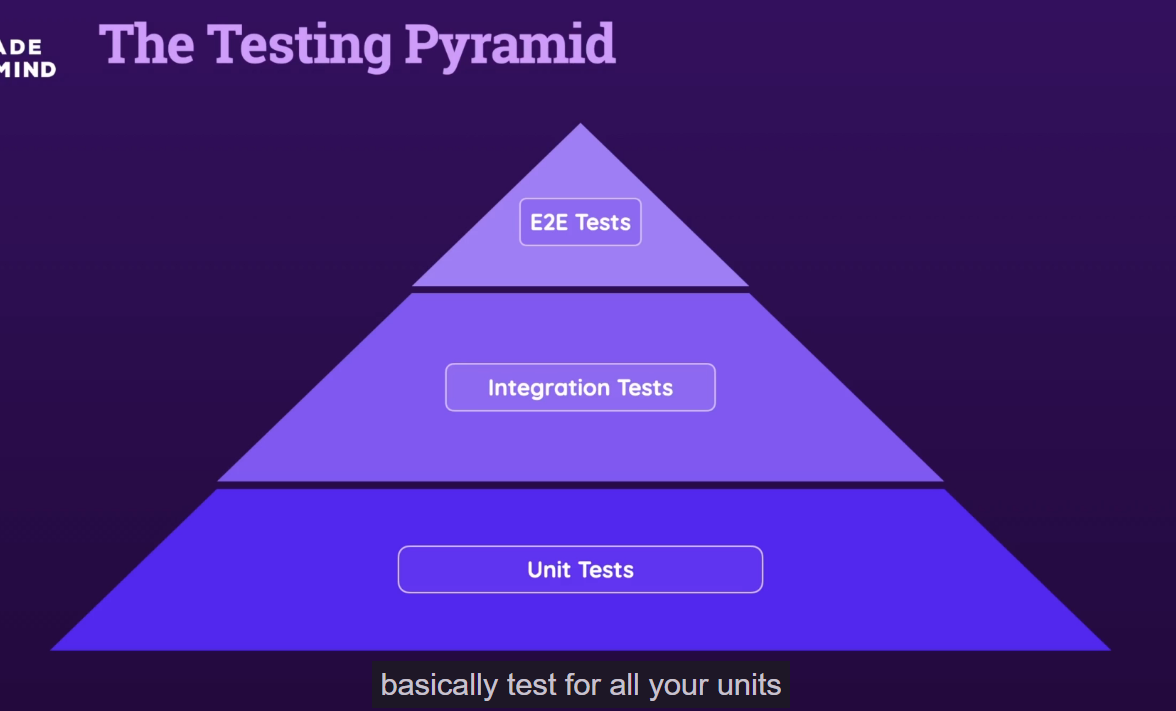


Unit testing se hace el test para cada unidad ósea cada función

Integration testing se hace el test de la combinación de las unidades, testeas las combinaciones que vos queres testear no todas las combinaciones posibles

End to end es ya verificar todo el recorrido por ejemplo de una api es mas relacionado a lo que el usuario puede llegar a hacer, se veulve complicado testear todos los posibles comportamientos.



Tenemos JEST y Vitest y Chai, en este vamos a ver Vitest

**VITEST:**

Para instalarlo, en la carpeta del proyecto hay que hacer, npm install –save-dev vitest

En el package.json tiene que estar esto,pueden haber mas cosas pero esos son los fundamentales:

{

  "scripts":{

    "test": "vitest --globals"

  },

  "devDependencies": {

    "vitest": "^0.14.2"

  }

}

**Proyectos:**

**En el primero:**

En la carpeta del proyecto le hace un npm install primero

Después npm start para abrir el browser

**En el segundo:**

Npm install

Node app.js

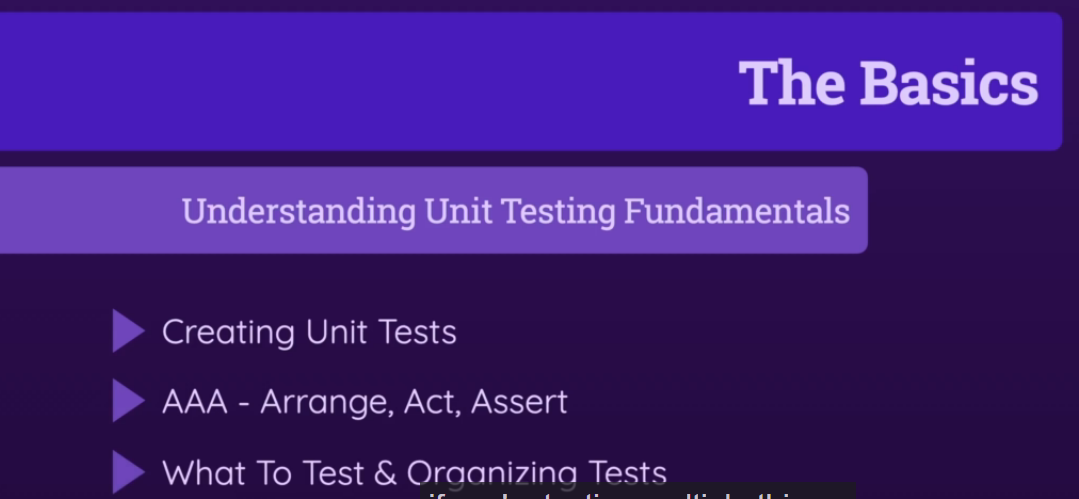
Toma el host en el localhost:3000

**En el tercero:**

Npm install

Node app.js

**Comenzamos:**

****

**Primer proyecto frontend:**

Hacemos el test de math.js , creamos un nuevo archivo , vitest te permite hacer el test en el mismo file, llamado math.test.js o math.spec.js

Para que en el archivo de test te tome la función test o it tenes que importar arriba

import { test } from 'vitest';

o lo que es lo mismo

import { it } from 'vitest';

igual, se puede hacer que sea global haciendo en el package.json

"test": "vitest --run --reporter verbose --globals",

Si lo importas en cada archivo te da un mejor autocompletado

Si queremos testear la función add la tenemos que importar

import { add } from './math';

Tambien importamos el expect

import { it, expect } from 'vitest';

la function es esta:

export function add(numbers) {

  let sum = 0;

  for (const number of numbers) {

    sum += number;

  }

  return sum;

}

Y el test quedaría asi:

it('should summarize all number values in a array', ()=>{

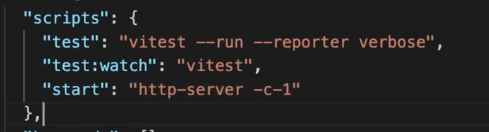
    const result = add([1,2,3]);

    expect(result).toBe(6);

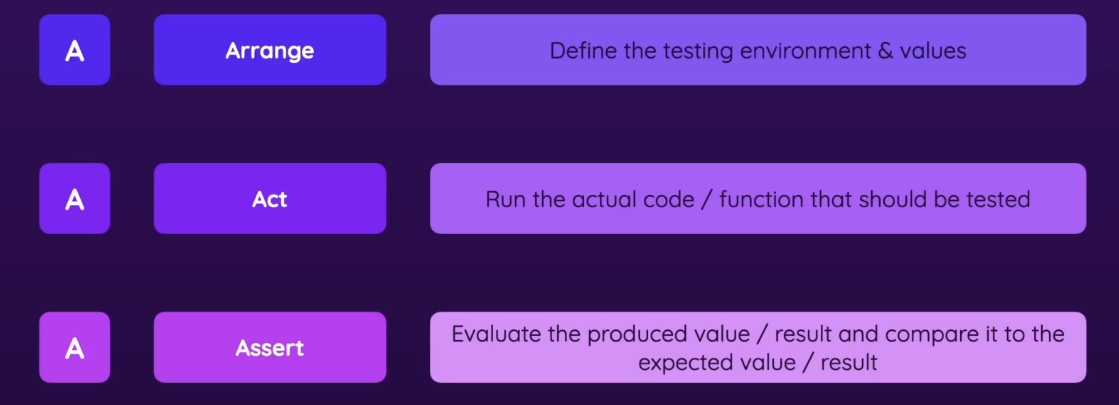
})

Para correr el test hacemos npm test

Mantener observando el test: se hace con npm run test:watch



**Triple A pattern: Arrange, act, assert**

****

El test que hicimos antes no cumple con el arrange, entonces lo que hacemos es separarlo en otra variable y pasarlo como se espera en la función.

La parte del expect que será el assert también se puede mejorar. Porque ahí le hardcodeamos el valor

En vez de hardcodear 6 le calculamos el resultado esperado

it('should summarize all number values in a array', ()=>{

    const result = add([1,2,3]);

    expect(result).toBe(6);

})

Quedaría así

it('should summarize all number values in a array', ()=>{

    //arrange

    const numbers = [1,2,3]

    //act

    const result = add(numbers);

    //assert

    // El reduce va sumando el valor anterior con el actual y va recorriendo, el cero es desde donde empieza

    const expectedResult = numbers.reduce((prevValue, curValue) => prevValue + curValue, 0)

    expect(result).toBe(expectedResult);

})

Test si se envia un dato mal, x ejemplo un texto en vez de un numero. Acá igual esto tira un error, porque no trae NaN sino que te concatena y te tira un string. Debería ser de otra forma creo yo

it('should yield NaN if at leat one invalid number is provided', ()=>{

    const inputs = ['invalid', 1];

    const result = add(inputs);

    expect(result).toBeNaN();

})

En este otro caso también va a fallar, porque si se pasan parámetro como string los va a contactenar

Los mas que le puso antes de las variables en el expected result son para que se sumen bien los números

Expected result va a ser 3 y result va a ser 012 porque en la función empieza con 0 entonces 0+1+2 = 012



it('should yield a correct sum if an array of numeric string values is provided', ()=>{

    const numbers = ['1', '2'];

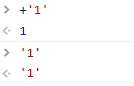
    const result = add(numbers);

    const expectedResult = numbers.reduce((prevValue, curValue) => +prevValue + +curValue, 0)

    expect(result).toBe(expectedResult)

})

Poniendole un mas antes a un string lo convierte en numero



Le agregamos entonces el +number a la función y con eso pasarían todos los tests

export function add(numbers) {

  let sum = 0;

  for (const number of numbers) {

    sum += +number;

  }

  return sum;

}

Si no se pasan valores que el resultado sea 0

it('should yield 0 if an empty array is provided', ()=>{

    const numbers = [];

    const result = add(numbers);

    expect(result).toBe(0);

})

**Testear errores:**

Los errores no se retornan en js entonces , se podría hacer con un try catch, pero mas fácil se podría hacer así: por ejemplo si debería tirar un error si no se pasan valores

Con el toThrow creo que verifica si tira algún error, entonces guardamos la función que queremos llamar en una constant y le hacemos el expect a la constante, si tira un error va a pasar el test. El error lo va a tirar porque no le pasamos parámetros a la función add(). Cuando queres preguntar por el error si o si hay que hacerlo con una constante así. Imagino que podría ser tipo var Tambien.

it('should throw an error if no value is passed into the function', ()=>{

    const resultFn = ()=>{

        add();

    };

    expect(resultFn).toThrow();

})

**Negar:**

expect(resultFn).toThrow();

expect(resultFn).not.toThrow();

Asi en todas las funciones

Otros test:

Si se pasan varios parametros en vez de un array x ej

add(1, 2, 3);

Eso no es lo que espera la función.

it('should throw an error if provided with multiple arguments instead of an array', ()=>{

    const num1 = 1;

    const num2 = 2;

    const resultFn = ()=>{

        add(num1,num2);

    }

    expect(resultFn).toThrow();

})

Seguimos con toThrow

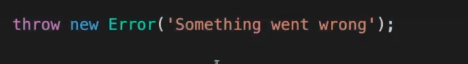
Como saber si estas chequeando por el tipo de error que vos queres , en el caso anterior si le pasas varios parámetros y ninguno es un array no se va a poder iterar entonces te va a tirar un error, y js tira un mensaje de “is not iterable”

Si al .toThrow le pones un not vas a ver el error exacto

Entonces le pasas el mensaje entre las dos barras

expect(resultFn).toThrow(/is not iterable/);

Ahora, si en la función hay un error en especifico algo así :

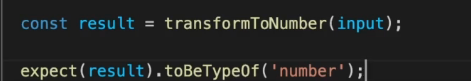


Va atirar ese error, y no el que pusimos en el test, entonces va a fallar

Saber el tipo de variable, ósea si es string, number, etc.

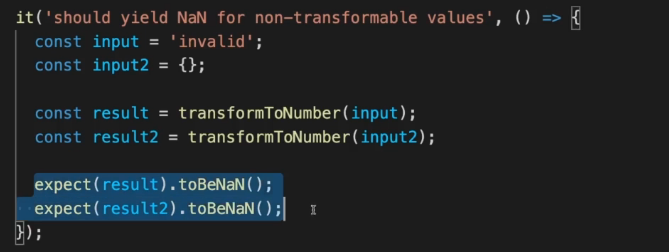
expect(result).toBeTypeOf('number');

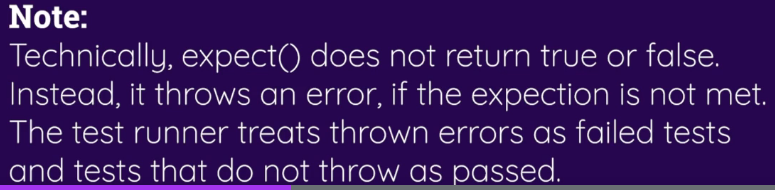
Si pones dos expect los dos tienen que pasar para que no de error el test



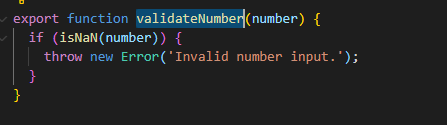
Se pueden probar todos los escenarios en un test:

Ahí podes seguir probando si mandas un objeto, un array o lo que sea

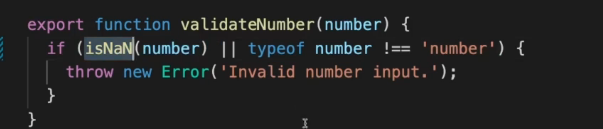




Si vos tenes esta función que te tira error si no se pasa un numero, pero si vos pasas un ‘1’ en string va a pasar igual y no va a tirar el error.



Entonces lo que podemos hacer ahí es, consultar si el tipo de dato no es numérico que entre al if y tire el error, ahí si le pasas entonces ‘1’ va a tirar el error



**FIN DEL PRIMER MODULO**

**Que tests escribir?**

Hasta ahora no escribimos tests de los archivos app.js ni parser.js

SOLO SE TIENE QUE TESTEAR MI CODIGO, NO TESTEAR CODIGO DE TERCEROS

Xq? Xq no podemos testear lo que no podemos cambiar.

Claro, por ejemplo no queremos chequear que el queryselector funcione bien. Xq ya es de js.

No vas a testear el fetch vas a testear lo que haces con la respuesta.

Probar una sola cosa por test

Testear lo necesario, en vez de pasar 50 numeros en un array si con 2 te alcanza para testear pasar 2.

En lo posible tener 1 solo expect, o la menor cantidad que se pueda

**FIN DE SEGUNDO MODULO**

**Integration tests:**

Testeamos este:

export function cleanNumbers(numberValues){

  const numbers = [];

  for (const numberInput of numberValues) {

    validateStringNotEmpty(numberInput);

    const number = transformToNumber(numberInput);

    validateNumber(number);

    numbers.push(number);

  }

  return numbers;

}

Acá se le pasa un array de numeros y los tiene que limpiar, vemos que llama a otras funciones, ahí entonces tenemos que aplicar integration tests. Esta función valida que no este vacio, los transforma a numero, y valida que sea un numero, después retorna con el array,

En el siguiente test al ejecutar cleanNumbers ya estamos ejecutando las otras funciones de adentro

    it('should return an array of number values if an array of string number values is provided', ()=>{

        const numberValues = ['1', '2'];

        const cleanedNumebrs = cleanNumbers(numberValues);

        expect(cleanedNumebrs[0]).toBeTypeOf('number');

    });

Es importante igual hacer integration tests, xq si en la función llamas en el lugar incorrecto a una función de adentro o pasas mal un parámetro puede fallar, ahí esta la importancia de testear ya con las otraS FUNCIONES integradas

También se podría hacer con esto

expect(cleanedNumebrs).toBe([1, 2]);

PERO, esto va a tirar error, si usas toBe va a esperar que el objeto sea exactamente igual a lo que le pasas, no es el mismo objeto en este caso, Entonces ahí usar .toEqual

expect(cleanedNumebrs).toEqual([1, 2]);

toEqual inspecciona mas a fondo

**Fin**

**Async code:**

Cuando queres testear algo asíncrono hay que poner el done como parámetro en el test

it('should generate a token value', (done)=>{

Se va a esperar hasta que done se ejecute;

Quedo así:

it('should generate a token value', (done)=>{

    const testUserEmail = 'test@gmail.com';

    generateToken(testUserEmail, (err, token)=>{

        try{

            expect(token).toBeDefined();

            done();

        }catch (err){

            done(err);

        }

    });

})

El segundo parámetro de la función es el callback;

Para esta función:

export function generateTokenPromise(userEmail) {

  const promise = new Promise((resolve, reject) => {

    jwt.sign({ email: userEmail }, 'secret123', (error, token) => {

      if (error) {

        reject(error);

      } else {

        resolve(token);

      }

    });

  });

  return promise;

}

Se hizo

//con las promises se puede hacer asi:

it('should generate a token value', ()=>{

    const testUserEmail = 'test@gmail.com';

    expect(generateTokenPromise(testUserEmail)).resolves.toBeDefined();

})

//o con async await

it('should generate a token value', async ()=>{

    const testUserEmail = 'test@gmail.com';

    const token = await generateTokenPromise(testUserEmail);

    expect(token).toBeDefined();

})

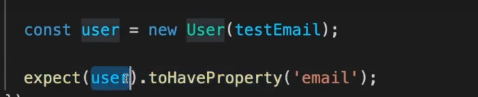
**Hooks:**

Hooks son funciones que son ejecutadas automáticamente, por el test runner en ciertos puntos de tiempo.

Se pueden también testear clases

Paréntesis:

El toHaveProperty sirve para saber si la clase tiene una propiedad



Cierro paréntesis

para no tener que andar escribiendo el mismo código un montón de veces.

Se puede asignar una constante global arriba de todo y usarla.

const testEmail = 'test@test.com';

Los test se ejecutan como código js, entonces si definis una variable global y después en un test ejecutas una función que modifica el valor, va a quedar con ese valor para los otros test, entonces puede estar haciendo que falle.

Eso lo podemos arreglar con los **hooks**

Dentro de los hooks tenemos

* AfterAll
* BeforeAll
* AfterEach
* BeforeEach

Estas se pueden ejecutar para registrar otras funciones antes de todos los test, después de todos, antes o después de uno en especifico.

Se definen arriba de todo antes de los test

beforeAll(() =>{  //Se ejecuta antes de todos los test

  console.log('beforeAll');

})

beforeEach(()=>{ //se ejecuta antes de cada test

  console.log('beforeEach');

})

afterEach(()=>{ //se ejecuta despues de cada test

  console.log('afterEach');

})

afterAll(()=>{ // se ejecuta despues de todos los test

  console.log('afterAll');

})

En vez de crear las variables globales, conviene crearlas sin definirle el valor y después inicializarlas en el beforeAll



**Concurrently tests:**

Se pueden ejecutar test en simultaneo o al mismo tiempo. X defecto se ejecuta uno abajo del otro.

Para hacer que tests se ejecuten en simultaneo se les agrega concurrent.

it.concurrent('should update the email', () => {

Entonces todos los que tengan ese concurrent se van a ejecutar al mismo tiempo y ahorrar tiempo.

Si se lo pones al describe todos los tests dentro se van a ejecutar al mismo tiempo.

describe.concurrent() // si se le agrega al describe todos los de adentro se ejecutan en simultaneo

**Spies And mocks:**

En este archivo importamos eso, y guardamos en data lo que devuelve la función generateReportData y en sus parámetros le pasamos el log que importamos, no estamos ejecutando log, solo lo pasamos como parámetro.

import { generateReportData, storeData } from './src/data.js';

import log from './src/util/logger.js';

const data = generateReportData(log);

storeData(data);

hay un problema x ejemplo si tu test llama a una función que tiene que crear un archivo, cada vez que se ejecute va a crear un archivo, y no se quiere eso , o por ejemplo si borra algún dato.

Punto aparte

**Spies:** Son wrappers o envolturas alrededor de mis funciones o remplazos vacíos , es para las funciones te permite trackearlas y saber como o cuando una función fue llamada.

**Mocks:** Un reemplazo para una API que puede proveer algún comportamiento especifico de test.

**Spies:**

**Para esta función**

export function generateReportData(logFn) {

  const data = 'Some dummy data for this demo app';

  if (logFn) {

    logFn(data);

  }

  return data;

}

**Este test**

    it('should execute logFn if provided', ()=>{

        const logger = vi.fn(); // fn crea una funcion vacia

        generateReportData(logger); // aca le mandamos la funcion

        expect(logger).toBeCalled(); // aca preguntamos si el logger fue llamado

    })

Se puede chequear no solo si fue llamado sino también en este caso que se haya llamado 2 veces



MocK:

El fs también se uso en el .js

import  {promises as fs} from 'fs';

vi.mock('fs'); //encuentra todas las funciones del modulo fs, y las reemplaza con funciones spy vacias

it('should execute the writeFile method', ()=>{

    const testData = 'test';

    const testFileName = 'test.txt';

    writeData(testData, testFileName);

    expect(fs.writeFile).toBeCalled();

    // return expect(writeData(testData, testFileName)).resolves.toBeUndefined(); este es viejo

})

Mas de mock

vi.mock('fs'); //encuentra todas las funciones del modulo fs, y las reemplaza con funciones spy vacias

vi.mock('path', () =>{ // se agrega un segundo parametro como funcion, para seleccionar solo el metodo join

    return {

        default:{

            join: (...args)=>{

                return args[args.length - 1]

            }

        }

    }

});

it('should execute the writeFile method', ()=>{

    const testData = 'test';

    const testFileName = 'test.txt';

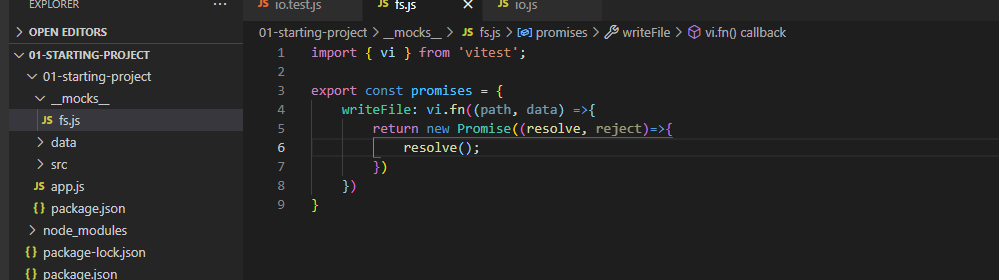
    writeData(testData, testFileName);

    expect(fs.writeFile).toBeCalledWith(testFileName,testData);

})

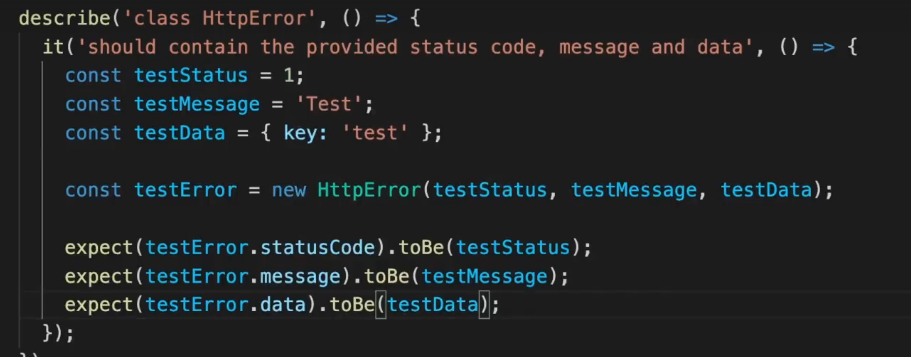
Para definir mocks globales podes crear una carpeta llamada \_\_mocks\_\_ en el root

Dentro creas archivos js con el nombre del modulo que quieras mockear, lo de adentro no entendí un pito



ME perdi

Para saber si los datos se están guardando bien en el constructor de una clase:

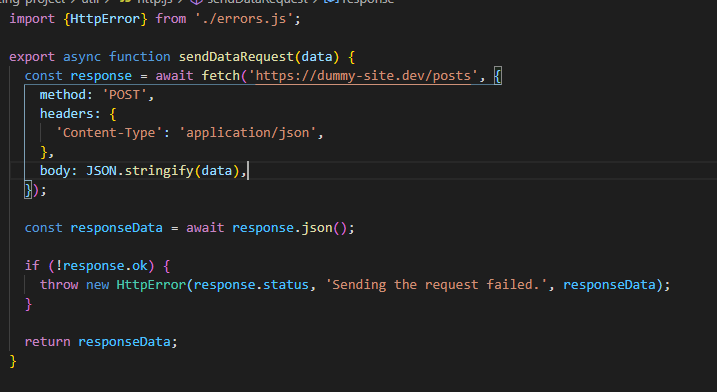


Que algo sea Undefined

toBeUndefined();

para reemplazar funciones globales como fetch con mock se usa vi.stubGlobal()

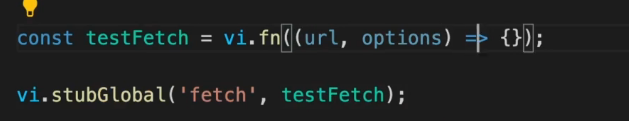
el js es este:



Entonces:



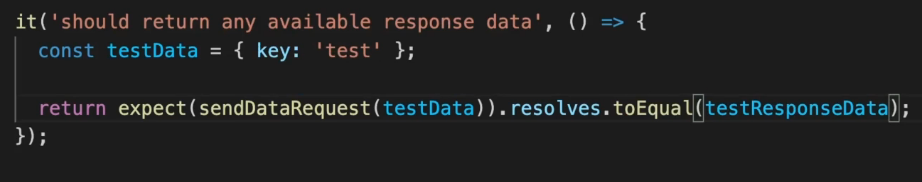
Se crea también un vi.fn y dentro se les pasa los parámetros del fetch



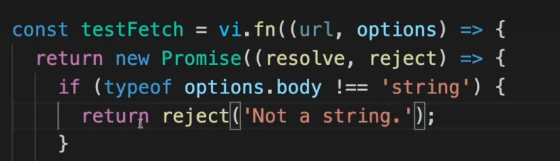
Fetch retorna una promesa. Dentro de la promesa resuelve un objeto ficticio

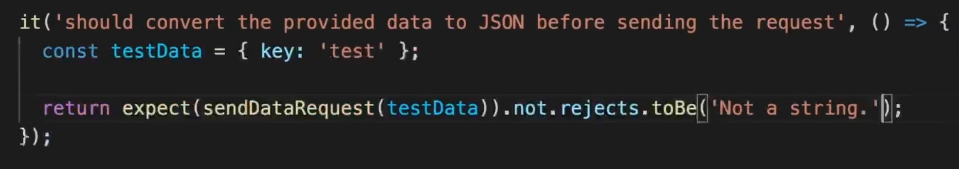
Al final queda así:



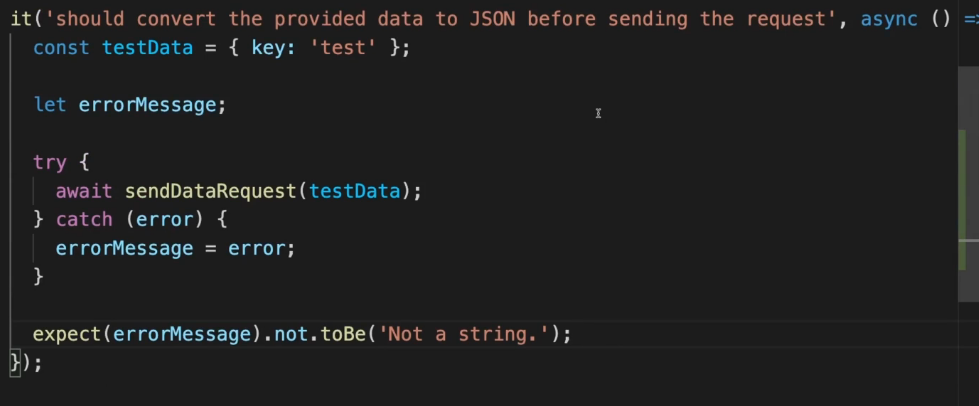


Antes de enviar un request convertir los datos a JSON, se envia por POST:

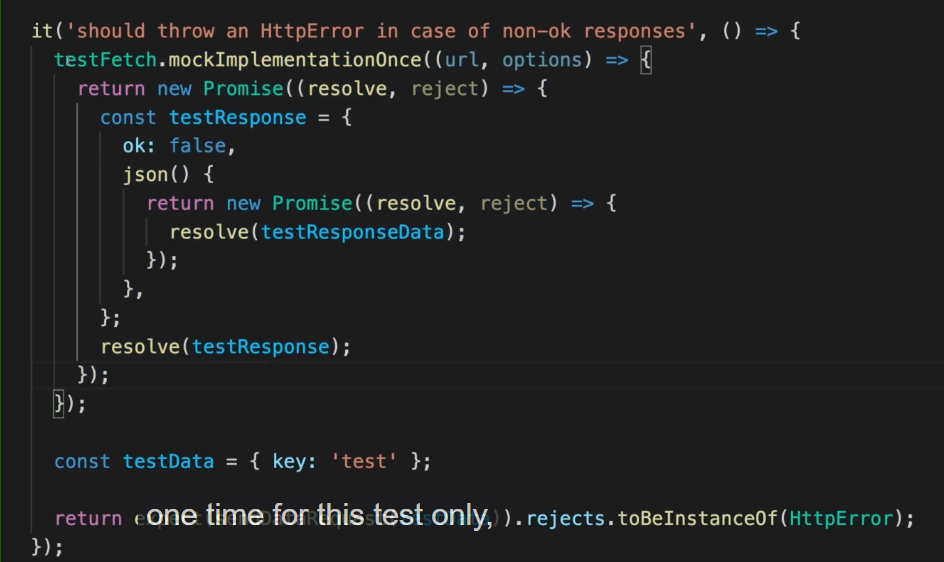
Arriba hace esta condición 



También se puede hacer con async await



Devolver http error si no es ok la respuesta:



**Frontend Dom testing:**

Te va a tirar error al no tener definido el document, como del document.querySelector. Entonces podemos hacer un mock

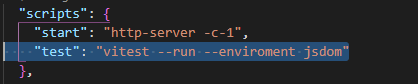
**Elegir entorno de desarrollo**

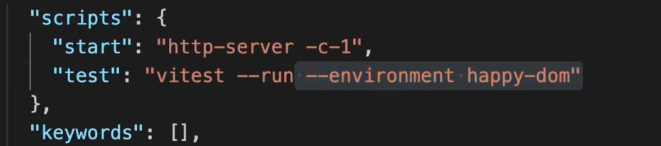
**JSDOM** simula que el código corre en el browser

**Happy-DOM** también simula



En test en el package.json se agrega esto para usar JSDOM



O 

**Setear una pagina html virtual:**

Con lo siguiente crear la pagina virtual

import { fs } from 'fs';

import { path } from 'path';

import { window } from 'happy-dom';

const htmlDocPath = path.join(process.cwd(), 'index.html');

const htmlDocumentContent = fs.readFileSync(htmlDocPath).toString();

const window = new Window();

const document = window.document;

document.write(htmlDocumentContent);

vi.stubGlobal('document', document);

chequear si se agrega el párrafo

it('should add an error paragraph to the id="errors" element', ()=>{

    showError('test');

    const errorsEl = document.getElementById('errors');

    const errorParagraph = errorsEl.firstElementChild;

    expect(errorParagraph).not.toBeNull();

})

Si ejecutas una sola vez el párrafo va a quedar guardado porque el document esta global, entonces si haces otro test hay que limpiarlo antes

Y lo dejamos así:

const window = new Window();

const document = window.document;

vi.stubGlobal('document', document);

beforeEach(()=>{

    document.body.innerHTML = '';

    document.write(htmlDocumentContent);

})

En total con los test queda así:

import { it, vi, expect, beforeEach } from 'vitest';

import { showError } from './dom';

import { fs } from 'fs';

import { path } from 'path';

import { window } from 'happy-dom';

const htmlDocPath = path.join(process.cwd(), 'index.html');

const htmlDocumentContent = fs.readFileSync(htmlDocPath).toString();

const window = new Window();

const document = window.document;

vi.stubGlobal('document', document);

beforeEach(()=>{

    document.body.innerHTML = '';

    document.write(htmlDocumentContent);

})

it('should add an error paragraph to the id="errors" element', ()=>{

    showError('test');

    const errorsEl = document.getElementById('errors');

    const errorParagraph = errorsEl.firstElementChild;

    expect(errorParagraph).not.toBeNull();

})

it('should not contain an error paragraph initially', ()=>{

    const errorsEl = document.getElementById('errors');

    const errorParagraph = errorsEl.firstElementChild;

    expect(errorParagraph).toBeNull();

})

it('should output the provided message in the error paragraph', ()=>{

    const testErrorMsg = 'test';

    showError(testErrorMsg);

    const errorsEl = document.getElementById('errors');

    const errorParagraph = errorsEl.firstElementChild;

    expect(errorParagraph.textContent).toBe(testErrorMsg);

})

Si tenes mucho frontend capaz se hace muy largo entonces podemos usar una librería,

**TESTING LIBRARY**

<https://testing-library.com/docs/vue-testing-library/examples>