pruebas unitarias y de integración

las pruebas de integración y unitarias consisten en crear un modelo de prueba que permita comprobar el funcionamiento de nuestra aplicación, con los test comprobaríamos que nuestro proyecto ejecute los resultados necesarios para que siempre del resultado esperado, en caso tal de que la aplicación no ejecute el resultado que esperemos el test programado dará la alerta de que el resultado no es el esperado.

Para esto crearíamos un archivo que termine en spect.js y tendría el siguiente condigo, por ejemplo

describe('Example Component', () => {

  test('Debe de ser mayor a 10', () => {

    let value = 5

    value = value + 2

    if (value > 10) {

      // TODO: todo bien

    } else {

      throw `${value} no es mayor a 10`

    }

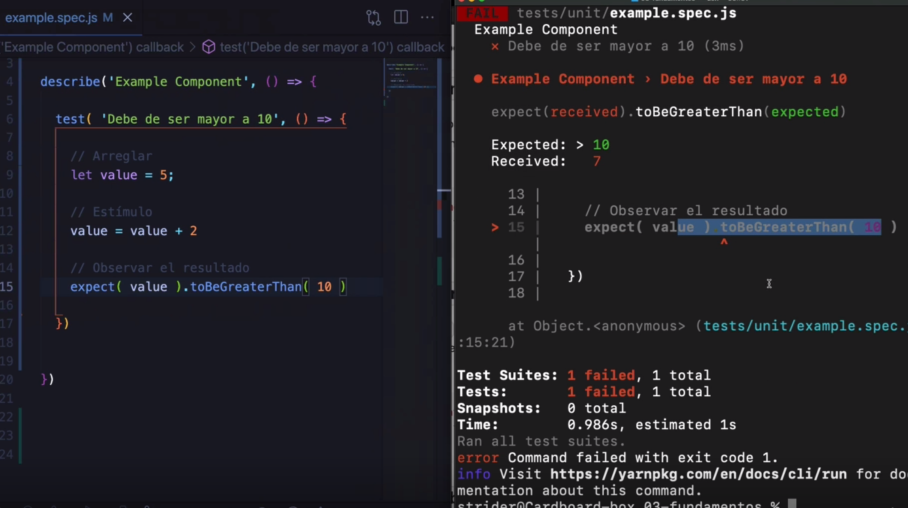
  })

})

Luego con ayuda de alguna librería como lo puede ser jest ejecutaríamos un comando que corra el test

-npm run test:unit

Y esto arroja el mensaje de advertencia si el resultado no es el esperado para manejar las respuestas y los errores manejaremos la exención el cual estaría entre retur (respuesta esperado) throw (excepción) pero con la ayuda de los métodos de la librería de jest podemos tener un resultado mas adecuado



Esto controlaría los resultados dando mensajes de lo que se esperaba y lo que se obtuvo

Snapshot  
haremos pruebas en counter.vue



Podemos hacer pruebas de muchos casos esto con la finalidad de que si alguna persona o unos mismo comete un error antes de pasarlo a producción el test debe de conprobar que todo funciona bien.

Recuerde que se debe de crear una carteta test que tenga los elementos que queremos conprobar con el mismo nombre con la diferencia que termina en counter.spect.js entonces en el caso de hacer un test de counter.vue se llamaría counter.spec.js.

La base de el código es

describe('Example Component', () => {

  test('Debe de ser mayor a 10', () => {

  })

})

A esto le agregaremos dentro de la función test la sentencia

expect(ALGO).toMatchShapshoot()

esta sentencia lo que hace es evaluar un componente y verificar que los datos no han sido cambiados.

Para ello vamos a importar el componente de la siguiente forma

import Counter from '@/components/Counter'

importaremos el counter con la ruta del componente haciendo referencia al punto de entrada de la aplicación, en el @ nos ahorraremos el tener que buscar el patch exacto del archivo después de esto usaremos la librería que nos servirá de shapshot

import {shallowMount, mount } from '@vue/test-utils'

ya con esto importado usaremos un dato de prueba

const wrapper = shallowMount ( Counter )

expect ( wrapper.html() ).toMatchSnapshot()

el wrapper.html tomara el html del componente una prueba html y creara un snapshot, el snapshot es un archivo llamado igual al componente a evaluar con la diferencia que le agregara una extensión mas en el nombre del archivo que será. snap. en este archivo vamos a tener la oportunidad de que el componente no pueda ser cambiado

leer props desde pruebas y test

podemos usar las porps para ver los datos que están llegando desde el componente al test.

weapper.props()

llamando el weapper.prpos podemos acceder a las propiedades, de esta forma trabajaríamos con las propiedades de distintas formas

console.log( weappeer.props )

o podemos almacenar el resultado en props

const start = wrapper.props('start')

vamos a lograr que se establezcan los valores por defecto

test('debe de establecer el valor por defecto', () => {

  const { start } = wrapper.props()

  const value = wrapper.find('[data-testid="counter"]').text(

    expect (Number(value)).toBe(start)

  )

})

Enviar props y evaluarlas

Haremos una prueba en donde enviaremos el props para luego evaluar estos datos

test('debe de mostrar la prop title', () => {

  const wrapper = shallowMount( Counter, {

    props:{

      title: 'Hola mundo'

    }

  })

  expect( wrapper.find( 'h2' ).text() ).toBe( 'Hola Mundo' )

})

De esta forma todo ocurre correctamente y se realiza el tes de los datos de forma correcta.

Vamos a crear un test para componente llamado indecisión.spec.js

import { shallowMount } from '@vue/test-utils'

import Indecision from '@/components/Indecision.vue'

describe('Indecision component', () => {

  let wrapper

  beforeEach( () => {

    wrapper = shallowMount( Indecision )

  })

  test('debe de hacer match con el snaphot', () => {

    expect( wrapper.html() ).toMatchSnapshot()

  })

})

En caso de que ocurra un cambio importante en indecisión.vue al correr el test nos arrojara un mensaje de error advirtiéndonos las fallas que cometimos

Spy y mocks

Es importante saber que el componente es físicamente usado por lo que todas las operaciones y consultas son utilizadas al realizar el test.

Suponiendo que queremos evaluar solo si una función o método se a ejecutado, para ello creamos una variable llamada clgSpy (espia)

beforeEach( () => {

    wrapper = shallowMount( indecision )

    clqSpy = jest.spyOn( console , 'log')

})

La variable creada va a evaluar en este caso que esta función ando un console.log correctamente, ahora tenemos una función espía que puede esperar que el consolé log halla sido llamado por lo menos una vez, podemos establecer cuantas veces hallan sido llamadas condicionando por lo menos un mínimo de veces para llamar.

expect( clgSpy ).toHaveBeenCalledTimes(2)

vamos a ver el caso en donde no haya sido disparado.

Esto es mediante se ejecuta mediante

Const getAnwerSpy = jest.spyOn( wrapper.vm, ‘getAnswer’)

expect( clgSpy ).toHaveBeenCalledTimes(2)

expect( getAnswerSpy ).not.toHaveBeenCalled()

estos dos métodos son utilizados para hacer esto con el que controlaremos el número de veces de se debería de ejecutar una función

para limpiar la consola tenemos en jest el siguiente comando que podemos ejecutar en el código de spec.js

jestConfig.clearAllMocks()

tenemos que aclarar que fetch no esta incluido en jest por lo que tenemos que hacer un moc dentro del objeto global de Windows. Fetch

podemos hacer entonces lo siguiente en caso de recibir un error en el fetch al cargar una imagen

en el componente

async getAnswer(){

    try {

        this.answer = 'Pensado...'

        const { answer, image } = await fetch('https://yesno.wft/api')

        this.answer  = answer === 'yes' ? 'Si!' : 'No!'

        this.img = image

    }

    catch (error) {

        this.answer = 'No se pudo cargar del API'

        this.img = null

    }

En el spec.js

fetch.mockImplementationOnce( () => Promise.reject('API is down'))

await wrapper.vm.getAnswer()

const img = wrapper.find('img')

expect( img.exists()).toBeFalsy()

expect(wrapper.vm.answer ).toBe('No se pudo cargar del API')