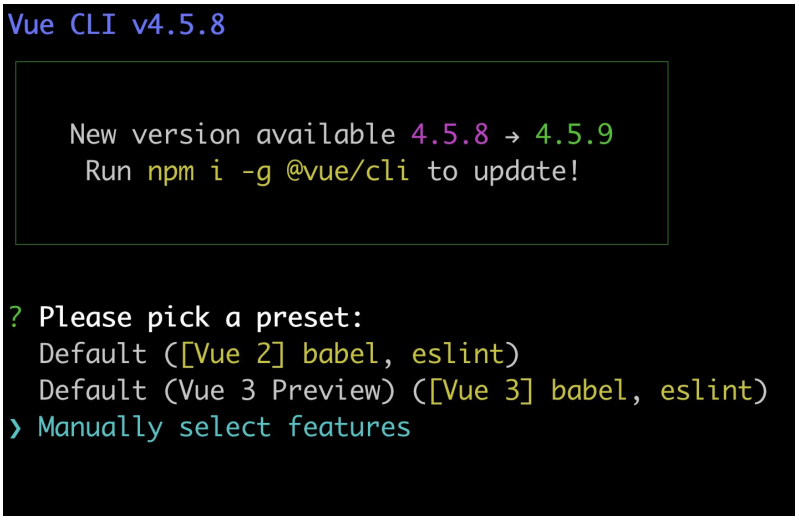
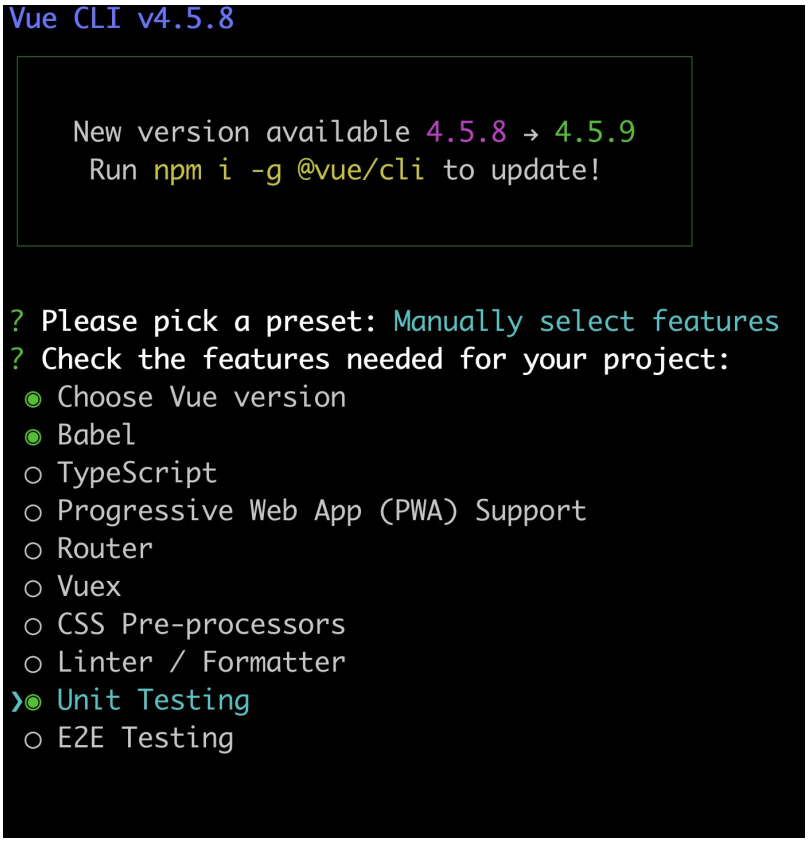
Framework de pruebas jest y Vue Testing

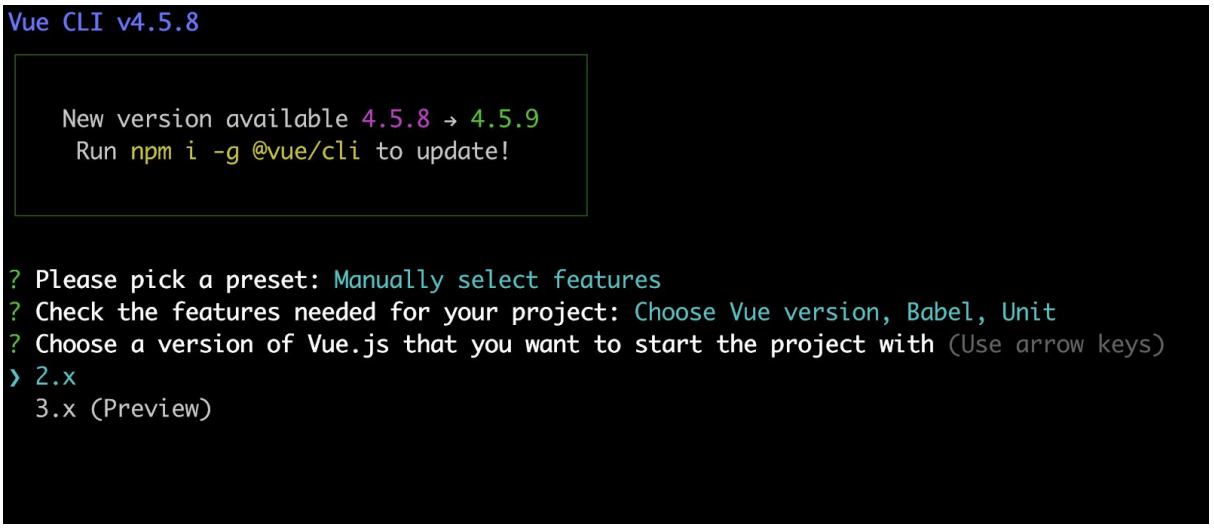
Jest - Jest es un marco de prueba de JavaScript desarrollado por Facebook. Funciona de inmediato con una configuración mínima y tiene un corredor de pruebas incorporado, una biblioteca de afirmaciones y soporte para simulaciones.

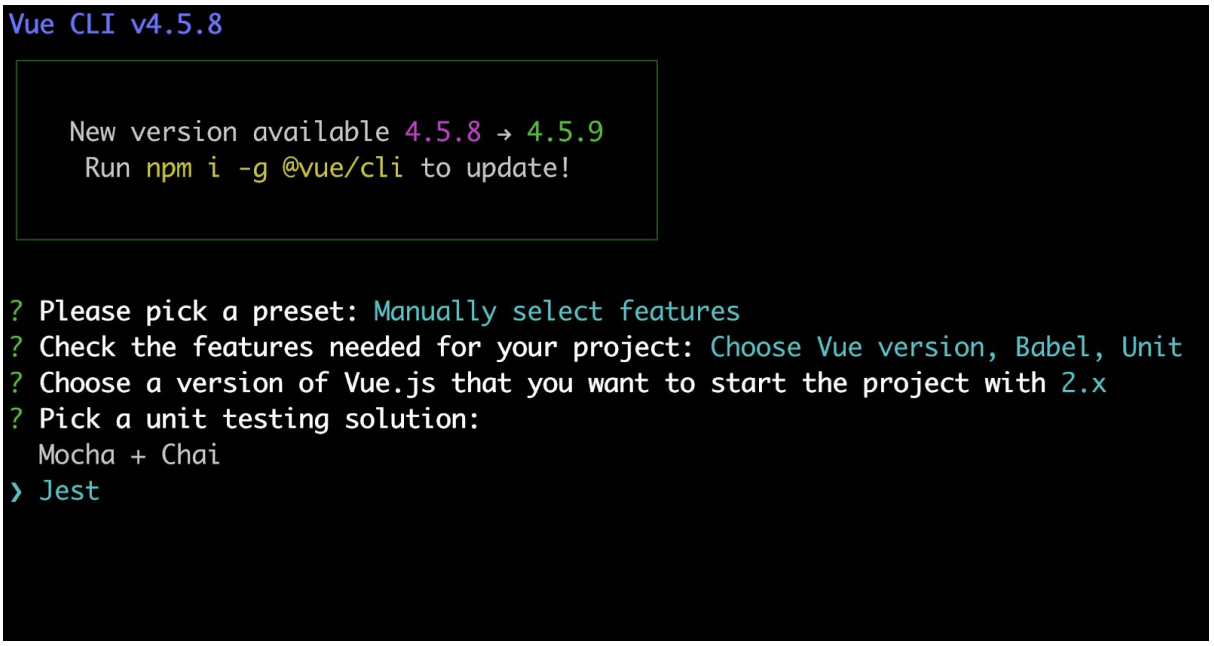
Vue Test Utils : Vue Test Utils (VTU) es un conjunto de funciones de utilidad destinadas a simplificar las pruebas de los componentes de Vue.js. Proporciona algunos métodos para montar e interactuar con componentes de Vue de manera aislada.Se puede instalar por defecto en un proyecto de Vue-CLI usando Jest test framework.

Pasos a seguir para tener jest y testing en un proyecto Vue




Vue CLI v4.5.8
New version available 4.5.8 → 4.5.9
Run npm i -g @vue/cli to update!
? Please pick a preset: Manually select features
? Check the features needed for your project:
◉ Choose Vue version
◉ Babel
○ TypeScript
○ Progressive Web App (PWA) Support
○ Router
○ Vuex
○ CSS Pre-processors
○ Linter / Formatter
❯◉ Unit Testing
○ E2E Testing





Para correr las pruebas se puede correr el comando
`npm run  test:unit`

Consejos de como hacer testeo en el front

En el caso de los componentes de la interfaz de usuario (osea en el front), no se recomienda probar todo el código, ya que conduce a un enfoque excesivo en los detalles de implementación interna de los componentes y podría resultar en pruebas frágiles.

En su lugar, recomendamos escribir pruebas que afirmen la interfaz de su componente y traten sus partes internas como una caja negra. Un solo caso de prueba afirmaría que alguna entrada (interacción del usuario) proporcionada al componente da como resultado la salida esperada (resultado de procesamiento o eventos personalizados emitidos).

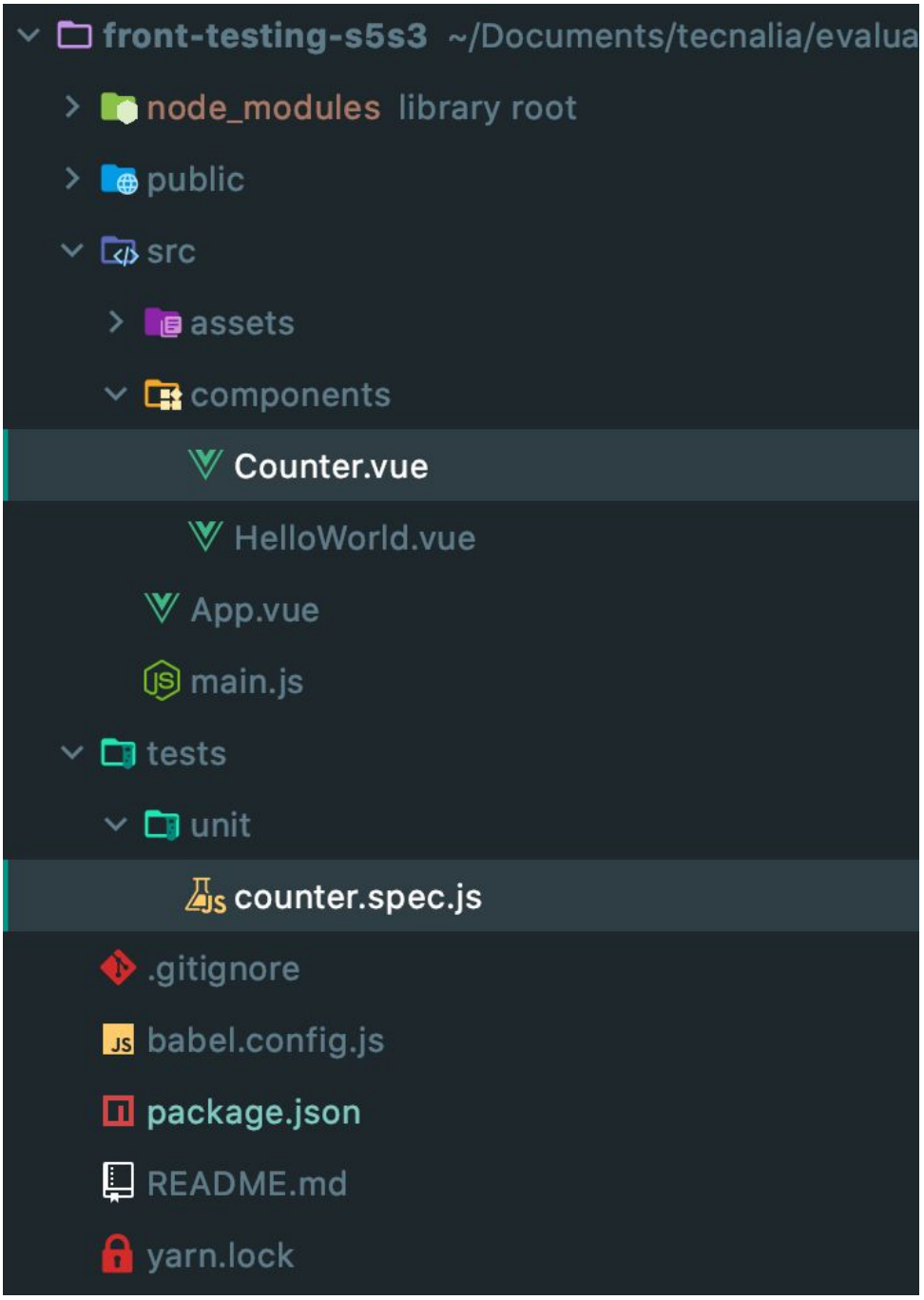
Por ejemplo, imagine un componente Counter que incrementa un contador de pantalla en 1 cada vez que se hace clic en un botón. Su caso de prueba es al hacer clic se debe afirmar

que la salida renderizada ha aumentado en 1. La prueba no debería preocuparse por cómo Counter incrementa el valor, solo le importa la entrada y la salida.

El beneficio de este enfoque es que siempre que la interfaz de su componente siga siendo la misma, sus pruebas pasarán sin importar cómo cambié la implementación interna del componente con el tiempo.

Implementación ejemplo counter.

Vamos a crear un componente Counter en la carpeta components y a su vez su prueba unitaria que va en la carpeta tests y se va llamar counter.spec.js.



Al querer hacer pruebas lo que vamos a hacer es describir el funcionamiento del componente y vamos a importar el componente creado que aún no tiene la implementación.

Empezamos con lo más sencillo, cuando el counter empieza empieza en 0.

```
import Counter from "../../src/components/Counter";

describe('Counter', () => {
    it('defaults to a count 0', () => {

    })
})
```

Podemos observar dos funciones, una que *"describe"* que usa para describir una parte de la funcionalidad, puede ser un componente o una función específica de la app. Vamos a describir el comportamiento del Counter.

Después podemos ver un *"it"* eso va explicar cada caso de la descripción del componente, como por ejemplo que debe empezar en 0, otro podría ser al hacer click sobre el botón más se debe incrementar el contador y mostrarlo.

Como sabemos el ciclo de vida de componente vue pasa por algunos "eventos" una de ellas que se usa mucho es el mounted que es cuando se ha montado el componente. Al querer hacer pruebas unitarias de un componente debemos montar el componente para poder empezar a interactuar con el.

Esto se puede hacer importando el mount de vue test utils.

En este ejemplo vamos a montar el componente y acceder a su data y ver si empieza en 0.

```
import Counter from "../../src/components/Counter";
import { mount } from "@vue/test-utils";

describe('Counter', () => {
    it('defaults to a count 0', () => {
        const wrapper = mount(Counter)
        expect(wrapper.vm.count).toBe(0)
    })
})
```

Como se puede ver importamos una función mount para poder montar el componente y se puede acceder al componente, también podemos ver un expect que nos ayuda para comparar valores.

Con esto ya el primero caso de las pruebas quedó listo ahora lo que vamos a hacer es ejecutar las pruebas que sabemos que va fallar.

npm run test:unit

```
FAIL tests/unit/counter.spec.js
  Counter
    × defaults to a count 0 (12ms)

  ● Counter › defaults to a count 0

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

    Expected: 0
    Received: undefined

       5 |     it('defaults to a count 0', () => {
       6 |         const wrapper = mount(Counter)
    >  7 |         expect(wrapper.vm.count).toBe(0)
         |                                  ^
       8 |     })
       9 | })
      10 |
```

Nos falla por dice que el componente no tiene en el data el count. Vamos a empezar a implementar el componente a medida que vamos haciendo los tests.

```
<template>
   <div>

   </div>
</template>

<script>
   export default {
       name: "Counter",
       data: () => ({
           count: 0
       }),
   }
</script>
```

Si ahora volvemos a ejecutar las pruebas pasa la prueba.

La nueva prueba va ser que el componente debe de mostrar el contador actual. En vue test utils podemos acceder al rendered html.

La prueba quedaría de esta manera, como sabemos que el componente empieza en 0 podemos decir que cuando se monta el componente debe de mostrar por lo menos un 0.

```
describe('Counter', () => {
  it('defaults to a count 0', () => {
    const wrapper = mount(Counter)
    expect(wrapper.vm.count).toBe(0)
  })

  it('shows the current count', () => {
    const wrapper = mount(Counter)
    expect(wrapper.html()).toContain(0)
  })
})
```

Al volver a correr las pruebas va volver a fallar el componente porque no estamos mostrando el valor del contador en el template.

Al agregar para que el componente queda de la siguiente manera vuelve a funcionar

```
<template>
   <div>
       <span>
           {{count}}
       </span>
   </div>
</template>

<script>
   export default {
       name: "Counter",
       data: () => ({
           count: 0
       }),
   }
</script>
```

Estamos mostrando el count en el html entonces cuando se renderiza el html quedaría como

```
<div>
  <span>
    0
  </span>
</div>
```

Si volvemos a correr otra vez las pruebas ahora vuelve y pasar.

El próximo paso es agregar cuando se hace click sobre el botón más se incrementa el contador. En este paso vamos a ver cómo podemos hacerlo.

Sabemos que necesitamos un botón con un id de ejemplo como id=increment, y al hacer click debe de mostrar 1.

```
it('increments the count on increment button click', () => {
   const wrapper = mount(Counter)
   wrapper.find('#increment').trigger('click')
   expect(wrapper.vm.count).toBe(1)
})
```

Con el find le decimos que encuentre el botón con id increment y que emite un evento click sobre el botón

```
wrapper.find('#increment').trigger('click')
```

De el count ahora debe ser 1.

Otra vez al correr las pruebas falla, y es porque no tenemos un botón y no hacemos la lógica de incrementar el contador.

Al implementar el increment quedaría de la siguiente forma.

```
<template>
   <div>
       <span>
           {{count}}
       </span>
       <button id="increment" @click="count++">
           +
       </button>
   </div>
</template>
```

```
<script>
   export default {
       name: "Counter",
       data: () => ({
           count: 0
       }),
   }
</script>
```

Ahora se puede hacer otros casos como el de restar y poner restricciones como no ser números negativos. Esto se puede lograr creando funciones y hacer la lógica.

Se puede encontrar el código fuente en:

https://github.com/Tecnalia-Cilco-3/semana-5-session-3

Referencias:
https://es.vuejs.org/v2/guide/unit-testing.html
https://vue-test-utils.vuejs.org/