Guía de ejercicios - Pruebas end-to-end en un entorno Vue

¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo practicar y ejercitar los contenidos que hemos visto en clase.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

Actividad guiada: Repaso de pruebas unitarias

A continuación, realizaremos un ejercicio donde realizaremos pruebas unitarias con Jest con Vue JS. Este ejercicio servirá para repasar conceptos asociados al desarrollo de pruebas en una aplicación Vue JS. La aplicación final será un visualizador de tareas.

Nota: En esta ocasión **no** implementaremos pruebas E2E dado que estas suelen ser costosas en tiempo para su implementación.

- **Paso 1:** Crea la aplicación usando el comando `npm create vue@latest` con el nombre `repaso-vue-test`, siguiendo esta configuración:

```
Vue.js - The Progressive JavaScript Framework

✔ Project name: … repaso-vue-test
✔ Add TypeScript? … No / Yes
✔ Add JSX Support? … No / Yes
✔ Add Vue Router for Single Page Application development? … No / Yes
✔ Add Pinia for state management? … No / Yes
✔ Add Vitest for Unit Testing? … No / Yes
✔ Add an End-to-End Testing Solution? › No
✔ Add ESLint for code quality? … No / Yes
✔ Add Prettier for code formatting? … No / Yes
✔ Add Vue DevTools 7 extension for debugging? (experimental) … No / Yes
```

- **Paso 2:** Ahora instalaremos Jest y las dependencias necesarias para realizar las pruebas:

```
npm install --save-dev jest jest-environment-jsdom babel-jest @babel/preset-env
@vue/vue3-jest @vue/test-utils
```

- **Paso 3:** A continuación configuraremos Jest, Babel y ESLint.

Jest: jest.config.cjs

```
module.exports = {
 testEnvironment: 'jsdom',
 transform: {
  '^.+\\.vue$': '@vue/vue3-jest',
  '^.+\\js$': 'babel-jest'
 },
 testRegex: '(/__tests__/.*|(\\.|/)(test|spec))\\.(js|ts)$',
 moduleFileExtensions: ['vue', 'js'],
```

```
  moduleNameMapper: {
    '^@/(.*)$': '<rootDir>/src/$1'
  },
  coveragePathIgnorePatterns: ['/node_modules/', '/tests/'],
  coverageReporters: ['text', 'json-summary'],
  testEnvironmentOptions: {
    customExportConditions: ['node', 'node-addons']
  }
}
```

Babel: babel.config.cjs

```
module.exports = {
  env: {
    test: {
      presets: [
        [
          '@babel/preset-env',
          {
            targets: {
              node: 'current'
            }
          }
        ]
      ]
    }
  }
}
```

ESLint: .eslintrc.cjs : Agregamos Jest como plugin

```
module.exports = {
  ...,
  plugins: ['jest']
}
```

- **Paso 4:** Ahora debemos agregar un script en nuestro package.json que nos permita correr las pruebas. Lo nombraremos test:unit

```
{
 . . .,
  "scripts": {
    . . . ,
    "test:unit": "jest"
  },
 . . .
}
```

- **Paso 5:** Creamos un componente que llamaremos `TodoApp.vue` con el siguiente código y lo importamos en App.js.

```
<template >
    <div>
        <h1>Lista de tareas</h1>

        <div>
            <li v-for="todo in todoList" :key="todo.id" data-test="tarea">
              {{ todo.description }}
            </li>
        </div>
    </div>
</template>
<script>
export default {
    name: 'TodoApp',
    data() {
        return {
           todoList: [
            {id: 1, description: 'Description toDo 1', completed: false}
           ]
        }
    }
}
</script>
```

Entendamos el código

- Tenemos un componente llamado `TodoApp.vue` que muestra tareas.
- Estas tareas están guardadas en la propiedad data, es decir, es reactiva.
- Creamos un `<li>` por cada elemento disponible en el arreglo de tareas.
- Los li tienen un atributo llamado `data-test="tarea"`

Este `data-test` cumple la función de identificador de elemento, es parecido a definir el atributo id. El `data-test` lo agregamos como buena práctica dado que deseamos generar una prueba que identifique si existe un texto en específico en el elemento seleccionado.

- **Paso 6:** Escribimos el código de nuestra prueba en @/tests/unit/components/TodoApp.spec.js.

```
import { shallowMount } from "@vue/test-utils";
import TodoApp from '@/components/TodoApp'

describe('TodoApp.vue', () => {
  test('Se muestra la descripción de la tarea', () => {
    const wrapper = shallowMount(TodoApp)

    const todo = wrapper.get('[data-test="tarea"]')
    expect(todo.text()).toBe("Description toDo 1")
  })
})
```

1. Esta prueba está utilizando `shallowMount` para el montaje del componente que deseamos testear.
2. Luego creamos el contenedor con el wrapper del componente `TodoApp.vue`.
3. Seguidamente, generamos una variable llamada `todo` que tomará el componente montado en el wrapper y buscará u obtendrá un elemento del HTML que contenga el atributo `data-test="tarea"`.
4. Por último, esperamos que el texto sea estrictamente `"Description toDo 1"`.

Ejecutamos la prueba con el comando `npm run test:unit`

Hasta este punto del ejercicio, hemos realizado una prueba que valida la existencia de un texto en específico dentro del componente. Dado que Jest provee un conjunto de opciones para probar de manera unitaria elementos o funciones de nuestra aplicación, entonces vamos a validar funcionalidades que se ejecutarán a partir de la interacción del usuario.

- **Paso 7:** Supongamos que vamos a probar la funcionalidad de un formulario que al ser enviado agregará una nueva tarea al listado. Para ello, podemos escribir primero el test que al correrlo fallará y luego agregar el HTML al componente para que el test pase.

```
test('Deberá agregar una nueva tarea', () => {
    const wrapper = shallowMount(TodoApp)

    expect(wrapper.findAll('[data-test="tarea"]')).toHaveLength(1)

    wrapper.get('[data-test="nueva-tarea"]').setValue('Nueva tarea')
    wrapper.get('[data-test="form"]').trigger('submit')

    expect(wrapper.findAll('[data-test="tarea"]')).toHaveLength(2)
})
```

1. Primero declaramos una aserción donde buscamos todos los elementos con el atributo `data-test="tarea"` y esperamos que la búsqueda arroje un valor igual a 1.
2. Luego, modificamos el valor de `data-test="nueva-tarea"` y le decimos que su nuevo valor será 'Nueva tarea'
3. Seguidamente, buscamos un elemento que contenga el atributo `data-test="form"` el cual esperamos que sea un formulario y que el `setValue` se ejecuta al disparar el evento `submit`. Recordemos que el evento submit es propio de los formularios.
4. Después de lo antes mencionado, esperamos que el nuevo valor `data-test="tarea"` ahora contenga dos elementos.

- **Paso 8:** Ahora, creamos el código necesario en nuestro componente `TodoApp.vue` para el formulario y el evento submit de dicho formulario.

```
<template >
    <div>
        <h1>Lista de tareas</h1>

        <div>
            <li v-for="todo in todoList" :key="todo.id" data-test="tarea">
               {{ todo.description }}
            </li>
        </div>
        <form data-test="form" @submit.prevent="crearTarea">
            <input data-test="nueva-tarea" v-model="nuevaTarea">
        </form>
    </div>
</template>
<script>
export default {
    name: 'TodoApp',
    data() {
        return {
            nuevaTarea: '',
            todoList: [
                {id: 1, description: 'Description toDo 1', completed: false}
            ]
        }
    },
    methods: {
        crearTarea() {
            this.todoList.push({
                id: 2,
                description: this.nuevaTarea,
                completed: false
            })
        }
```

```
    }
}
</script>
```

Hasta este punto, si ejecutamos el test veremos el siguiente error:

Imagen 1. Error al correr el test
Fuente: Desafío Latam

Este error se genera dado que nuestra prueba está realizando acciones síncronas, pero en realidad Vue está esperando que el proceso sea asíncrono. Recordemos que cuando realizamos una modificación del DOM este proceso demora algunos segundos, ante esa situación debemos definir que nuestro test sea asíncrono.

- **Paso 9:** Modificamos el código de la prueba a asíncrono.

```
test('Deberá agregar una nueva tarea', async () => {
    const wrapper = shallowMount(TodoApp)

    expect(wrapper.findAll('[data-test="tarea"]')).toHaveLength(1)

    await wrapper.get('[data-test="nueva-tarea"]').setValue('Nueva tarea')
    await wrapper.get('[data-test="form"]').trigger('submit')

    expect(wrapper.findAll('[data-test="tarea"]')).toHaveLength(2)
  })
```

Ahora, simularemos el comportamiento de marcaje de una tarea como completada. Para lograrlo, escribiremos un test que simule cuando una tarea cambie su estado de false a true.

- **Paso 10:** Escribimos el nuevo test que evalúa el cambio de estado de la tarea con la presencia de una clase definida como completed.

```
test('Se deberá marcar como tarea completada', async () => {
    const wrapper = shallowMount(TodoApp)
    await wrapper.get('[data-test="checkbox"]').setValue(true)

expect(wrapper.get('[data-test="tarea"]').classes()).toContain('completed')
  })
```

- **Paso 11:** Escribimos el código del componente `TodoApp.vue` que ejecutará esta prueba.

```
<div>
      <li
        v-for="todo in todoList"
        :key="todo.id"
        data-test="tarea"
        :class="[todo.completed ? 'completed' : '']"
      >
        {{ todo.description }}
        <input type="checkbox" v-model="todo.completed"
data-test="checkbox">
      </li>
</div>
```

Ahora definimos el código de los estilos para modificar la tarea cuando se seleccione el elemento checkbox.

```
<style>
    .completed {
        text-decoration: line-through;
    }
</style>
```

Al ejecutar la aplicación en el servidor local veremos el siguiente resultado:

Imagen 2. Resultado de ejecución
Fuente Desafío Latam

En la plataforma tendrás el código finalizado de este ejercicio con el nombre **"Código Actividad - Repaso de pruebas unitarias"**.

¡Manos a la obra! - Validación de props

Utilizando la aplicación de ejemplo del ejercicio anterior, escribe un nuevo test que valide que el texto del h1 sea enviado mediante props.

Tips: Puedes utilizar de ejemplo el archivo HelloWorld.spec.js