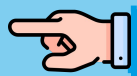
Pruebas Unitarias y end-to-end en un entorno Vue

Herramientas para el testing unitario

{desafío} latam_

Implementar pruebas unitarias utilizando las herramientas provistas por Vue para verificar el correcto funcionamiento del aplicativo.

- Unidad 1:
  Vue Router
- Unidad 2:
  Vuex
- Unidad 3:
  Firebase
- Unidad 4:
  Pruebas Unitarias y end-to-end en un entorno Vue



Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Reconoce los conceptos y herramientas utilizadas para la realización de pruebas unitarias en un entorno Vue.*

¿Por qué son importantes las pruebas en una aplicación web?

Según lo visto en la sesión anterior, ¿En qué consiste el desarrollo basado en pruebas?

{desafío} latam_

Contexto antes de iniciar

Conociendo ya la importancia de las pruebas unitarias, es importante conocer las herramientas que dispone Vue JS para su ejecución. Entre estas podemos encontrar Jest, Mocha y Chai.
En esta sesión, estaremos revisando algunas particularidades de estas herramientas y cómo estas nos ayudan en determinadas situaciones.

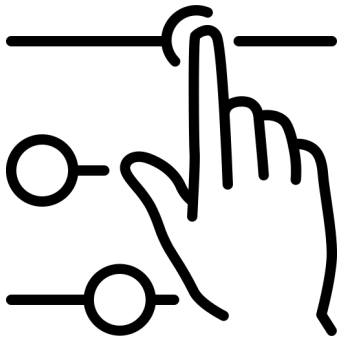
/* Jest y Mocha */

Consideraciones acerca de Jest

Es un marco de pruebas de JavaScript que corre bajo el entorno de Node JS (JavaScript del lado del servidor) y fue desarrollado por Facebook.
Este marco se basa en que el desarrollador ejecute la menor configuración posible para ejecutar pruebas. Esto hace que los tiempos de desarrollo sean óptimos y el esfuerzo esté enfocado en la escritura de los test.

Consideraciones acerca Mocha

Es un marco de pruebas de JavaScript que corre bajo el entorno de Node JS (JavaScript del lado del servidor).
A diferencia de Jest que contiene un marco establecido y recomendado de trabajo, con Mocha se pueden establecer configuraciones personalizables para su funcionamiento durante la construcción de los test.



/* Ventajas y desventajas */

¿Qué marco de pruebas elegir?

En el ámbito del desarrollo existen marcos de trabajo que pueden ayudar a realizar tareas específicas. En el caso de la ejecución de pruebas en una aplicación web, es recomendable revisar la documentación oficial de las tecnologías para sustentar una buena toma de decisiones.

- Documentación oficial de Jest
- Documentación oficial de Mocha

¿Qué marco de pruebas elegir?

En el caso particular de Jest y Mocha, existe un elemento fundamental al momento de elegir, una de las bondades de Jest es la fácil integración y el robusto sistema de funcionalidades que contiene para ejecutar pruebas.

En este sentido, podríamos elegir Jest si lo que deseamos es optimizar tiempo de escritura de pruebas más que la configuración del marco de pruebas en sí.

Demostración: "Escribiendo la primera prueba"

Escribiendo la primera prueba

A continuación, realizaremos un ejercicio en el cual escribiremos nuestras primeras pruebas en Vue JS haciendo uso de Jest. Para este ejercicio utilizaremos el **Material de apoyo - Escribiendo la primera prueba** que estará disponible en la plataforma.

Este apoyo consta de una aplicación simple en VueJS que ya contiene las dependencias necesarias, solo deberás descargar y ejecutar el comando npm install para la instalación de las dependencias.

Sigue los pasos…

Sigue los pasos…

- **Paso 1:** Accedemos al material de apoyo entregado desde Visual Studio Code.
- **Paso 2:** Una vez abierto en el editor, nos dirigimos al archivo `example.spec.js` y procedemos a escribir la estructura básica en una prueba.

```
describe("Nombre del componente", () => {
  test("Descripción de la prueba a realizar", ()=> {
    //Lógica para la implementación de la prueba
  })
})
```

Sigue los pasos…

- **Paso 3:** Ahora, simulemos una prueba en la cual queremos evaluar si un número es mayor a 30.

```
describe('MiComponente.vue', () => {
  test('Evaluamos si la variable numero es mayor a 30', () => {
    let numero = 25
    if(numero > 30) {
    }else {
      throw `${numero} no es mayor a 30`
    }
  })
})
```

Resultado de ejecución

```
> ejercicio-primera-prueba@0.0.0 test:unit
> jest

 FAIL  src/tests/unit/example.spec.js
  MiComponente.vue
    × Evaluamos si la variable numero es mayor a 30

  ● MiComponente.vue › Evaluamos si la variable numero es mayor a 30

    thrown: "25 no es mayor a 30"

      1 | describe('MiComponente.vue', () => {
    > 2 |   test('Evaluamos si la variable numero es mayor a 30', () => {
        |   ^
      3 |     let numero = 25
      4 |     if (numero > 30) {
      5 |     } else {

      at test (src/tests/unit/example.spec.js:2:3)
      at Object.describe (src/tests/unit/example.spec.js:1:1)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.593 s, estimated 1 s
Ran all test suites.
```

/* Aserciones a través de funcionalidades de Jest */

Funcionalidades de Jest

Aserciones

En programación, el concepto de aserciones podría referirse a las acciones que se ejecutan durante la etapa de desarrollo de una aplicación, para validar suposiciones acerca del comportamiento del código.

Patrón AAA:

- **Arrange**: Definimos el objeto y sus valores, que posteriormente serán testeados.
- **Act**: Ejecuta una acción que va a generar el resultado al momento de probar.
- **Assert**: Comprobamos que las pruebas cumplan cierta condición o criterio.

Funcionalidades de Jest

Estructura del patrón AAA en código

```
describe('MiComponente.vue', () => {
  test('Evaluamos si la variable numero es mayor a 30', () => {
    //Arrange
    let numero = 25

    //Act
    numero = numero + 6

    //Assert
    if(numero > 30) {

    }else {
      throw `${numero} no es mayor a 30`
    }
  })
})
```

Asserts

- Como mencionamos anteriormente, Jest posee una documentación oficial.
- En ella podemos encontrar una funcionalidad que permita realizar aserciones, esta se define como Expect.

- Según la documentación oficial, Expect nos provee un conjunto de métodos validadores que permitirán que el Assert de nuestro código de pruebas sea más sencillo.

Veamos esto en un ejemplo continuando con el ejercicio anterior.

Sigue los pasos…

- **Paso 4:** Utilicemos el “matcher” `toBeGreaterThan()` en la etapa de aserciones.

```
describe('MiComponente.vue', () => {
  test('Evaluamos si la variable numero es mayor a 30', () => {
    //Arrange
    let numero = 25
    //Act
    numero = numero + 5
    //Assert
    expect(numero).toBeGreaterThan(30)
  })
})
```

Resultado de ejecución

```
FAIL src/tests/unit/example.spec.js
  MiComponente.vue
    × Evaluamos si la variable numero es mayor a 30 (1 ms)

  ● MiComponente.vue › Evaluamos si la variable numero es mayor a 30

    expect(received).toBeGreaterThan(expected)

    Expected: > 30
    Received:   30

       6 |     numero = numero + 5
       7 |     //Assert
    >  8 |     expect(numero).toBeGreaterThan(30)
         |                    ^
       9 |   })
      10 | })
      11 |

      at Object.toBeGreaterThan (src/tests/unit/example.spec.js:8:20)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.584 s, estimated 1 s
Ran all test suites.
```

Sigue los pasos…

- **Paso 5:** Otro matcher interesante es el .toBeGreaterThanOrEqual().

```
describe('MiComponente.vue', () => {
  test('Evaluamos si la variable numero es mayor a 30', () => {
    //Arrange
    let numero = 25
    //Act
    numero = numero + 5
    //Assert
    expect(numero).toBeGreaterThanOrEqual(30)
  })
})
```

Sigue los pasos…

- **Paso 6:** Implementemos un matcher para cadenas de texto.

```
describe("MiComponente.Vue", () => {
  test("Evaluamos si una clave es mayor a 6 dígitos", ()=> {
    //Arrange
    let clave = '12345'
    //Act

    //Asserts
    expect(clave).toHaveLength(6)
  })
})
```

Ejercicio:
"Escribe tu prueba"

{desafío}
latam_

Escribe tu prueba

Contexto

A continuación, deberán realizar un ejercicio individual que consiste en escribir sus propias pruebas a partir de un requerimiento entregado:

Requerimientos:

- Escribe una nueva prueba donde se espera que el resultado de una división es exacta. Puedes utilizar el matcher toEqual().
- Escribe una prueba donde se valide que el texto de una variable es igual a “Estoy aprendiendo test con Jest”. Puedes utilizar el matcher toBe().

Nota: Trabaja con la misma aplicación del ejercicio anterior.

Resumen de la sesión

- Las diferencias puntuales entre los marcos de pruebas unitarias como Mocha y Jest consiste básicamente en la facilidad de integración y de implementación, en este sentido, Jest es mucho más óptimo.
- Jest provee un conjunto de métodos conocidos como matchers, estos nos permiten realizar aserciones en nuestras pruebas y agilizar los resultados que se esperan.
- Los matchers de Jest se pueden encontrar en su documentación oficial.

¿Para qué nos sirven los matchers en Jest?

Próxima sesión...

- *Implementa pruebas unitarias utilizando las herramientas provistas por Vue para verificar el funcionamiento de un componente.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales