

## EXERCISES QUE TRABAJAREMOS EN LA CUE

- EXERCISE 1: CORRIENDO EL PRIMER TEST DE UN COMPONENTE.
- EXERCISE 2: CREANDO EL PRIMER TEST.
- EXERCISE 3: USANDO UN MOCK.

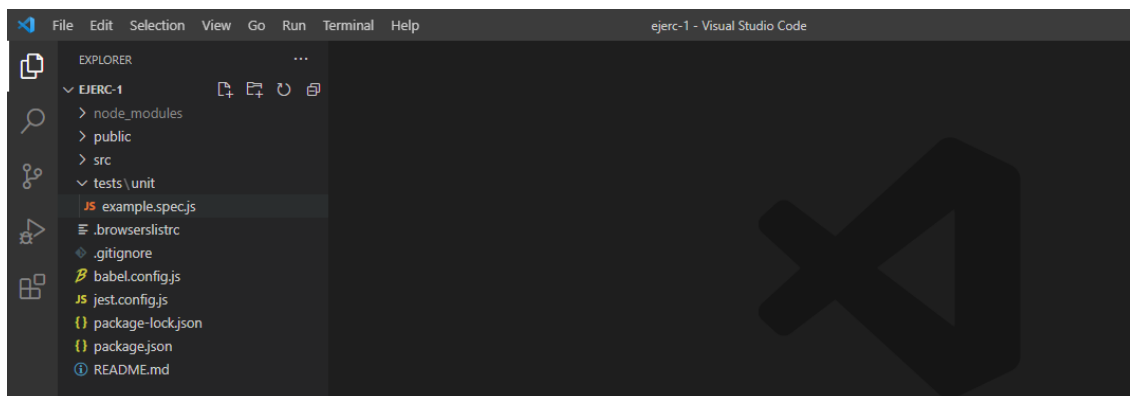
## EXERCISE 1: CORRIENDO EL PRIMER TEST DE UN COMPONENTE

### INTRODUCCIÓN

Para comenzar a testear, podemos crear un proyecto nuevo, o utilizar el del CUE anterior. Al iniciar un proyecto con **Vue-Test-Utils**, éste instala una prueba preconstruida, por lo que vamos a revisar dicha prueba, su estructura y como ejecutarla.

### REVISANDO LA PRUEBA

Empezaremos revisando la carpeta de test, que es donde se almacenan nuestras pruebas unitarias. Ahí encontraremos el ejemplo, y lo vamos a revisar rápidamente.



Durante una prueba de componentes, éste no puede ser montado en su entorno real. Sin embargo, necesitamos hacerlo. **Vue-test-utils** permite renderizar en memoria un componente, importándolo junto con el que se desea someter a prueba, usando:

- **Mount**: creará un envoltorio alrededor del componente e intentará montarlo, junto con sus dependencias internas, y sus componentes hijos.
- **shallowMount**: creará un envoltorio alrededor del componente, y lo montará sin sus componentes hijos, pero creará **stubs** de todos ellos.

En ambos casos, cuando el componente es montado, éste nos devuelve un **wrapper**.

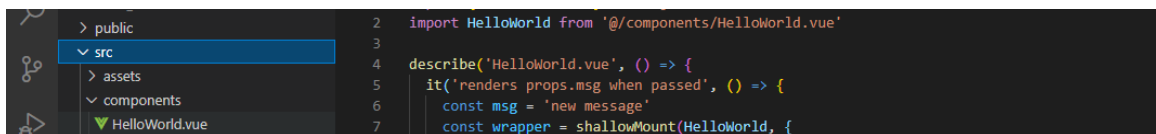
Wrapper (envoltorio): es un objeto que contiene el componente de **Vue**, además de algunos métodos para realizar test. Los métodos: **mount**, **shallow**, **find** y **findAll**, lo emplean para montar el componente.

En la línea 1, encontraremos la importación de la función **shallowMount** desde **Vue-Test-Utils**, que nos permite montar nuestro componente para realizar la prueba.



```
JS example.spec.js x
tests > unit > JS example.spec.js > ...
1 import { shallowMount } from '@vue/test-utils'
```

En la línea 2, encontraremos la importación del componente **HelloWorld.vue** desde las carpetas **"src">"components"**.



```
> public
  > src
    > assets
    > components
      > HelloWorld.vue

2 import HelloWorld from '@components/HelloWorld.vue'
3
4 describe('HelloWorld.vue', () => {
5   it('renders props.msg when passed', () => {
6     const msg = 'new message'
7     const wrapper = shallowMount(HelloWorld, {
```

En la línea 4, podemos ver la palabra “describe”. Este es un método de **Jest**, que permite crear un bloque para agrupar varias pruebas relacionadas. Recibe como parámetros un **String** y una función.

En este caso, el **String** corresponde a una mención del componente que vamos a probar.

```
1 import HelloWorld from './components/HelloWorld.vue'
2
3
4 describe('HelloWorld.vue', () => {
5   it('renders props.msg when passed', () => {
```

En la siguiente, encontramos la palabra “it”, variable proveniente de **Jest**, que permite crear una prueba unitaria. Recibe dos parámetros, un **string**, que corresponde al nombre del test, y la función de la prueba. En este caso, el **string** será una descripción de lo que hará la prueba, que es renderizar la **props** del mensaje cuando se le entregue.

```
5   it('renders props.msg when passed', () => {
6     const msg = 'new message'
```

En la línea 6, encontraremos una constante, llamada **msg**, con el **string** “new message”.

```
6     const msg = 'new message'
```

En la línea 7, veremos otra constante, llamada **wrapper**, que corresponderá al montaje, a través de **shallowMount**, del componente **HelloWorld**.

```
7     const wrapper = shallowMount(HelloWorld, {
8       propsData: { msg }
```

En la siguiente, se declara el uso de **propsData** con **{msg}** como contenido. Éste entrega **props** a una instancia **Vue** durante su creación, y está destinado principalmente a facilitar las pruebas unitarias.

```
7   const wrapper = shallowMount(HelloWorld, {  
8     propsData: { msg }  
9   })
```

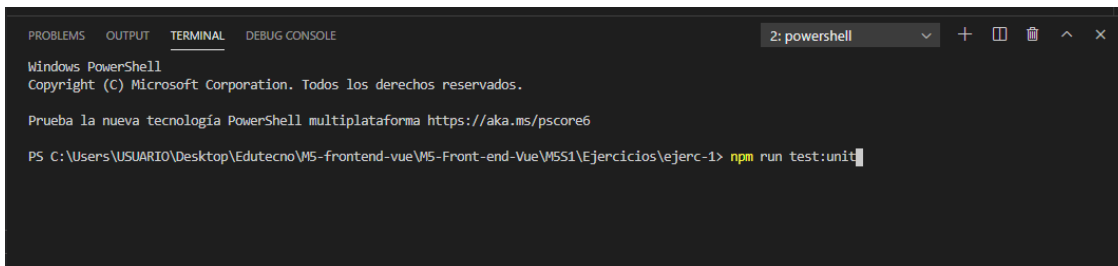
Luego, encontramos el llamado a la función **“expect”** de **Jest**. Ésta nos permite comprobar que los valores cumplen determinadas condiciones, y también da acceso a una serie de "comparadores" que permiten validar diferentes cosas. En este caso, se usa para verificar que el texto que contiene la constante **wrapper**, proveniente del texto contenido en el componente **HelloWorld**, coincida con el contenido en la constante **msg**. Se logra usando el método **“toMatch”** de **Jest**, que nos da la posibilidad de comparar un **string** contra una expresión regular, u otro **string**.

```
9   })  
10  expect(wrapper.text()).toMatch(msg)  
11  })
```

Por último, cuando nos pregunte si queremos guardar esta configuración para futuros proyectos, le pondremos **"N"**, y presionamos "enter".

## Correr la prueba

Como la prueba ya está escrita, vamos a correrla. Para ello, abrimos una nueva terminal, escribimos **“npm run test:unit”**, y presionamos "enter".



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  
2: powershell  
Windows PowerShell  
Copyright (C) Microsoft Corporation. Todos los derechos reservados.  
  
Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6  
  
PS C:\Users\USUARIO\Desktop\Edutecno\W5-Frontend-vue\W5-Front-end-Vue\W551\Ejercicios\ejerc-1> npm run test:unit
```

Mientras se ejecuta la prueba, veremos esto:



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 2: node
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\USUARIO\Desktop\Edutecno\MS-frontend-vue\MS-Front-end-Vue\M5S1\Ejercicios\ejerc-1> npm run test:unit

> ejerc-1@0.1.0 test:unit C:\Users\USUARIO\Desktop\Edutecno\MS-frontend-vue\MS-Front-end-Vue\M5S1\Ejercicios\ejerc-1
> vue-cli-service test:unit

RUNS tests/unit/example.spec.js
```

Cuando ha terminado de ejecutarse, podemos ver que la prueba **“renders props.msg when passed”**, aplicada en el componente **HelloWorld.vue**, pasó el test.

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 2: powershell
> vue-cli-service test:unit

PASS tests/unit/example.spec.js
  HelloWorld.vue
    ✓ renders props.msg when passed (43ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 3.438s
Ran all test suites.
PS C:\Users\USUARIO\Desktop\Edutecno\MS-frontend-vue\MS-Front-end-Vue\M5S1\Ejercicios\ejerc-1>
```

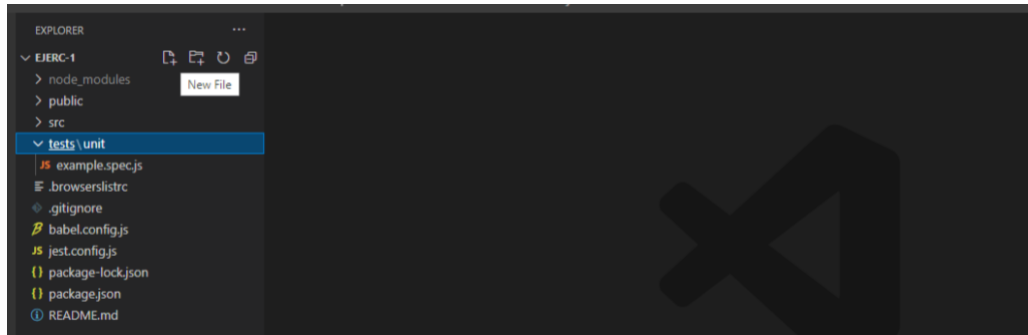
## EXERCISE 2: CREANDO EL PRIMER TEST

### INTRODUCCIÓN

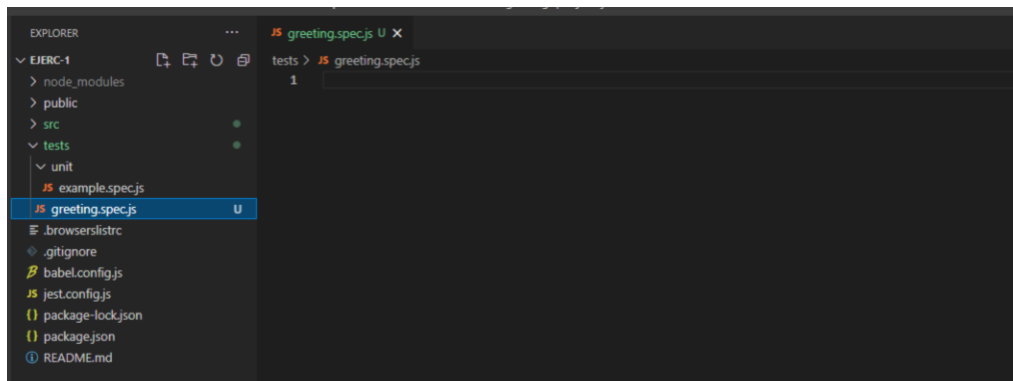
Ya que hemos analizado la estructura de una prueba hecha con **Jest** y **Vue-Test-Utils**, y se ha ejecutado, crearemos nuestra primera prueba unitaria. El objetivo será realizar una prueba sobre un componente que renderiza un saludo.

### CREANDO LA PRUEBA

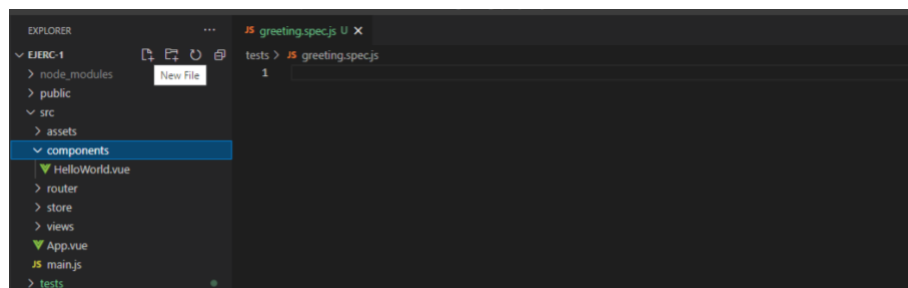
Para comenzar nuestro test, lo primero que haremos será crear el archivo dentro de la carpeta **“test/unit”**.



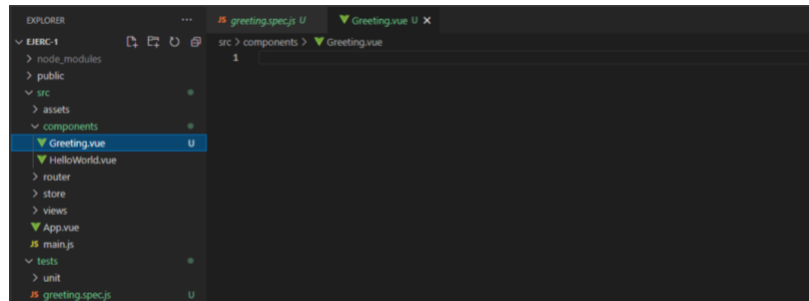
Lo llamaremos: **greeting.spec.js**.



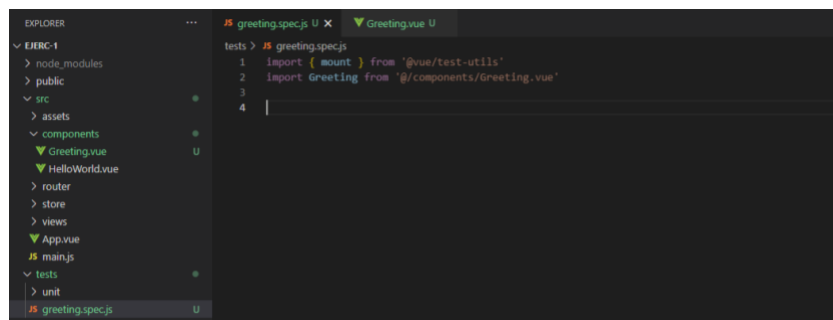
Lo siguiente que haremos, es ir a la carpeta **"src" > "components"**, y crear un nuevo archivo.



Será un componente llamado **Greeting.vue**.



Una vez creado el archivo, vamos a importar el método **"mount"** desde **Vue-Test-Utils**, para montar nuestro componente, y de inmediato importaremos **Greeting.vue** desde los componentes.



Lo siguiente que haremos, es usar la sintaxis de **Jest** para crear nuestra prueba. Usaremos **"describe"**, que según vimos en el primer ejercicio, nos permitía agrupar varios test unitarios relacionados, y facilitar la creación de jerarquías de pruebas. Le agregamos los parámetros, comenzando por el **string** con el nombre del componente que estamos testeando, y declaramos la función. Luego, creamos la prueba usando la variable **"it"**, le asignamos el nombre como un **string** **"renders a greeting"**, y continuamos con la función.

```

tests > JS greeting.spec.js > describe('Greeting.vue') callback > it('renders a greeting') callback
1 import { mount } from '@vue/test-utils'
2 import Greeting from '@/components/Greeting.vue'
3
4 describe('Greeting.vue', () => {
5   it('renders a greeting', () => {
6   })
7 })
8
9

```

Ahora, vamos a hacer el renderizado del componente usando **“mount”**. Una buena práctica es declarar una variable llamada **“wrapper”**, a la cual se le asigna el montaje.

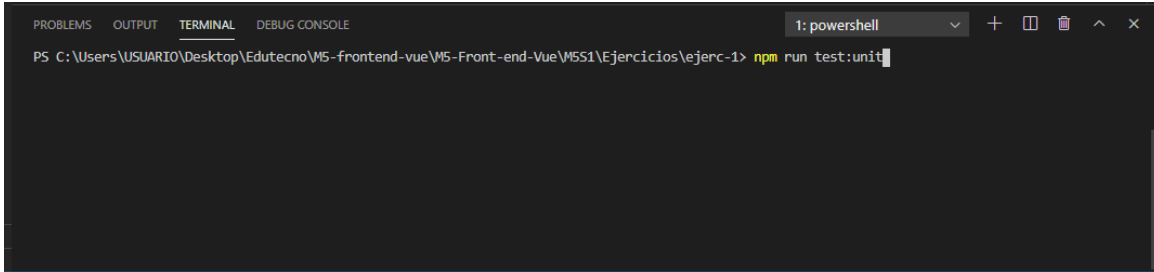
```

tests > JS greeting.spec.js > describe('Greeting.vue') callback > it('renders a greeting') callback
1 import { mount } from '@vue/test-utils'
2 import Greeting from '@/components/Greeting.vue'
3
4 describe('Greeting.vue', () => {
5   it('renders a greeting', () => {
6     const wrapper = mount(Greeting)
7   })
8 })
9
10
11
12

```

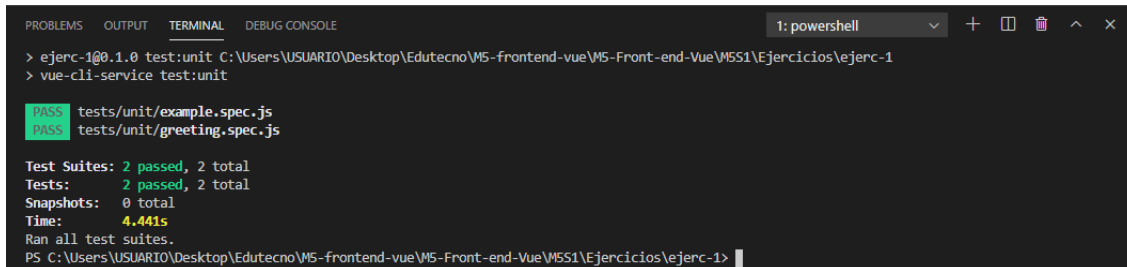
De inmediato corremos las pruebas, escribiendo en la terminal el comando: **“npm run test:unit”**, y presionamos “enter”.





```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 1: powershell
PS C:\Users\USUARIO\Desktop\Edutecno\W5-frontend-vue\W5-Front-end-Vue\W5S1\Ejercicios\ejerc-1> npm run test:unit
```

Como ya lo habíamos mencionado con anterioridad, **Jest** corre todas las pruebas existentes al mismo tiempo, siempre y cuando los archivos terminen con **“.spect.js”**, y que se encuentren en la carpeta **“test/unit”**. Podemos observar que ambas pruebas pasan, pero la que está escrita de tal forma, jamás va a fallar.



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 1: powershell
> ejerc-1@0.1.0 test:unit C:\Users\USUARIO\Desktop\Edutecno\W5-frontend-vue\W5-Front-end-Vue\W5S1\Ejercicios\ejerc-1
> vue-cli-service test:unit

PASS tests/unit/example.spec.js
PASS tests/unit/greeting.spec.js

Test Suites: 2 passed, 2 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 4.441s
Ran all test suites.
PS C:\Users\USUARIO\Desktop\Edutecno\W5-frontend-vue\W5-Front-end-Vue\W5S1\Ejercicios\ejerc-1>
```

Como un test que nunca falla no es útil, lo vamos a mejorar. Para eso, crearemos una aserción con el comportamiento deseado del componente, para confirmar. Usaremos **“expect”**. La sintaxis para dicha aserción, es algo como esto:

```

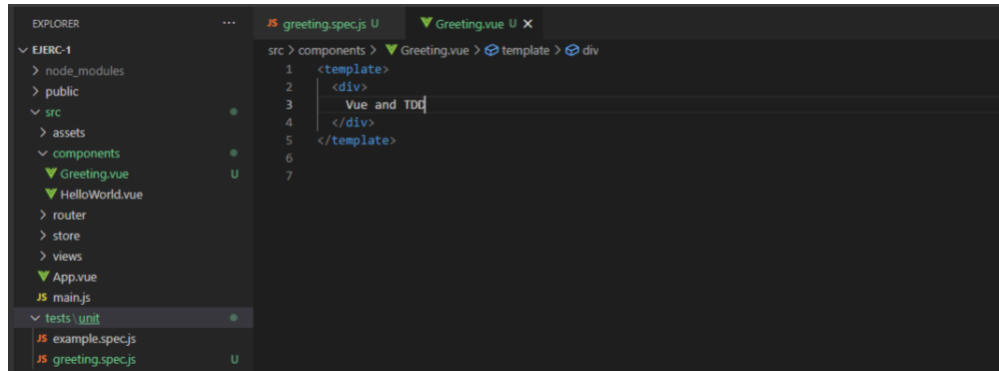
tests > JS greeting.spec.js > describe('Greeting.vue') callback
1  import { mount } from '@vue/test-utils'
2  import Greeting from '@/components/Greeting.vue'
3
4  describe('Greeting.vue', () => {
5    it('renders a greeting', () => {
6      const wrapper = mount(Greeting)
7
8      expect(received).toMatch(expected)
9    })
10 })
11
12
  
```

“Received” debe ser reemplazado con el contenido del componente, así que para poder visualizarlo, borramos temporalmente la aserción. Probamos escribiendo: `console.log(wrapper.html())`:

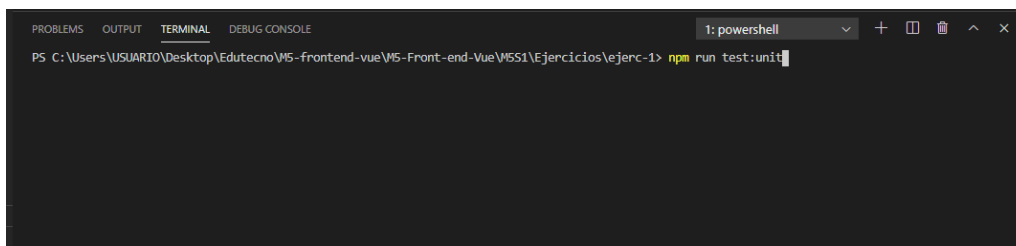
```

tests > JS greeting.spec.js > ...
1  import { mount } from '@vue/test-utils'
2  import Greeting from '@/components/Greeting.vue'
3
4  describe('Greeting.vue', () => {
5    it('renders a greeting', () => {
6      const wrapper = mount(Greeting)
7
8      console.log(wrapper.html())
9    })
10 })
11
12
  
```

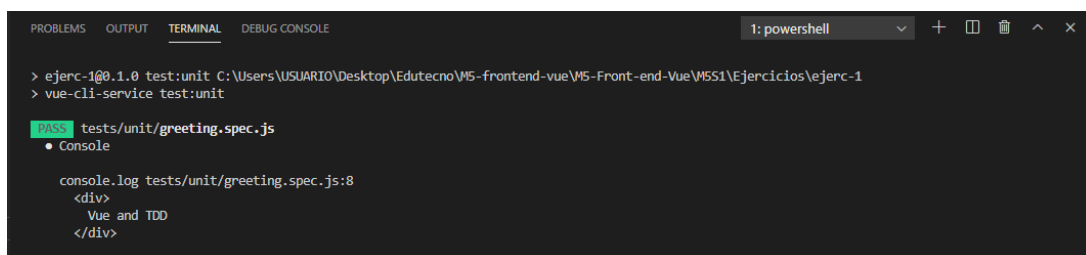
Pero para que funcione, y podamos visualizar algo del contenido del componente, debemos agregarlo. Comencemos por escribir el `template`:



Ahora, podemos correr la prueba usando: **"npm run test:unit"**, y presionamos "enter":

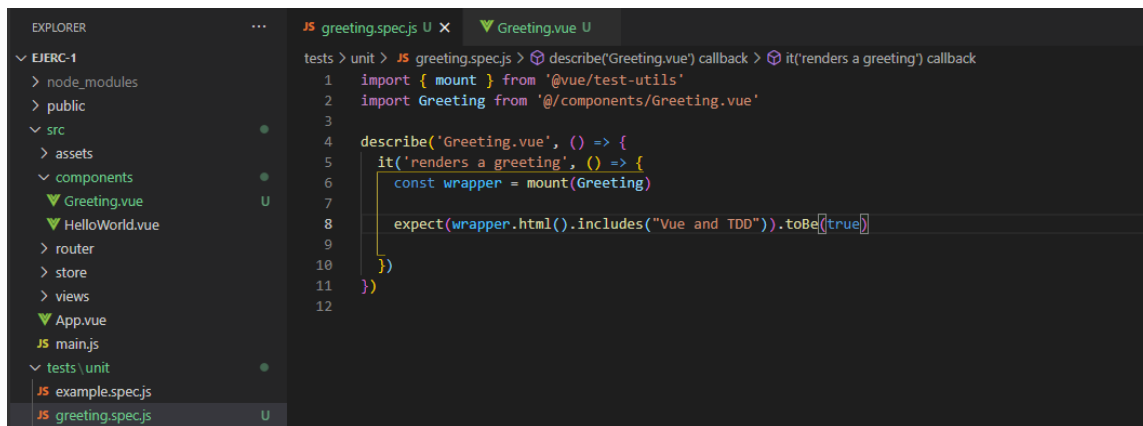


Antes de ver el resultado de la prueba, la terminal nos muestra como se vería la consola:



Luego vemos el resultado de las pruebas, pero como aún no hemos creado la aserción, no lo tomaremos en consideración. En este punto, lo que tiene importancia, es que a través de **"wrapper.html()"**, podemos visualizar todo el contenido del componente, que nos sirve para construir nuestra aserción. Usando: **"wrapper.html().includes("Vue and TDD")"**,

describiremos lo que queremos que encuentre. Entonces, comenzamos a escribir la aserción usando la función **“expect”**, como referencia a la expectativa de que esperamos que el resultado del contenido del componente, coincida con el valor que nosotros configuraremos. Para comparar valores y objetos, de diferentes formas, es que usaremos un método o función **“Matcher”**. **Vue-Test-Utils** no incluye ninguno, por eso, cuando queramos conocer todas las opciones disponibles, nos dirigiremos a la documentación de Jest: <https://jestjs.io/docs/expect>. Seleccionaremos un **“matcher”**, que nos permita comparar el contenido de **“result”** con el valor “actual”. En primera instancia, podemos usar **“toBe”**, que comparará valores primitivos o verificará la identidad referencial de instancias de objetos, esperando un valor booleano.



```

EXPLORER
  EJERC-1
    > node_modules
    > public
    > src
      > assets
      > components
        Greeting.vue
        HelloWorld.vue
      > router
      > store
      > views
      App.vue
      JS main.js
    > tests \ unit
      JS example.spec.js
      JS greeting.spec.js

tests > unit > JS greeting.spec.js > describe('Greeting.vue') callback > it('renders a greeting') callback
1  import { mount } from '@vue/test-utils'
2  import Greeting from '@components/Greeting.vue'
3
4  describe('Greeting.vue', () => {
5    it('renders a greeting', () => {
6      const wrapper = mount(Greeting)
7
8      expect(wrapper.html().includes("Vue and TDD")).toBe(true)
9
10   })
11 })
12
  
```

Sin embargo, **Vue-test-Utils** nos provee de un método para lograr obtener el contenido del componente, a través de **“wrapper.text”**, con lo que podremos obtener el texto de manera más simple, pero eso nos obligará a cambiar nuestro **“matcher”**. Usaremos ahora **“toMatch”**, que lo conocimos en el ejercicio 1, y le daremos el parámetro del mensaje.

```

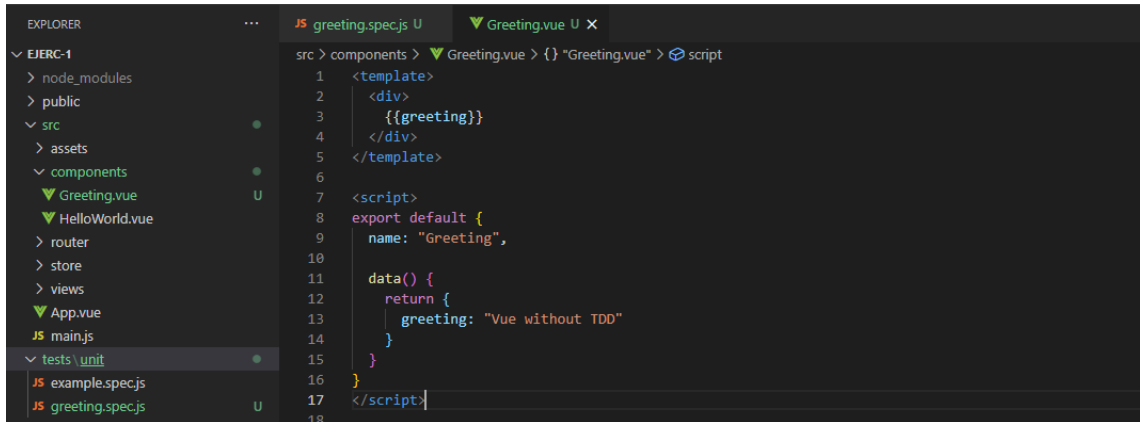
tests > unit > JS greeting.spec.js > ...
1  import { mount } from '@vue/test-utils'
2  import Greeting from '@/components/Greeting.vue'
3
4  describe('Greeting.vue', () => {
5    it('renders a greeting', () => {
6      const wrapper = mount(Greeting)
7
8      expect(wrapper.text()).toMatch("Vue and TDD")
9    })
10 })
11
12
  
```

Volvemos a escribir el comando:

```

PS C:\Users\USUARIO\Desktop\Edutecno\W5-frontend-vue\W5-Front-end-Vue\W51\Ejercicios\ejerc-1> npm run test:unit
  
```

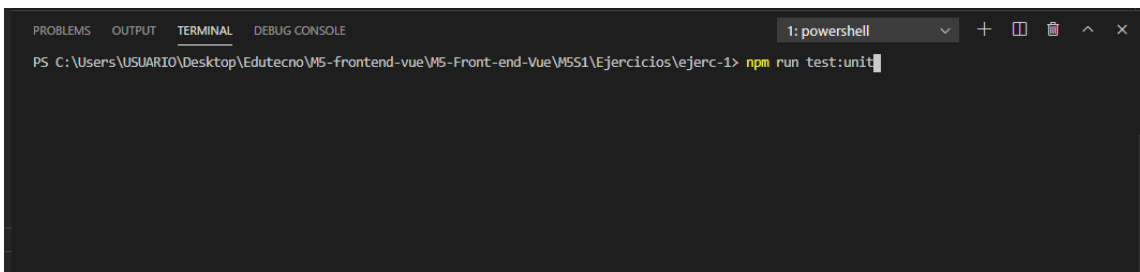
Presionamos “enter”. Nuevamente vemos que el test funciona, pero eso no nos garantiza todavía que el componente lo esté haciendo correctamente. Una de las reglas de construcción de pruebas en **TDD**, es que necesitan fallar al principio, y usar el informe de errores para guiar la corrección del código. Como último paso, y para asegurarnos de que la prueba realmente funcione, terminaremos de construir el componente **Greeting.vue** con una falla a propósito:



The screenshot shows the VS Code interface. On the left, the Explorer panel displays the project structure: EJERC-1, node\_modules, public, src, assets, components, Greeting.vue, HelloWorld.vue, router, store, views, App.vue, main.js, tests, unit, example.spec.js, and greeting.spec.js. The main editor shows the Greeting.vue component with the following code:

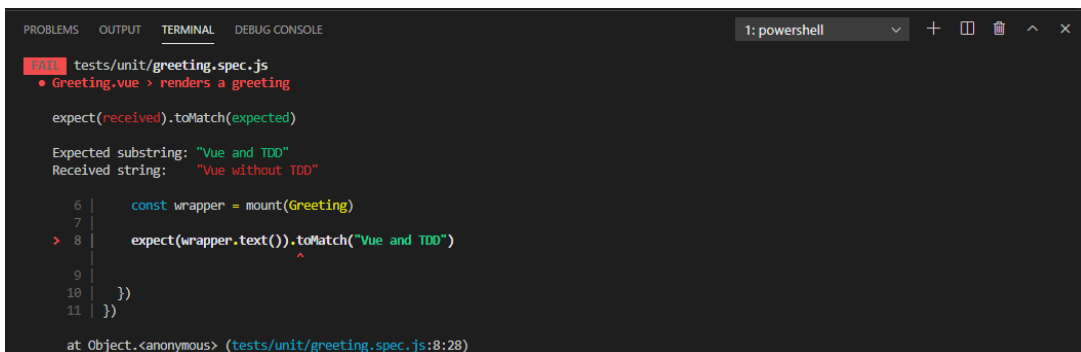
```
1 <template>
2   <div>
3     {{greeting}}
4   </div>
5 </template>
6
7 <script>
8   export default {
9     name: "Greeting",
10
11     data() {
12       return {
13         greeting: "Vue without TDD"
14       }
15     }
16   }
17 </script>
```

Este componente realiza solo una tarea: renderizar el valor de **greeting** que pusimos en data. Escribimos el comando para correr la prueba.



The screenshot shows a PowerShell terminal window with the command `npm run test:unit` entered at the prompt. The terminal title bar indicates it is a PowerShell window.

Y, después de presionar “enter”, podemos ver:



The screenshot shows the output of the `npm run test:unit` command in a PowerShell terminal window. The output indicates a failure in the unit test for `greeting.spec.js`. The test expects the rendered string to be "Vue and TDD", but the actual rendered string is "Vue without TDD".

```
FAIL tests/unit/greeting.spec.js
  • Greeting.vue > renders a greeting

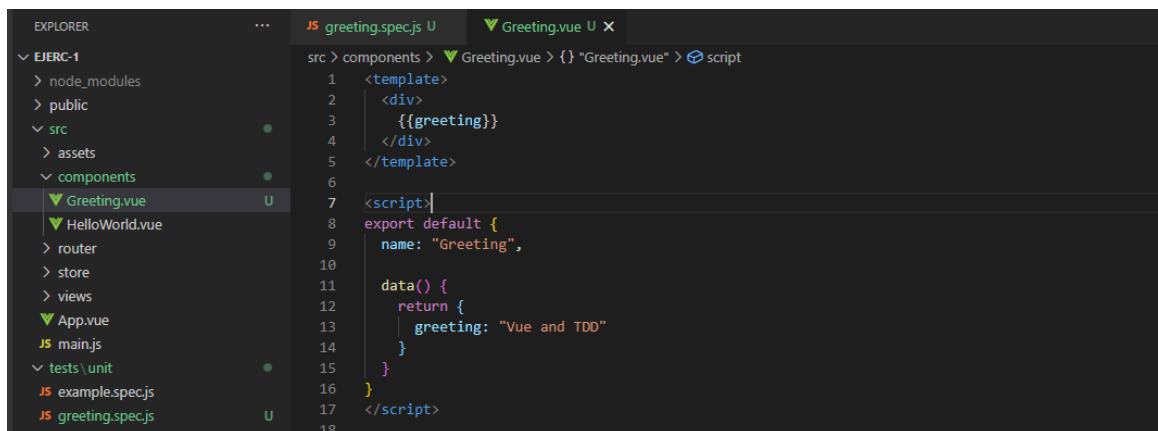
    expect(received).toMatch(expected)

    Expected substring: "Vue and TDD"
    Received string:    "Vue without TDD"

      6 |     const wrapper = mount(Greeting)
      7 |
    >  8 |     expect(wrapper.text()).toMatch("Vue and TDD")
        |                               ^
      9 |
     10 |   })
     11 | })

at Object.<anonymous> (tests/unit/greeting.spec.js:8:28)
```

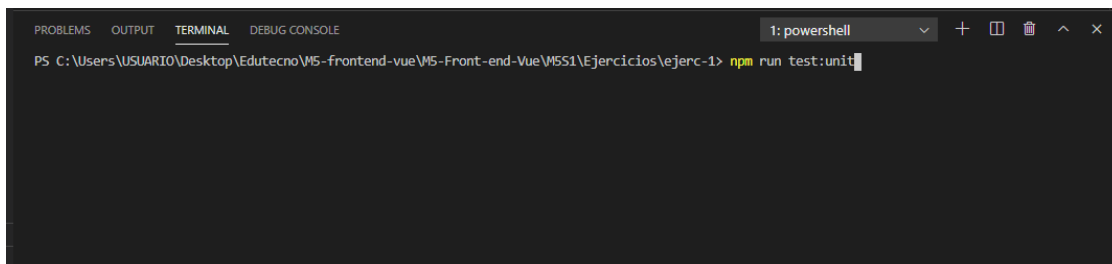
Esta vez el mensaje es diferente, y nos indica que la prueba ubicada en **"test/unit/greeting.spec.js"** ha fallado. Seguidamente, indica un esquema general de la aserción, e inmediatamente nos muestra el valor del **string** esperado, y el **string** recibido. Antes de terminar, nos señala la línea dentro del archivo en la que se encuentra la aserción de la prueba fallida, y finalmente, nos indica la línea y el número del carácter donde está lo que dio origen al error. En este caso, corresponde a que el método **"toMatch"** no logra cumplir la expectativa por la diferencia entre el mensaje recibido, y el esperado. Como podemos ver, **Jest** nos entrega una retroalimentación muy clara. Con eso ya podemos cambiar el componente para que pase la prueba.



The screenshot shows the VS Code interface with the Explorer on the left and the Editor on the right. The Explorer shows a project structure with a 'tests/unit' directory containing 'greeting.spec.js'. The Editor shows the 'Greeting.vue' component with the following code:

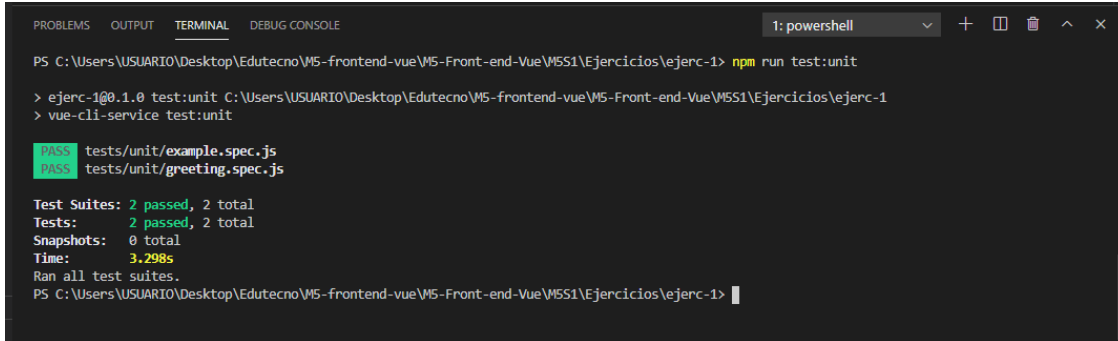
```
1 <template>
2   <div>
3     {{greeting}}
4   </div>
5 </template>
6
7 <script>
8   export default {
9     name: "Greeting",
10
11     data() {
12       return {
13         greeting: "Vue and TDD"
14       }
15     }
16   }
17 </script>
18
```

Y volvemos a correr la prueba.



The screenshot shows a terminal window with the command `npm run test:unit` being executed. The terminal output is currently empty, indicating that the tests are running.

Ahora, confirmamos que la prueba si corre de manera correcta.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
1: powershell

PS C:\Users\USUARIO\Desktop\Edutecno\MS-frontend-vue\MS1-Ejercicios\ejerc-1> npm run test:unit

> ejerc-1@0.1.0 test:unit C:\Users\USUARIO\Desktop\Edutecno\MS-frontend-vue\MS1-Ejercicios\ejerc-1
> vue-cli-service test:unit

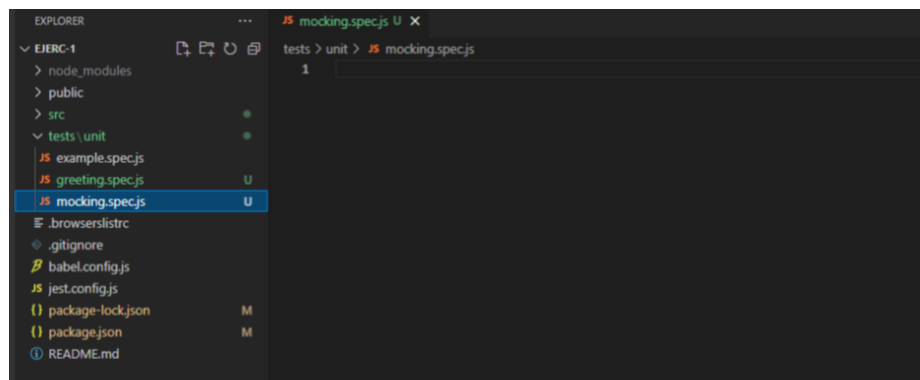
PASS tests/unit/example.spec.js
PASS tests/unit/greeting.spec.js

Test Suites: 2 passed, 2 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 3.298s
Ran all test suites.
PS C:\Users\USUARIO\Desktop\Edutecno\MS-frontend-vue\MS1-Ejercicios\ejerc-1>
```

Con esto, finalmente hemos terminado de crearla.

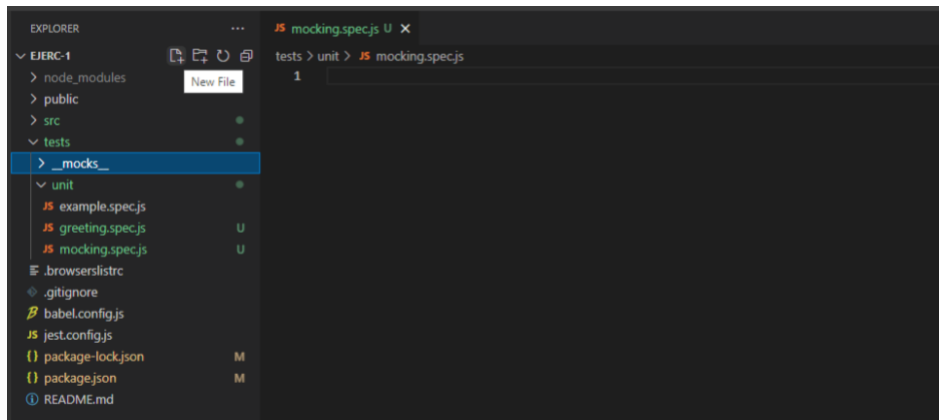
## EXERCISE 3: USANDO UN MOCK

Ya que hemos creado la estructura de una prueba, hecha con **Jest** y **Vue-Test-Utils**, y se ha ejecutado, desarrollaremos un test usando un objeto simulado, en este caso, un **mock** para imitar el uso de Axios, en una llamada a una **API**. Para comenzar nuestro test, lo primero que haremos será crear el archivo de la prueba dentro de la carpeta **“test/unit”**. Lo llamaremos: **“mocking.spec.js”**

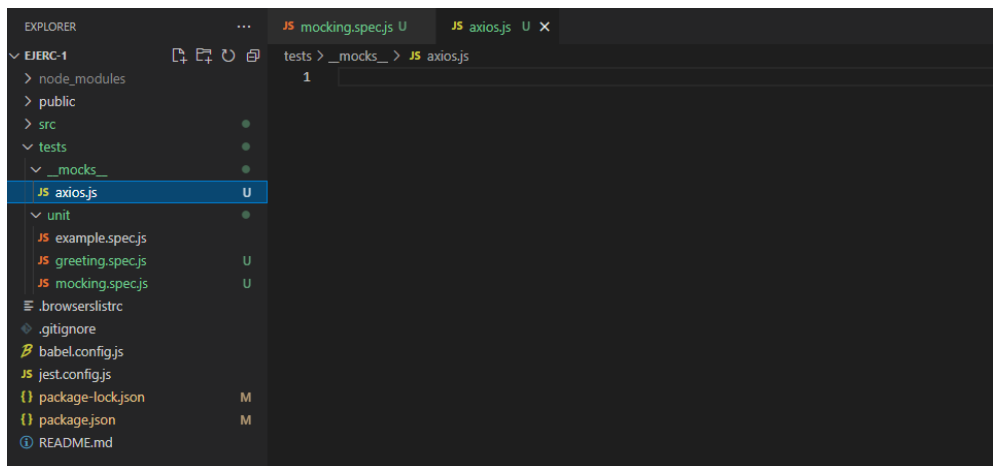




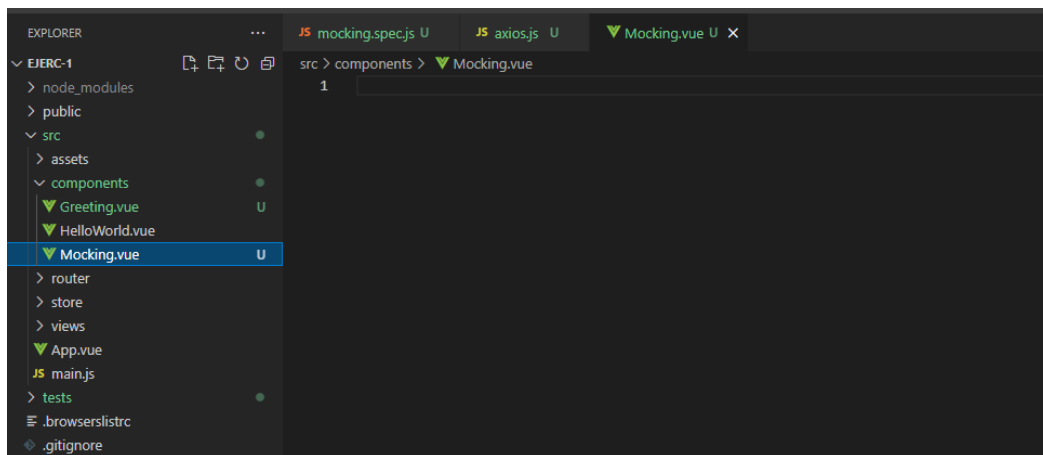
Lo siguiente que haremos es ir a **“test”**, y crear una nueva carpeta, a la que llamaremos: **“\_\_mocks\_\_”**.



En su interior, crearemos un nuevo archivo, al que llamaremos: **“axios.js”**.



De inmediato iremos a la carpeta `src > components`, y crearemos un nuevo componente llamado: `Mocking.vue`.



Con todo eso listo, vamos a regresar a la prueba, y escribimos: `import { shallowMount } from '@vue/test-utils'`, para importar el método que nos permita montar nuestro componente, y en la línea que sigue, escribimos. `import Mocking from '@/components/Mocking.vue'`, para importar el componente `Mocking.vue`.

```
1 test/unit/mocking.spec.js
2
3 import {shallowMount} from '@vue/test-utils'
4 import Mocking from '@/components/Mocking.vue'
```

Para continuar, en la siguiente línea escribimos: `jest.mock('axios')`.

```
1 jest.mock('axios')
```

Las funciones `mock` de `Jest`, nos permiten testear los enlaces, por ejemplo: de llamada a una `API`, borrando la implementación real en el código; capturando las llamadas a la función (junto con los parámetros entregados); y permitiendo configurar el tiempo de prueba de los valores retornados.

Hay dos formas de usar funciones **mock**: ya sea creando una función **mock** para usar en el código de prueba, o escribiendo un **mock** manual para emular la dependencia de un módulo, que es lo que acabamos de crear.

Ahora, nos dirigimos a `"test"- "__mock__"- "axios.js"`, y escribimos:

```
1 export default {
2   get: () => new Promise(resolve => {
3     resolve({ data: 'value' })
4   })
5 }
```

Este **mock** nos retornará, a través de una promesa, un valor `"data:value"` que pasará a la `"data"` de nuestro componente. Con esto, reemplazamos la llamada a la API que pondremos en `"methods"` del mismo componente.

Lo siguiente que haremos, es usar la sintaxis de **Jest** para crear nuestra prueba. Utilizaremos `"describe"`, que según vimos en el primer ejercicio, nos permitía agrupar varios test unitarios relacionados, y facilitar la creación de jerarquías de pruebas. Le agregaremos los parámetros, comenzando por el **string** con el nombre del componente que estamos testeando, y declaramos la función, escribiendo inmediatamente debajo del llamado al **mock**: `"describe ('Mocking.vue', () => { })"`.

```
1 describe('Mocking.vue', () => {
2
3 })
```

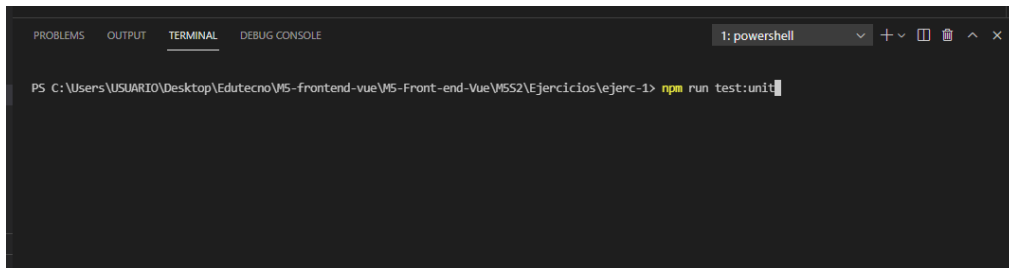
De inmediato, crearemos la prueba usando la variable **"it"**. Dentro de la función de `"describe"`, le asignamos el nombre a la prueba como un **string** `"fetches async when a button is clicked"`, y continuamos con la función: `"it('fetches async when a button is clicked', () => { })"`.

```
1 describe('Mocking.vue', () => {  
2  
3     it('fetches async when a button is clicked', () => {  
4     })  
5 })
```

Ahora, dentro de la función **“it”**, haremos el montaje del componente usando **“shallowMount”**, y se lo asignaremos a la constante **“wrapper”**. Por eso, escribimos: **“const wrapper = shallowMount(Mocking)”**.

```
1 describe('Mocking.vue', () => {  
2  
3     it('fetches async when a button is clicked', () => {  
4         const wrapper = shallowMount(Mocking)  
5     })
```

De inmediato corremos las pruebas, escribiendo en la terminal el comando: **“npm run test:unit”**, y presionamos “enter”.



```
1: powershell  
PS C:\Users\USUARIO\Desktop\Edutecno\M5-frontend-vue\M5-Front-end-Vue\M5S2\Ejercicios\ejerc-1> npm run test:unit
```

Como ya lo habíamos mencionado con anterioridad, **Jest** corre todas las pruebas existentes en paralelo.

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
1: powershell

> ejerc-1@0.1.0 test:unit C:\Users\USUARIO\Desktop\Edutechno\VS-front-end-Vue\VS2\Ejercicios\ejerc-1
> vue-cli-service test:unit

 PASS  tests/unit/greeting.spec.js
 PASS  tests/unit/example.spec.js
 PASS  tests/unit/mocking.spec.js
  ● Console

    console.error node_modules/vue/dist/vue.runtime.common.dev.js:621
      [Vue warn]: Failed to mount component: template or render function not defined.

      found in

      -> <Anonymous>
           <root>

Test Suites: 3 passed, 3 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        3.755s
Ran all test suites.
  
```

Podemos observar que todas las pruebas pasan, pero en la prueba **“Mocking”**, nos alerta que tuvo problemas para montar el componente. Esto es correcto, ya que aún no le hemos creado contenido. Como esta prueba pasa, a pesar de lo anterior, necesitamos mejorarla. Para eso, vamos a buscar los elementos que gatillan nuestra llamada a la **API**. Comenzaremos por el botón. Así escribimos, justo debajo de **const wrapper**, **“wrapper.find('button')”**.

```
1 wrapper.find('button')
```

El método **“find()”** pertenece a **Vue-test-utils**, y sirve para buscar elementos utilizando cualquier selector válido. Devuelve el contenido del primer nodo **DOM**, que coincida con el selector. Usa como parámetros el selector en formato **“string”**. Como ya logramos encontrar nuestro botón, nos falta agregar un método que sea capaz de captar el “clic” que gatilla la llamada a la **API**. Para eso, usaremos el método **trigger('click')**. Éste activa un evento asíncrono en el nodo **DOM** de **Wrapper**, y devuelve una “Promesa”, que cuando se resuelve, garantiza que el componente está actualizado, y solo funciona con eventos **DOM** nativos. Recibe como argumentos un **string** del evento. El código queda así:

```

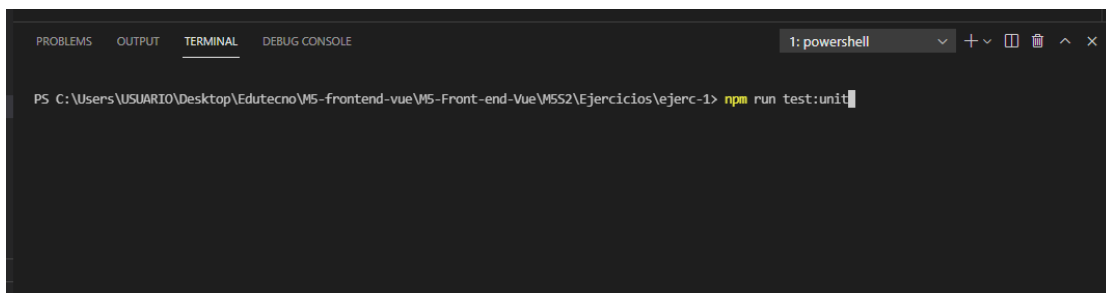
1 describe('Mocking.vue', () => {
2   it('fetches async when a button is clicked', () => {
3     const wrapper = shallowMount(Mocking)
4     wrapper.find('button').trigger('click')
5   })
6 })
  
```

```
6 })
```

Ahora que ya tenemos el evento que gatilla la llamada a la **API** capturado, crearemos una aserción con el comportamiento deseado del componente. Para eso, usaremos **“expect”**. Como vimos en el ejercicio pasado, la sintaxis para crear una aserción es: **“expect(received).to [matcher] (expected)”**, donde **“Received”** debe ser reemplazado con el **“value”** del componente, así que escribimos: **“expect(wrapper.vm.value)”**. **“Wrapper.vm”** se usa para emitir un evento personalizado. **Vm** es la instancia de **Vue**, y a través de ella, podemos acceder a los métodos y propiedades. En este caso, lo usamos para acceder al valor de **“data.value”**. Continuando con la sintaxis de la aserción, usaremos un método o función **“Matcher”** de **Jest**. Seleccionaremos uno que nos permita comparar el contenido de **“received”**, con el valor **“expected”**. Podemos usar **“toBe”**, que comparará valores primitivos, o verificará la identidad referencial de instancias de objetos, quedando así: **“expect(wrapper.vm.value).toBe('value')”**.

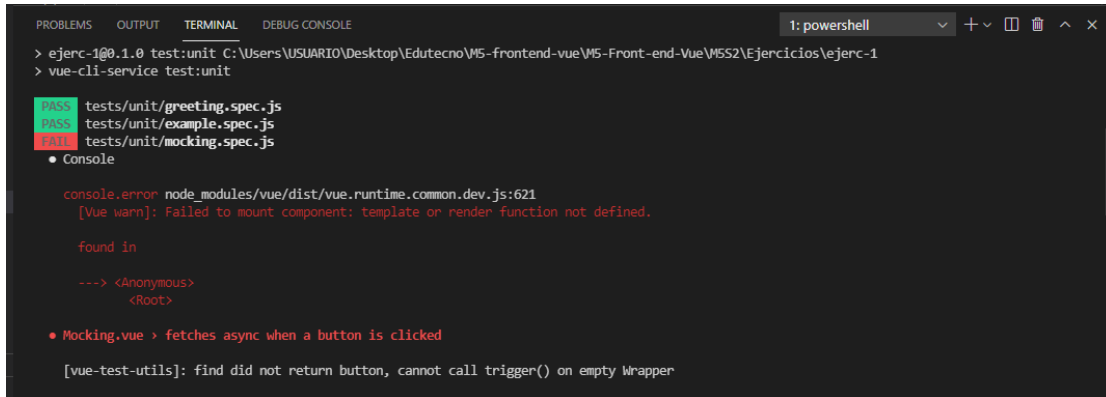
```
1 describe('Mocking.vue', () => {
2   it('fetches async when a button is clicked', () => {
3     const wrapper = shallowMount(Mocking)
4     wrapper.find('button').trigger('click')
5     expect(wrapper.vm.value).toBe('value')
6   })
7 })
8
```

Ahora, podemos correr la prueba usando: **“npm run test:unit”**, y presionamos **“enter”**.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
1: powershell
PS C:\Users\USUARIO\Desktop\Edutecno\MS-frontent-vue\MS-Front-end-Vue\MS2\Ejercicios\ejerc-1> npm run test:unit
```

Podemos observar que todas las pruebas pasan, excepto **“Mocking”**, donde nos alerta que tuvo problemas para encontrar el botón.



```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
1: powershell
> ejerc-1@0.1.0 test:unit C:\Users\USUARIO\Desktop\Edutechno\MS-front-end-vue\MS-Front-end-Vue\MS2\Ejercicios\ejerc-1
> vue-cli-service test:unit
PASS tests/unit/greeting.spec.js
PASS tests/unit/example.spec.js
FAIL tests/unit/mockings.spec.js
  ● Console
    console.error node_modules/vue/dist/vue.runtime.common.dev.js:621
      [Vue warn]: Failed to mount component: template or render function not defined.

      found in
        ---> <Anonymous>
              <Root>

    ● Mocking.vue > fetches async when a button is clicked
      [vue-test-utils]: find did not return button, cannot call trigger() on empty wrapper
  
```

Esto es correcto, ya que aún no le hemos creado contenido al componente llamado por la prueba. Vamos a generar el contenido mínimo del componente para revisar nuevamente. Así que vamos a **“Mocking.vue”**.

```

1 <template>
2   <button @click="fetchResults" />
3 </template>
  
```

Como se puede notar, inmediatamente escribimos el evento **“click”**, además de enlazarlo a la función que llamamos **“fecthResults”**, que es la que traerá los datos desde la **API**. Continuamos escribiendo el **script**, donde agregaremos:

```

1 import axios from 'axios'
2 export default {
3   data () {
4     return {
5       value: null
6     }
7   },
8   methods: {
  
```

```

9      async fetchResults () {
10          const response = await axios.get('mock/service')
11          this.value = response.data
12      }
13  }
14 }
15
16

```

Así importamos **Axios**. Asignamos en **"data"**, el **"value = null"**, y en los métodos creamos una función asíncrona para el llamado a la **API**, pero, en vez de llamar directamente, le pasamos el llamado al **mock** que hemos creado para sustituirla y, de inmediato, le indicamos que renderice **"data.value"** con el valor obtenido de la llamada.

Nuevamente corremos la prueba usando **"npm run test:unit"**, y presionamos "enter".



```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
PASS tests/unit/greeting.spec.js
PASS tests/unit/example.spec.js
FAIL tests/unit/mocking.spec.js
  ● Mocking.vue > fetches async when a button is clicked

    expect(received).toBe(expected) // Object.is equality

    Expected: "value"
    Received: null

    10 |         const wrapper = shallowMount(Mocking)
    11 |         wrapper.find('button').trigger('click')
    > 12 |         expect(wrapper.vm.value).toBe('value')
       |                                   ^
    13 |
    14 |     })
    15 | })

at Object.<anonymous> (tests/unit/mocking.spec.js:12:34)

```

Podemos ver que la prueba falló nuevamente, pero esta vez porque el valor obtenido en **"received"** fue **"null"** por defecto, que dejamos al crear el componente. Eso significa que no hemos gestionado la promesa correctamente, y no hemos dado tiempo a que **"wrapper.vm.value"** tenga el valor que le corresponde. **Vue** posee la función **\$nextTick**, que aplaza la ejecución de la devolución de la llamada, hasta que se vuelva a actualizar el **DOM**, tal como lo necesitamos en nuestro caso. Así que lo usaremos inmediatamente después del cambio de datos, para esperar la actualización del **DOM**, justo antes de correr la aserción. Cambiaremos nuestro código, para que quede así:



```
1 describe('Mocking.vue', () => {
2   it('fetches async when a button is clicked', () => {
3     const wrapper = shallowMount(Foo)
4     wrapper.find('button').trigger('click')
5     wrapper.vm.$nextTick(() => {
6       expect(wrapper.vm.value).toBe('value')
7     })
8   })
9 })
```

Si corremos esta prueba tal como está, veremos que pasa. Sin embargo, hay un detalle que no es visible, y que puede generar errores. Dado que el componente se comporta de manera asíncrona, pero el test es secuencial, y Jest no entiende de asincronismo tal y como está planteado, la prueba terminará de correrse antes de esperar a la ejecución de las aserciones (**expect**). La solución la entrega **Jest**, a través de la función callback llamada **“done()”**. Ésta nos permite indicar cuándo un test se tiene que dar por terminado, considerando un tiempo máximo. Si pasado ese tiempo, no se ha ejecutado **“done()”**, **Jest** sigue con el resto de las pruebas. Haciendo los cambios que corresponden, el código completo queda así:

```
1 describe('Mocking.vue', () => {
2   it('fetches async when a button is clicked', (done) => {
3     const wrapper = shallowMount(Mocking)
4     wrapper.find('button').trigger('click')
5     wrapper.vm.$nextTick(() => {
6       expect(wrapper.vm.value).toBe('value')
7       done()
8     })
9   })
10 })
```

Volvemos a escribir el comando **“npm run test:utils”**, y presionamos **“enter”**.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
1: powershell

PS C:\Users\USUARIO\Desktop\Edutecno\VS-front-end-vue\VS-Front-end-Vue\MSS2\Ejercicios\ejerc-1> npm run test:unit
```

Vemos que el test funciona. Así confirmamos que la prueba corre de manera correcta.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
1: powershell

> ejerc-1@0.1.0 test:unit C:\Users\USUARIO\Desktop\Edutecno\VS-front-end-vue\VS-Front-end-Vue\MSS2\Ejercicios\ejerc-1
> vue-cli-service test:unit

 PASS   tests/unit/example.spec.js
 PASS   tests/unit/greeting.spec.js
 PASS   tests/unit/mocking.spec.js

Test Suites: 3 passed, 3 total
Tests:      3 passed, 3 total
Snapshots:  0 total
Time:       3.426s
Ran all test suites.
PS C:\Users\USUARIO\Desktop\Edutecno\VS-front-end-vue\VS-Front-end-Vue\MSS2\Ejercicios\ejerc-1>
```

Con esto, finalmente hemos terminado de crear un test unitario asíncrono, usando un **mock** de **axios** para saltar la llamada a la **API**.