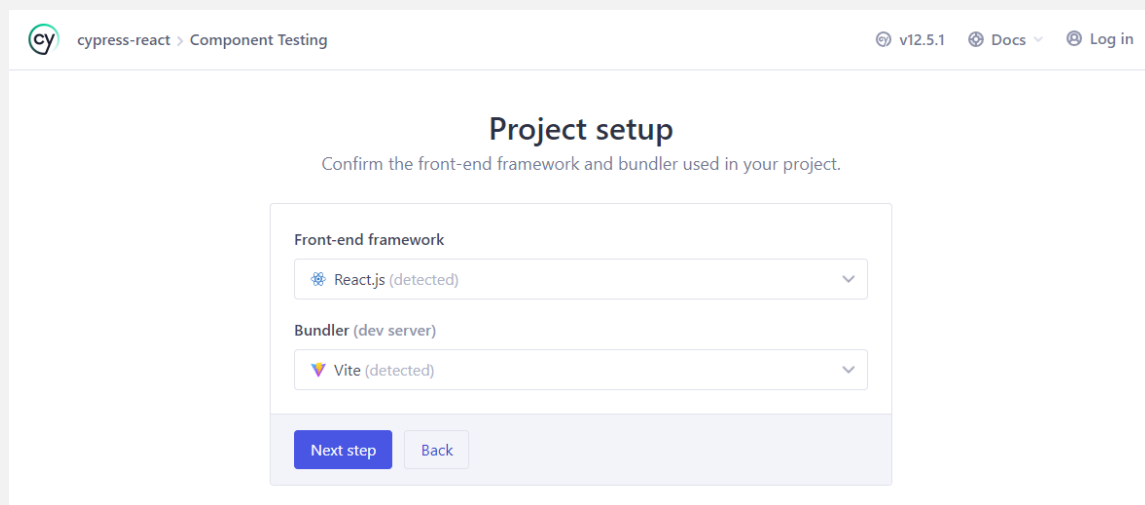


Puedes hacer uso de estos tan solo copiando y pegando en el archivo donde hemos estado escribiendo nuestras pruebas y ver por tu cuenta que es lo que van haciendo de una manera muy visual y descriptiva, la página está llena de estos códigos, explóralos, ve probando varios de ellos para entender de una manera muy práctica el funcionamiento de Cypress. Para una mayor información de lo que poder realizar con Cypress y sus pruebas E2E visita su documentación: [Writing Your First E2E Test | Cypress Documentation](#)

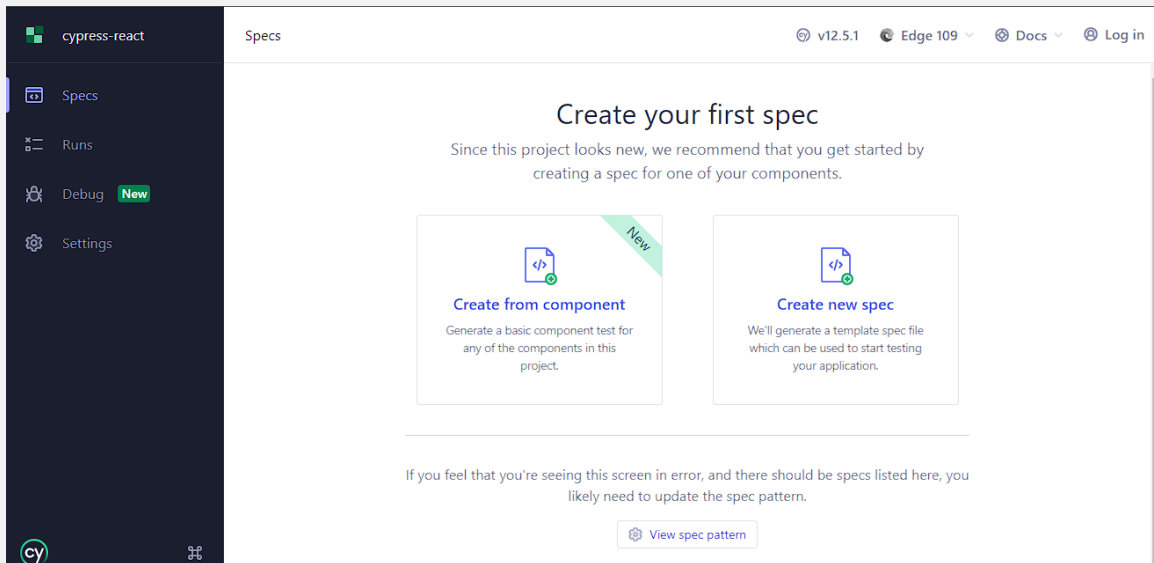
Cypress – React

Ahora veremos cómo hacer test de componentes de React con Cypress, para ello crea una nueva carpeta donde guardar el proyecto, ábrela en VSCode y crea un nuevo proyecto de React con Vite como hemos visto en los anteriores temas de test, entra en dicho proyecto usa el comando de **npm install** y finalmente instala Cypress con el siguiente comando **npm install cypress -D** una vez termine la instalación abre la aplicación como explicamos antes por medio de **npx cypress** open y estaremos listos para comenzar.

Ahora no seleccionaremos la opción de E2E, sino que vamos a seleccionar la opción de **component testing** lo cual ahora nos aparecerá la siguiente ventana:



Como vemos ya selecciona de manera automática el framework y el bundler que estamos usando, en este caso React y Vite respectivamente, simplemente le damos en Next step, a todos los Continue que nos aparecerán, escoge el navegador de pruebas que prefieras y finalmente deberemos tener la siguiente ventana abierta:



Para comenzar dentro de la carpeta src crearemos una de components, y dentro de esta crearemos un componente llamado Stepper.jsx y dentro de este colocaremos el siguiente código:

```
import { useState } from 'react'

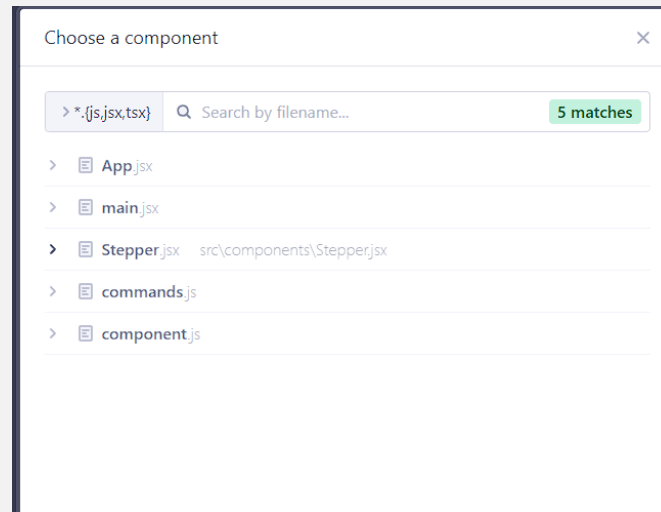
export default function Stepper({ initial = 0, onChange = () => { } }) {
  const [count, setCount] = useState(initial)

  const handleIncrement = () => {
    const newCount = count + 1
    setCount(newCount)
    onChange(newCount)
  }

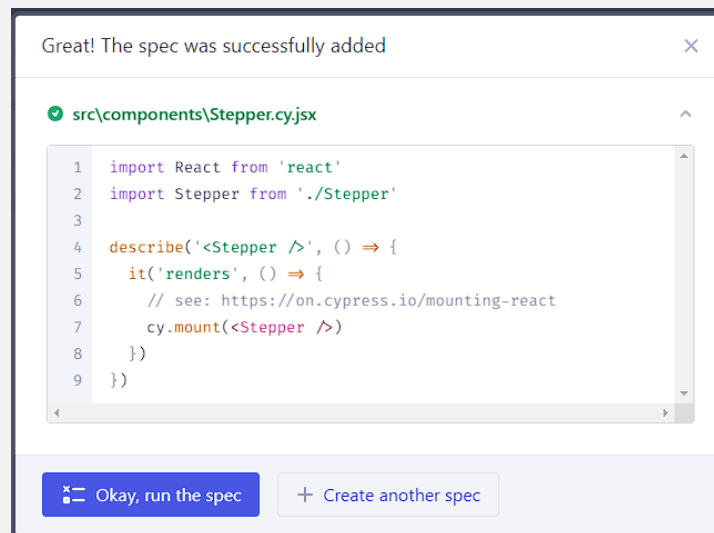
  const handleDecrement = () => {
    const newCount = count - 1
    setCount(newCount)
    onChange(newCount)
  }

  return (
    <div>
      <button data-cy="decrement" onClick={handleDecrement}>
        -
      </button>
      <span data-cy="counter">{count}</span>
      <button data-cy="increment" onClick={handleIncrement}>
        +
      </button>
    </div>
  )
}
```

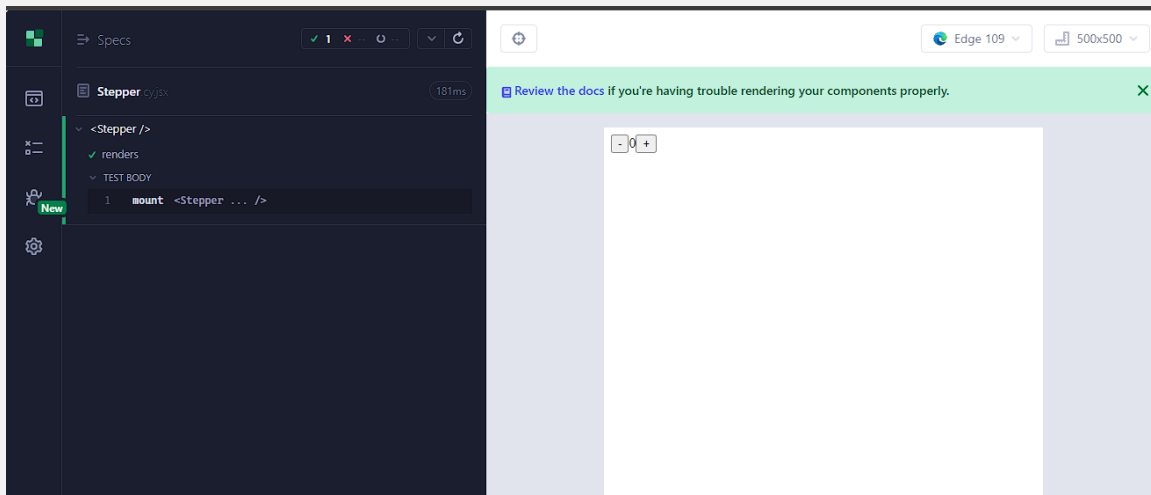
Una vez hecho esto guardamos y en Cypress seleccionaremos la opción de Create from component y posterior a esto seleccionaremos el componente que acabamos de crear:



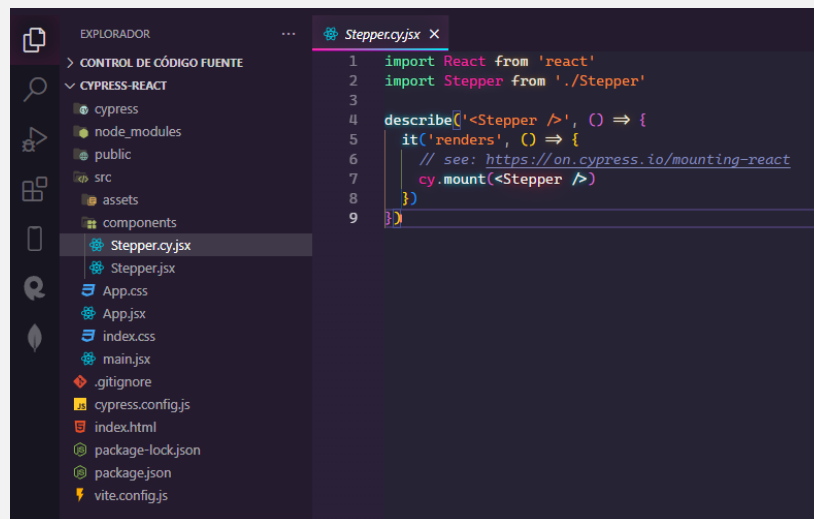
Nos saldrá la siguiente ventana:



Damos clic en Okay, run the spec y veremos cómo se nos renderiza dicho componente en el navegador de pruebas:



Con esto ya podemos empezar a realizar test como lo que hemos visto en el E2E, si revisamos nuestro VSCode veremos que se nos creó un nuevo archivo:



Que es justo donde deberemos colocar las pruebas que vayamos a realizar a dicho componente. Vamos a sustituir dicho código con lo siguiente (no elimines los import):

```

it('stepper should default to 0', () => {
  cy.mount(<Stepper />)
  cy.get('span').should('have.text', '0')
})

```

Guarda el archivo y ve al navegador de pruebas para comprobar que la prueba paso con éxito, como puedes notar lo que hicimos fue obtener el elemento span del componente y verificar que su valor sea de '0'. Si queremos ser más específicos con la obtención de un elemento podemos hacer lo siguiente, nótese que en el componente Stepper.jsx tenemos la siguiente línea:

```
<span data-cy="counter">{count}</span>
```

Lo importante a notar es que tenemos un **data-cy** con el valor de **"counter"** el cual funge como un ID único para cypress, por lo que podemos obtener el elemento por medio de dicho tag de la siguiente manera:

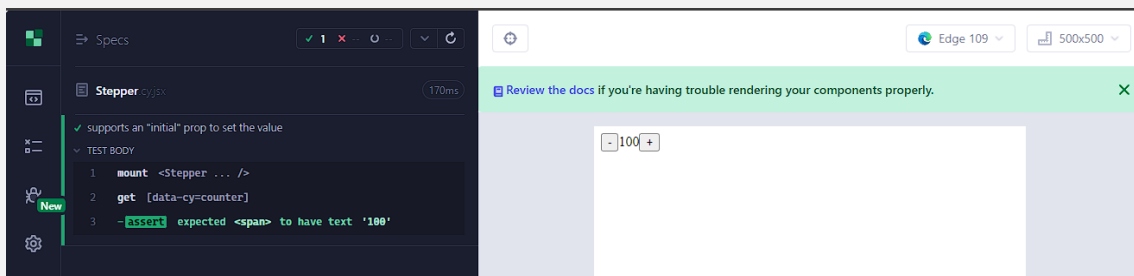
```
cy.get('[data-cy=counter]').should('have.text', '0')
```

Guarda los cambios que hayas realizado y volvamos al navegador de pruebas para comprobar que todo salió correctamente. La documentación de Cypress nos da la siguiente página para tener buenas prácticas al momento de realizar este tipo de pruebas: [Best Practices | Cypress Documentation](#)

También podemos hacer cosas interesantes como enviar ciertos valores a nuestros componentes, si te fijas bien el componente `Stepper.jsx` puede recibir props, específicamente el valor de `initial` que es con el cual el contador va a empezar, en este caso es de 0 por default, pero desde el archivo `Stepper.cy.jsx` podemos enviar un nuevo valor, borra el código anterior y coloca lo siguiente:

```
it('supports an "initial" prop to set the value', () => {  
  cy.mount(<Stepper initial={100} />)  
  cy.get('[data-cy=counter]').should('have.text', '100')  
})
```

Guarda el archivo y regresa al navegador de pruebas para comprobar que la prueba paso con éxito y notarás que el valor del contador paso de 0 a 100.



Como recordaras, el punto de Cypress es hacer test automáticos, de manera que hagamos parecer que es un usuario que está interactuando con nuestra página/aplicación web, por lo que vamos a realizar un par de pruebas para las interacciones que podemos tener en nuestro componente `Stepper.jsx`, para ello borra el código anteriormente realizado y ahora coloca lo siguiente:

```

it('when the increment button is pressed, the counter is incremented',
() => {
  cy.mount(<Stepper />)
  cy.get('[data-cy=increment]').click()
  cy.get('[data-cy=counter]').should('have.text', '1')
})

it('when the decrement button is pressed, the counter is decremented',
() => {
  cy.mount(<Stepper />)
  cy.get('[data-cy=decrement]').click()
  cy.get('[data-cy=counter]').should('have.text', '-1')
})

```

Guarda y revisa el navegador de pruebas, puedes ver que lo que se hizo fue montar el componente, obtener el elemento con el tag correspondiente y dar clic en este, para finalmente revisar que el texto del contador tenga el valor de 1 en el primer test y -1 en el segundo.

Cypress también puede hacer uso de los “spies” o “espías”, un espía es una función especial que realiza un seguimiento de cuántas veces se llamó y cualquier parámetro que fuera llamado con. Podemos pasar un espía al evento onChange e interactuar con él.

Borra las pruebas anteriores y coloca el siguiente código:

```

it('clicking + fires a change event with the incremented value', () =>
{
  const onChangeSpy = cy.spy().as('onChangeSpy')
  cy.mount(<Stepper onChange={onChangeSpy} />)
  cy.get('[data-cy=increment]').click()
  cy.get('@onChangeSpy').should('have.been.calledWith', 1)
})

```

Guarda y revisa el navegador de pruebas. Primero, creamos un nuevo espía llamando al método cy.spy(). Pasamos una cadena que le da al espía un alias, que le asigna un nombre por el cual podemos hacer referencia a él más tarde. En cy.mount(), inicializamos el componente y le pasamos el espía. Después de eso, hacemos clic en el botón de incremento.

La siguiente línea es un poco diferente. Hemos visto cómo podemos usar el método cy.get() para seleccionar elementos, pero también podemos usarlo para capturar cualquier alias que hayamos configurado previamente. Usamos cy.get() para obtener el alias del espía (anteponiendo un ampersand al nombre del alias). Afirmamos que el método fue llamado con el valor esperado.

Con esto habremos terminado las pruebas a nuestro componente Stepper.jsx, para profundizar más en lo que podemos hacer en Cypress revisa esta parte de la documentación: [Introduction to Cypress | Cypress Documentation](#)

Sinon JS

Introducción

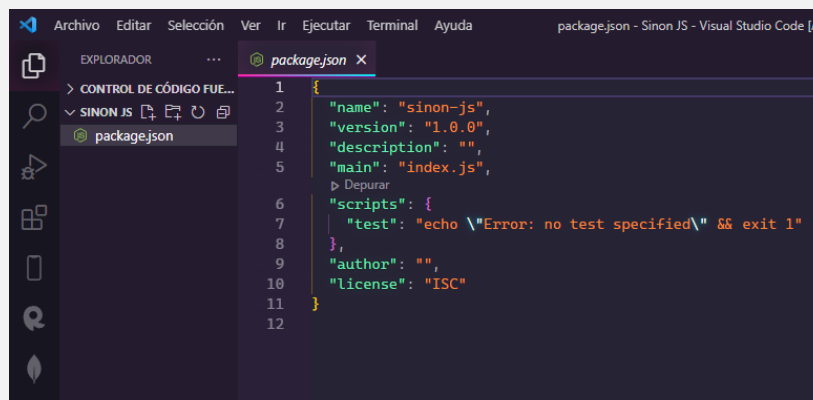
Sinon.js es una librería de JavaScript que se utiliza principalmente para simular objetos y funciones en pruebas unitarias. Con Sinon.js, se pueden crear "falsificaciones" o "falsos" de objetos y funciones que son utilizados en un módulo o componente de una aplicación, lo que permite a los desarrolladores probar su código en un entorno controlado y predecible.

Sinon.js también ofrece otras funciones útiles para pruebas, como espías, stubs y mocks. Los espías permiten a los desarrolladores verificar si una función ha sido llamada y con qué argumentos, mientras que los stubs se utilizan para reemplazar funciones en un módulo o componente. Por otro lado, los mocks se utilizan para simular todo un objeto o módulo.

En general, Sinon.js es una herramienta útil para mejorar la calidad y la eficiencia del proceso de pruebas unitarias en JavaScript. Con su amplia gama de funciones y facilidad de uso, Sinon.js es una opción popular para desarrolladores que desean mejorar la calidad y fiabilidad de su código.

Instalación

Vamos a crear una carpeta donde almacenar nuestro proyecto y abrimos nuestro VSCode en dicha carpeta, abrimos la terminal y lo primero que necesitamos instalar es mocha por medio del siguiente comando **npm install -g mocha** seguido de esto usaremos el comando **npm init** y simplemente iremos dando Enter hasta que finalice y aparezca un **package.json** similar a este:



```
1 {
2   "name": "sinon-js",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
8   },
9   "author": "",
10  "license": "ISC"
11 }
12
```

Ahora usamos el siguiente comando ***npm install --save-dev mocha*** una vez finalizada dicha instalación estaremos listos para trabajar con sinon JS.

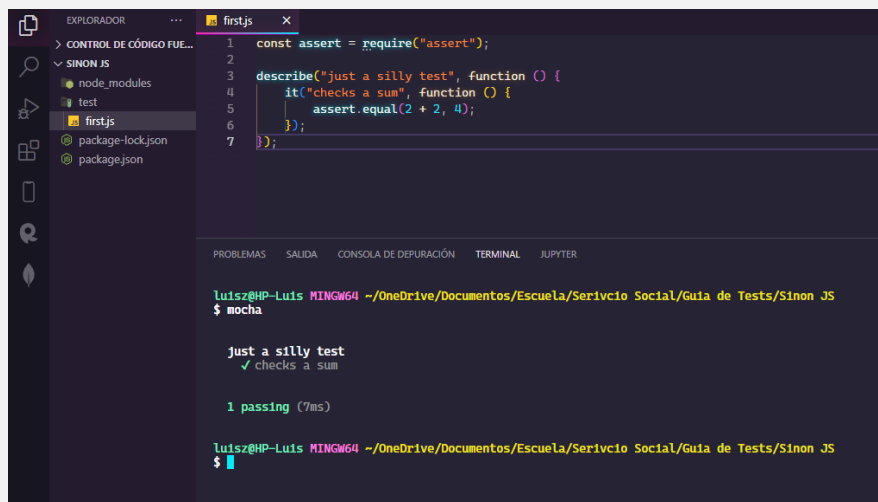
Primeros pasos

Crearemos una carpeta llamada "test" y dentro de ella vamos a crear un archivo de nombre "first.js" y en el agregaremos el siguiente código:

```
const assert = require("assert");

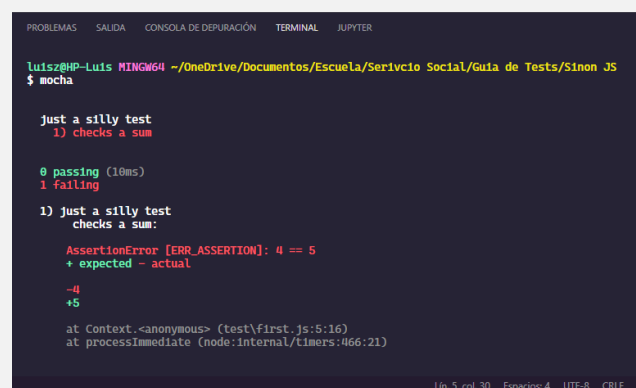
describe("just a silly test", function () {
  it("checks a sum", function () {
    assert.equal(2 + 2, 4);
  });
});
```

Como puedes ver, nuestra prueba es supersimple. Simplemente, verifica si dos más dos son cuatro. Para correr la prueba simplemente en la terminal escribe ***mocha***:



The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows a project structure with a 'test' folder containing 'first.js'. The main editor displays the content of 'first.js', which is the same code as shown in the previous block. Below the editor, the TERMINAL panel is active, showing the command 'mocha' being executed. The output indicates that the test 'just a silly test' passed, specifically the 'checks a sum' assertion, with a result of '1 passing (7ms)'.

Ahora cambia el 4 por cualquier otro número para ver como falla:

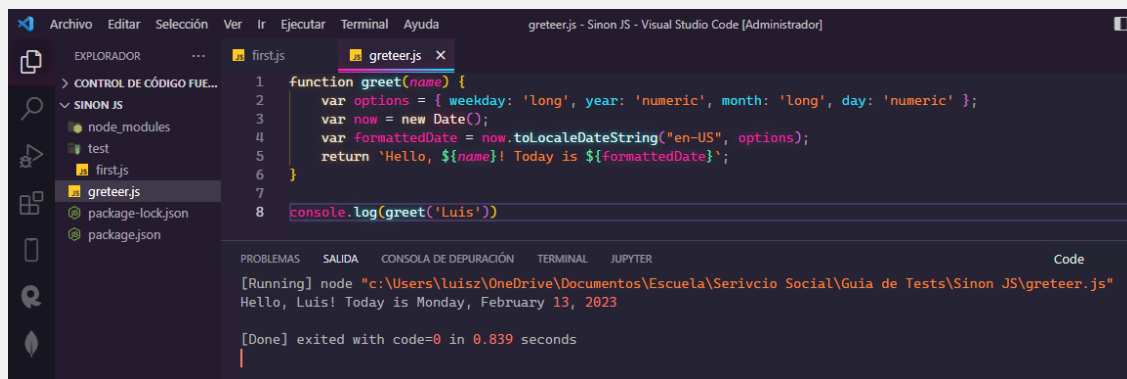


The screenshot shows the VS Code interface with the TERMINAL panel active. The command 'mocha' has been executed, but the test has failed. The output shows '0 passing (10ms)' and '1 failing'. The failing test is 'just a silly test' with the assertion 'checks a sum'. The error message is 'AssertionError [ERR_ASSERTION]: 4 == 5', indicating that the expected value (4) did not match the actual value (5). The stack trace shows the error occurred at 'test\first.js:5:16'.

Pero realmente aún no estamos haciendo uso de Sinon JS por lo que llego la hora de instalarlo con el siguiente comando: **`npm install --save-dev sinon`** Una vez finalizada la instalación vamos a crear una función para poder probar los tests, en la carpeta principal creamos un archivo de nombre **`greeter.js`** y colocamos el siguiente código:

```
function greet(name) {  
  var options = { weekday: 'long', year: 'numeric', month: 'long',  
day: 'numeric' };  
  var now = new Date();  
  var formattedDate = now.toLocaleDateString("en-US", options);  
  return `Hello, ${name}! Today is ${formattedDate}`;  
}
```

Lo que hace esta función es algo muy sencillo: tras recibir un nombre, devuelve una cadena en la que saluda a la persona e informa del día actual. Entonces, eso es lo que obtengo cuando le paso un nombre: **`"Hello, Luis! Today is Thursday, October 22, 2020"`**.



Vamos a usar SinonJS para eso. Al proporcionar una hora actual "falsa/fake", podremos verificar si la función funciona correctamente. La prueba va a ser rápida y determinista, como deberían ser las pruebas unitarias. Tampoco dependerá de ninguna dependencia volátil externa. Entonces, vamos a ello.

Vamos a modificar el código de greeter para que se pueda exportar la función:

```
(function (exports) {  
  
  function greet(name) {  
    var options = { weekday: 'long', year: 'numeric', month:  
'long', day: 'numeric' };  
    var now = new Date();  
    var formattedDate = now.toLocaleDateString("en-US", options);  
    return `Hello, ${name}! Today is ${formattedDate}`;  
  }  
})
```

```

    }

    exports.greet = greet;
  }) (this);

```

Ahora en el archivo de first.js borramos el anterior código y ahora colocamos lo siguiente:

```

const assert = require("assert");
const greeter = require("../greeter.js");

describe("testing the greeter", function () {
  it("checks the greet function", function () {
    assert.equal(greeter.greet('Alice'), 'Hello, Alice! Today is Friday, January 15, 2021.');
```

Si guardamos todo y corremos la prueba nos deberá salir algo así:

```

luisz@HP-Luis MINGW64 ~/OneDrive/Documentos/Escuela/Servicio Social/Guia de Tests/Sinon JS
$ mocha

testing the greeter
  1) checks the greet function

0 passing (88ms)
1 failing

1) testing the greeter
  checks the greet function:
    AssertionError [ERR_ASSERTION]: 'Hello, Alice! Today is Monday, February 13, 2023' == 'Hello, Alice! Today is Friday, January 15, 2021.'
      + expected - actual

      -Hello, Alice! Today is Monday, February 13, 2023
      +Hello, Alice! Today is Friday, January 15, 2021.

      at Context.<anonymous> (test\first.js:6:16)
      at processImmediate (node:internal/timers:466:21)

```

Para hacer que esta prueba pase, necesitamos una manera de fingir que estamos en un día diferente, algo que podemos hacer gracias a Sinon JS, para ello vamos a volver a modificar el código de first.js de la siguiente manera:

```

const assert = require("assert");
const greeter = require("../greeter.js");
const sinon = require("sinon");

describe("testing the greeter", function () {
  it("checks the greet function", function () {
    var clock = sinon.useFakeTimers(new Date(2021, 0, 15));
    assert.equal(greeter.greet('Alice'), 'Hello, Alice! Today is Friday, January 15, 2021');
```

Ahora si guardamos y corremos la prueba notaremos que ahora si pasa. Los cambios fueron mínimos. Primero, justo al comienzo del archivo, requerimos Sinon, para poder usarlo después. En el método de prueba, justo antes de la afirmación, llamamos a la función "useFakeTimers", pasando el 15 de enero de 2021, como la fecha. Recuerde que, en JavaScript, los meses comienzan en cero, por lo que enero es el número cero en lugar del esperado.

Lo anterior fue un pequeño calentamiento por así decirlo para Sinon JS, ahora vamos a ver cómo podemos hacer uso de los ejemplos que se nos da en la documentación de Sinon JS: [API documentation - Sinon.JS \(sinonjs.org\)](https://sinonjs.org/docs/); que desgraciadamente no explica bien lo que se debe tener previamente instalado para su correcto funcionamiento, pero no te preocupes aquí te explico como tener todo preparado.

Sinon JS – Pruebas con la Documentación

Primero, como siempre, crea una carpeta donde almacenar el proyecto y ábrela en el VSCode, vamos a usar los siguientes comandos en la terminal:

- **`npm install @fatso83/mini-mocha`**
- **`npm install @sinonjs/referee`**
- **`npm install pubsub-js`**

Una vez tengas instalado lo anterior simplemente queda instalar mocha y sinon, puedes hacerlo con los siguientes comandos: **`npm i sinon`** y **`npm i mocha`**.

OJO: puede que algunas pruebas fallen debido a que algunas de estas requieren de algún paquete que no se menciona de manera explícita, ante esto presta atención en los const que se definen antes de la prueba, fíjate en los require fíjate en los que coincida el error que mande la consola y simplemente usa "npm install nombre-del-paquete", por ejemplo: `const bluebird = require("bluebird");` Como no tenemos instalado "bluebird" te dará error, por lo que tendrás que hacer "npm install bluebird", cuando corras de nuevo la prueba veras que esta vez no fallará.

Ahora ya estamos listos para comenzar a trabajar con los ejemplos que nos ofrece la documentación, por ejemplo, podemos ir al apartado de Stubs y después de leer las explicaciones que nos da ejemplos como este:

```
const sinon = require("sinon");
const { assert } = require("@sinonjs/referee");

describe("stubbed callback", function () {
  it("should behave differently based on arguments", function () {
    const callback = sinon.stub();
    callback.withArgs(42).returns(1);
    callback.withArgs(1).throws("name");
  });
});
```

```

    assert.isUndefined(callback()); // No return value, no
exception
    assert.equals(callback(42), 1); // Returns 1
    assert.equals(callback.withArgs(42).callCount, 1); // Use
withArgs in assertion
    assert.exception(() => {
        callback(1);
    }); // Throws Error("name")
  });
});

```

Lo ponemos en un archivo de nombre **test.js**, guardamos y en la terminal hacemos uso del comando **mocha** para poder correr la prueba:

```

luisz@HP-Luis MINGW64 ~/OneDrive/Documentos/Escuela/Servicio Social/Guia de Tests/Sinon Doc
$ mocha

  stubbed callback
    ✓ should behave differently based on arguments

  1 passing (13ms)

luisz@HP-Luis MINGW64 ~/OneDrive/Documentos/Escuela/Servicio Social/Guia de Tests/Sinon Doc
$

```

Lee la documentación y entiende los ejemplos que esta te da, es la mejor forma de entender el cómo funciona este framework en particular.

Chai JS

Introducción

Chai.js es una biblioteca de aserciones (assertions) para JavaScript que se utiliza principalmente en pruebas unitarias y de integración. Chai.js proporciona una amplia gama de funciones para hacer afirmaciones sobre el comportamiento y la salida de un módulo o componente, lo que permite a los desarrolladores probar su código en un entorno controlado y predecible.

Chai.js se integra fácilmente con marcos de prueba populares como Mocha y Jasmine, lo que la hace una opción popular para desarrolladores de JavaScript. Chai.js proporciona varios estilos de aserciones, incluyendo el estilo de "debería" (should), el estilo de "esperar" (expect) y el estilo de "assert", lo que permite a los desarrolladores elegir el que mejor se adapte a sus necesidades y preferencias.

En general, Chai.js es una herramienta útil para mejorar la calidad y la eficiencia del proceso de pruebas en JavaScript. Con su amplia gama de funciones y estilos de aserciones, Chai.js es una opción popular para desarrolladores que desean mejorar la calidad y fiabilidad de su código.

Instalación

Con Chai haremos uso directamente de su documentación: [Chai \(chaijs.com\)](https://chaijs.com) para empezar, crea una nueva carpeta, ábrela en VSCode. La instalación de Chai y el poder usarla requiere de unas pequeñas adiciones, primero que nada, agrega el siguiente comando para instalar todo lo necesario: **`npm i chai cross-env mocha test`**

Una vez finalizada la instalación de toso los paquetes abre tu package.json, debería verse de esta manera:

```
{
  "dependencies": {
    "chai": "^4.3.7",
    "cross-env": "^7.0.3",
    "mocha": "^10.2.0",
    "test": "^3.3.0"
  }
}
```

Solo nos falta agregar algo a nuestro .json para poder hacer los test:

```
{
  "dependencies": {
    "chai": "^4.3.7",
    "cross-env": "^7.0.3",
    "mocha": "^10.2.0",
    "test": "^3.3.0"
  },
  "scripts": {
    "test": "cross-env NODE_ENV=test mocha --exit"
  }
}
```

Ya estamos listos para empezar a usar los ejemplos que nos da la documentación de Chai JS. Para empezar, crea una carpeta llamada "test" dentro de esta crea un archivo del nombre que gustes en mi caso será "test.js" y si te das cuenta el primer tema de la documentación de chai son los assert y aparece un código como este:

```
var assert = require('chai').assert
, foo = 'bar'
, beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };
```

```

assert.typeOf(foo, 'string'); // without optional message
assert.typeOf(foo, 'string', 'foo is a string'); // with optional message
assert.equal(foo, 'bar', 'foo equal `bar`');
assert.lengthOf(foo, 3, 'foo`s value has a length of 3');
assert.lengthOf(beverages.tea, 3, 'beverages has 3 types of tea');

```

Algo con lo que hay que tener cuidado es que no está la función para hacer la prueba como es debido, por ello debemos anotar lo anterior de la siguiente manera:

```

const assert = require('chai').assert
, foo = 'bar'
, beverages = { tea: ['chai', 'matcha', 'oolong'] };

describe('Primer test de la documentacion', () => {
  it('assert primer ejemplo', () => {
    assert.typeOf(foo, 'string'); // without optional message
    assert.typeOf(foo, 'string', 'foo is a string'); // with
optional message
    assert.equal(foo, 'bar', 'foo equal `bar`');
    assert.lengthOf(foo, 3, 'foo`s value has a length of 3');
    assert.lengthOf(beverages.tea, 3, 'beverages has 3 types of
tea');
    assert.equal(foo, 'bar', 'no es igual');
  });
});

```

Simplemente, guardamos y en la terminal usamos el comando **npm run test** y veremos qué pasa con éxito, con esto ya no deberías tener problemas para poder trabajar con la documentación de Chai JS, así que revisa los ejemplos y lee la información que se te proporciona.

Créditos

Guía creada por: Luis Emmanuel Ramírez Fernández

Fuentes

1. Documentación: Facebook. (2021). Getting Started - Jest. Retrieved September 30, 2021, from <https://jestjs.io/docs/en/getting-started>
2. Documentación: Cypress.io. (2021). Cypress Documentation. Retrieved September 30, 2021, from <https://docs.cypress.io/guides/overview/why-cypress>
3. Documentación: Chai.js. (2021). API Documentation. Retrieved September 30, 2021, from <https://www.chaijs.com/api/>
4. Sinon.js. (2021). Standalone and test framework agnostic JavaScript test spies, stubs and mocks (pronounced 'sigh-non'). Retrieved September 30, 2021, from <https://sinonjs.org/>