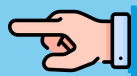
Pruebas Unitarias y end-to-end en un entorno Vue

Pruebas unitarias

{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.*

¿Consideran ustedes importante que se valide la calidad de las aplicaciones web?

{desafío} latam_

¿Cómo consideran que se puede validar que una aplicación se ejecuta correctamente?

Contexto antes de iniciar

En esta sesión, estaremos abordando el desarrollo de aplicaciones web basado en pruebas. Test Driven Development (TDD) es un proceso metodológico que nos servirá para desarrollar aplicaciones que cumplan con ciertos criterios y requisitos para que su funcionamiento sea correcto.

/* Pruebas unitarias */

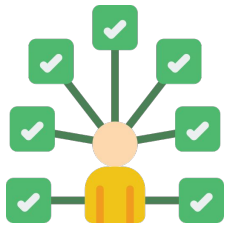
¿Qué son las pruebas unitarias?

- Imaginemos que tenemos una función que su única tarea es recibir una url como parámetro y en el cuerpo de ella se encuentra toda la lógica de conexión a una API REST.
- Como desarrolladores definimos que esa url será recibida como un string.
- Según lo anterior, podríamos entonces escribir una prueba que garantice:
  a. La presencia de la url
  b. Que el formato de la url efectivamente sea un string.

Características de las pruebas unitarias

Algunas de sus características pueden ser:

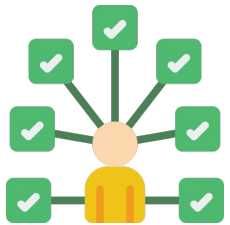
- **Fácil escritura:** Se garantiza gracias a que cada pieza del código puede ser probada.
- **Lectura simple:** La descripción de la funcionalidad de la prueba debe ser simple y concisa.
- **Unitarias:** Principalmente unitarias, esto nos permitirá dividir en pequeños bloques de código nuestras aplicaciones y sobre esos bloques hacer las pruebas.



Otras características...

Algunas de sus características pueden ser:

- **Calidad de código:** Se garantiza gracias a que cada pieza del código puede ser probada.
- **Confianza:** Al realizar pruebas, garantizamos el correcto funcionamiento futuro del código.
- **Optimización de errores:** Permiten encontrar errores de manera temprana.



Demostración: "Integrando el setup de pruebas unitarias con Vue"

Integrando el setup de pruebas unitarias con Vue

Contexto

A continuación, realizaremos un ejercicio para integrar el setup de pruebas unitarias en una aplicación Vue JS. Esto lo haremos utilizando create-vue y haremos una revisión de los archivos y dependencias que se generan al realizar esta configuración.

Sigue los pasos…

Sigue los pasos…

- **Paso 1:** Creamos una aplicación con `npm create vue@latest` con el nombre pruebas-vue, utilizaremos la siguiente configuración.

```
> npx
> create-vue


Vue.js - The Progressive JavaScript Framework

✔ Project name: … pruebas-vue
✔ 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
```

Sigue los pasos…

- **Paso 2:** Una vez dentro de nuestro proyecto y luego de instalar sus dependencias. Instalaremos **Jest**, la herramienta que utilizaremos para las pruebas unitarias, junto a las dependencias necesarias para las pruebas.

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

Sigue los pasos…

- **Paso 3:** Revisemos las dependencias que se integran al archivo package.json.

```
{
  "name": "pruebas-vue",
  "version": "0.0.0",
  "private": true,
  "type": "module",
    ▷ Debug
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "lint": "eslint . --ext .vue,.js,.jsx,.cjs,.mjs --fix --ignore-path .gitignore",
    "format": "prettier --write src/"
  },
  "dependencies": {
    "vue": "^3.4.29"
  },
  "devDependencies": {
    "@babel/preset-env": "^7.25.4",
    "@rushstack/eslint-patch": "^1.8.0",
    "@vitejs/plugin-vue": "^5.0.5",
    "@vue/eslint-config-prettier": "^9.0.0",
    "@vue/test-utils": "^2.4.6",
    "@vue/vue3-jest": "^29.2.6",
    "babel-jest": "^29.7.0",
    "eslint": "^8.57.0",
    "eslint-plugin-vue": "^9.23.0",
    "jest": "^29.7.0",
    "jest-environment-jsdom": "^29.7.0",
    "prettier": "^3.2.5",
    "vite": "^5.3.1"
  }
}
```

Sigue los pasos…

- **Paso 4.1:** Configuremos Jest, crearemos un archivo llamado jest.config.cjs en la raíz de nuestro proyecto y agregaremos la siguiente configuración:

```
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']
 }
}
```

Sigue los pasos…

- **Paso 4.2:** Jest necesita Babel para funcionar correctamente, para esto crearemos un archivo llamado babel.config.cjs en la raíz de nuestro proyecto y agregaremos la siguiente configuración:

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

Sigue los pasos…

- **Paso 4.3:** Ahora agregaremos Jest como plugin reconocible por ESLint, para eso, agregaremos: `plugins: ['jest']` en la raíz de nuestro archivo eslintrc.cjs quedando así:

```
/* eslint-env node */
require('@rushstack/eslint-patch/modern-module-resolution')

module.exports = {
 root: true,
 extends: [
   'plugin:vue/vue3-essential',
   'eslint:recommended',
   '@vue/eslint-config-prettier/skip-formatting'
 ],
 parserOptions: {
   ecmaVersion: 'latest'
 },
 plugins: ['jest']
}
```

Sigue los pasos…

- **Paso 5:** Ahora generemos nuestra primera prueba.

  Para eso crearemos el archivo `/src/test/unit/HelloWorld.spec.js`

  Y probaremos que el componente HelloWorld renderice correctamente la prop msg:

```
import { shallowMount } from '@vue/test-utils'
import HelloWorld from '@/components/HelloWorld.vue'

describe('HelloWorld.vue', () => {
 it('renders props.msg when passed', () => {
   const msg = 'new message'
   const wrapper = shallowMount(HelloWorld, {
     props: { msg }
   })

   expect(wrapper.text()).toMatch(msg)
 })
})
```

Sigue los pasos…

- **Paso 6:** Ahora, en nuestro `package.json` agregaremos un nuevo script para correr los tests.

  Lo nombraremos `test:unit`

```
"scripts": {
    …
    …
    "test:unit": "jest"
 }
```

Sigue los pasos…

- **Paso 6:** Corramos el archivo de pruebas HelloWorld.spec.js.

Esto lo realizaremos con el comando npm run test:unit. Luego, veamos el resultado de esta ejecución.

```
> pruebas-vue@0.0.0 test:unit
> jest

 PASS  src/test/unit/HelloWorld.spec.js
  HelloWorld.vue
    ✓ renders props.msg when passed (12 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.709 s, estimated 1 s
Ran all test suites.
```

¿Cómo se genera el reporte de pruebas?

Al correr el archivo de pruebas recibimos por terminal un reporte de aquellas que se ejecutaron satisfactoriamente.

- Una prueba que pasa como satisfactoria se define en el reporte como **passed**.

```
> pruebas-vue@0.0.0 test:unit
> jest

 PASS  src/test/unit/HelloWorld.spec.js
  HelloWorld.vue
    ✓ renders props.msg when passed (12 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.709 s, estimated 1 s
Ran all test suites.
```

¿Cómo se genera el reporte de pruebas?

- Veamos el código del componente Helloworld.vue.
- Este componente solo muestra un mensaje recibido mediante la props msg.

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

¿Cómo se genera el reporte de pruebas?

- Veamos el código del archivo de pruebas example.spec.js.

```
import { shallowMount } from '@vue/test-utils'
import HelloWorld from '@/components/HelloWorld.vue'

describe('HelloWorld.vue', () => {
  it('renders props.msg when passed', () => {
    const msg = 'new message'
    const wrapper = shallowMount(HelloWorld, {
      props: { msg }
    })
    expect(wrapper.text()).toMatch(msg)
  })
})
```

Hagamos que la prueba no pase…

- Modifiquemos el código del componente `HelloWorld.vue` y en vez de retornar el `{{msg}}`, mostraremos Hola mundo.

```
<template>
 <div class="greetings">
   <h1 class="green">Hola mundo…</h1>
 </div>
</template>
```

```
> pruebas-vue@0.0.0 test:unit
> jest

 FAIL  src/test/unit/HelloWorld.spec.js
  HelloWorld.vue
    × renders props.msg when passed (8 ms)

  ● HelloWorld.vue › renders props.msg when passed

    expect(received).toMatch(expected)

    Expected substring: "new message"
    Received string:    "Hola mundo..."

       9 |     })
      10 |
    > 11 |     expect(wrapper.text()).toMatch(msg)
         |                            ^
      12 |   })
      13 | })
      14 |

      at Object.toMatch (src/test/unit/HelloWorld.spec.js:11:28)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.242 s
Ran all test suites.
```

{desafío} latam_

¿Qué podemos hacer para que la prueba anterior vuelva a pasar?

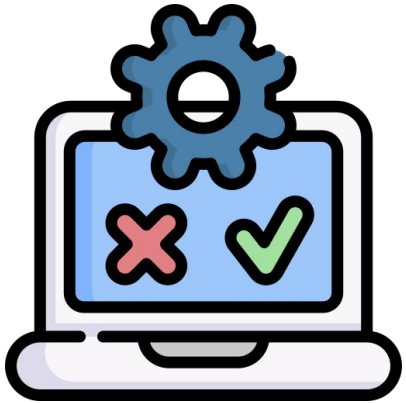
/* Desarrollo Dirigido por Test (TDD) */

¿En qué consiste?

Desarrollo Basado en Pruebas (Test Driven Development)

Consiste en que el código de nuestras aplicaciones esté basado en la escritura previa de código para que esta falle.

- Primero escribimos una prueba que falla.
- Luego, escribimos el código en la aplicación que hará que se cumplan los criterios de la prueba para que pase (***passed***).



/* El entorno de pruebas Vue Test Utils para Vue */

¿Qué es el Vue Test Utils?

Vue Test Utils le provee a Vue JS un conjunto de funciones útiles para simplificar el proceso de pruebas en nuestros componentes. Estas funcionalidades pueden ser mount, shallowMount, entre otras.

Veamos un ejemplo extraído de la documentación oficial acerca del método mount.

Método mount - shallowMount

Estos métodos permiten montar nuestros componentes en el entorno de los tests.

mount: Montará los componentres e incluso los hijos.

shallowMount: Monta únicamente el componente especificado.

```
// Import the `mount()` method from Vue Test Utils
import { mount } from '@vue/test-utils'

// The component to test
const MessageComponent = {
  template: '<p>{{ msg }}</p>',
  props: ['msg']
}

test('displays message', () => {
  // mount() returns a wrapped Vue component we can interact with
  const wrapper = mount(MessageComponent, {
    propsData: {
      msg: 'Hello world'
    }
  })

  // Assert the rendered text of the component
  expect(wrapper.text()).toContain('Hello world')
})
```

Mencionen una de las razones para realizar pruebas unitarias en una aplicación Vue JS.

{desafío} latam_

Resumen de la sesión

- Realizar pruebas unitarias en una aplicación Vue JS permitirá construir software de calidad y confiable.
- Las pruebas unitarias están enfocadas en probar funciones particulares dentro de las aplicaciones web.
- Al ejecutar pruebas antes de escribir el código final, generamos confianza y mitigamos los errores que se puedan generar.

¿Existe algún concepto visto en esta sesión que deseen reforzar?

Próxima sesión...

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

{desafío} latam_

{desafío}
latam_

Academia de
talentos digitales