Testeando componentes de **Vue.js** con **Jest**

Traducción de

Paul Melero

@paul_melero

Revisor y Autor original

Alex Jover

@alexjoverm

Testeando Componentes de Vue.js con Jest

Una guía concisa y práctica para testear componentes de Vue.js usando Jest y la librería oficial vue-test-utils

Alex Jover Morales y Paul Melero

Este libro está a la venta en http://leanpub.com/testeando-vuejs

Esta versión se publicó en 2018-12-26

Este es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener feedback del lector hasta conseguir tener el libro adecuado.

© 2018 Alex Jover Morales y Paul Melero

Índice general

1. Escribe tu Primer Test Unitario de un Componente de Vue.js con Jest

Vamos a aprender cómo escribir test unitarios con las herramientas oficiales de VueJS y el *framework* Jest.

vue-test-utils[1], es la librería oficial de testing de VueJS y está basada en su predecesora avoriaz[2]. @EddYerburgh[3] hizo un excepcional trabajo sentando las bases del testing en Vue con dicha librería.

Jest[4], por otro lado es un *framework* de testing desarrollado en Facebook, que hace que el testing unitario sea súper asequible, con funcionalidades como:

- Casi *0-config*, por defecto.
- Un "Modo interactivo".
- Ejecutar tests en paralelo.
- *Spies*, *stubs* y *mocks*.
- *Coverage* de código.
- Introduce el concepto de *Snapshot testing*.
- Y aporta utilidades de *Module mocking* (Simulación de módulos).

Probablemente ya hayas escrito tests sin estas herramientas, usando "solamente" Karma, Mocha, Chai, Sinon, Istanbul..., pero veréis que con Jest se facilita en gran medida el proceso .

[1] https://github.com/vuejs/vue-test-utils
[2] https://github.com/eddyerburgh/avoriaz
[3] https://twitter.com/EddYerburgh
[4] https://facebook.github.io/jest

Set up de un proyecto de ejemplo

Para el *scaffolding* vamos a utilizar `vue-cli`[5] y por defecto contestaremos NO a todas las preguntas que nos aparecen. Llamaremos al proyecto "vue-test":

```
npm install -g vue-cli
vue init webpack vue-test
cd vue-test
```

Lo siguiente será instalar algunas dependencias

```
# Instalar dependencias
npm i -D jest vue-jest babel-jest
```

`vue-jest`[6] se usa para que Jest entienda los archivos `.vue`, y `babel-jest`[7] para la integración con Babel.

Y además:

```
npm i -D @vue/test-utils
```

Vamos a añadir la siguiente configuración en nuestro `package.json`: (también puede ir en su propio archivo de configuración)

```
...
"jest": {
  "moduleNameMapper": {
    "^vue$": "vue/dist/vue.common.js"
  },
  "moduleFileExtensions": [
    "js",
    "vue"
  ],
  "transform": {
```

[5] https://github.com/vuejs/vue-cli
[6] https://github.com/vuejs/vue-jest
[7] https://github.com/babel/babel-jest

```
    "^.+\\.js$": "<rootDir>/node_modules/babel-jest",
    ".*\\.(vue)$": "<rootDir>/node_modules/vue-jest"
  }
}
...
```

`moduleFileExtensions` informa a Jest de qué extensiones tiene que buscar, y `transform` qué preprocesador usar para cada extensión.

Y por fin, añadimos un script `test` al `package.json`:

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

Testeando un Componente

Para los ejercicios, usaremos *Single File Components*, ya que no he probado qué tal funciona con archivos `html`, `css` y `js` sueltos, así que asumimos que estarás usando SFC también.

Crearemos un `ListaMensajes.vue` componente en `src/components`:

```
<template>
  <ul>
    <li v-for="mensaje in mensajes">{{ mensaje }}</li>
  </ul>
</template>

<script>
  export default {
    name: 'list',
    props: ['mensajes']
  }
</script>
```

Y lo añadimos al `App.vue` para que se utilice, de la siguiente manera:

```
<template>
  <div id="app"><ListaMensajes :mensajes="mensajes" /></div>
</template>

<script>
  import ListaMensajes from './components/ListaMensajes'

  export default {
    name: 'app',
    data: () => ({ mensajes: ['Hola Don Pepito', 'Hola Don José'] }),
    components: {
      ListaMensajes
    }
  }
</script>
```

Ya tenemos, entonces, dos componentes que podríamos testear. Crearemos la carpeta `test` en el directorio raíz del proyecto, y un `App.test.js`:

```
import Vue from 'vue'
import App from '../src/App'

describe('App.test.js', () => {
  let cmp, vm

  beforeEach(() => {
    cmp = Vue.extend(App) // Crea una copia del componente original
    vm = new cmp({
      data: {
        // Reemplazamos el estado local con datos _fake_
        mensajes: ['Gatito']
      }
    }).$mount() // Lo instanciamos y montamos
  })

  it('mensajes es igual a ["Gatito"]', () => {
    expect(vm.mensajes).toEqual(['Gatito'])
  })
})
```

Ahora, si ejecutamos `npm test` (o `npm t` como atajo), los tests deberían ejecutarse y pasar. Como vamos a estar modificando los tests, será mejor que lo ejecutemos en **watch mode**:

```
npm t -- --watch
```

El problema de los componentes anidados:

El test anterior era demasiado baladí. Vamos a comprobar que el *output* es también el esperado. Para ello utilizaremos la maravillosa funcionalidad de Jest que son los Snapshots. Generarán una foto o captura ("snapshot") del output y la comparará con ulteriores ejecuciones. Para ello, añadiremos después del `it()` anterior en `App.test.js`:

```
it('tiene la estructura html esperada', () => {
  expect(vm.$el).toMatchSnapshot()
})
```

Ésto creará un archivo `test/__snapshots__/App.test.js.snap`. Si inspeccionamos su contenido:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`App.test.js tiene la estructura html esperada 1`] = `
<div
  id="app"
>
  <ul>
    <li>
      Gatito
    </li>
  </ul>
</div>
`
```

Por si no te habías percatado, tenemos un problema aquí: el componente `ListaMensajes` se ha renderizado también. Y los **test unitarios tienen que ser unidades independientes**, con lo que en `App.test.js` intentaremos testear el componente `App` y no involucrarnos con nada más.

Si no seguimos esta norma, pronto nos encontraríamos con problemas. Por ejemplo si nuestro componente hijo (`ListaMensajes` en este caso) llevara a cabo *side effects* en el *hook* de `created()`, como hacer una petición `fetch()`, o a una acción de Vuex o que cambiara el estado local... Eso es algo que recomendaría evitar.

Por suerte, la funcionalidad del **Shallow Rendering** nos ayuda con este problema.

Qué es *Shallow Rendering*?

Shallow Rendering[8] ("renderizado superficial") es una técnica que nos asegura que nuestro componente se renderizará sin hijos. Lo cuá es útil para:

[8] http://airbnb.io/enzyme/docs/api/shallow.html

- Testear sólo el componente que queremos testear (que es lo que viene siendo el *testing* unitario).
- Evitar *side effects* que los componentes hijos pudieran causar, como llamadas HTTP, acciones del *store*...

Testeando con @vue/test-utils

`@vue/test-utils` nos brinda la funcionalidad del *Shallow Rendering*. Podríamos re-escribir el test anterior de la siguiente manera:

```
import { shallowMount } from '@vue/test-utils'
import App from '../src/App'

describe('App.test.js', () => {
  let cmp

  beforeEach(() => {
    cmp = shallowMount(App, {
      // Crea una instancia superficial del componente
      data: {
        mensajes: ['Gatito']
      }
    })
  })

  it('mensajes es igual a ["Gatito"]', () => {
    // En cmp.vm, podemos acceder a los métodos de la instancia componente
    expect(cmp.vm.mensajes).toEqual(['Gatito'])
  })

  it('tiene la estructura html esperada', () => {
    expect(cmp.element).toMatchSnapshot()
  })
})
```

A continuación, si todavía estamos en *watching mode*:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`App.test.js tiene la estructura html esperada 1`] = `
<div
  id="app"
>
  <!--  -->
</div>
`
```

Ves? AHora ningún hijo se renderiza y hemos testeado el componente `App` de manera **totalmente aislada** del árbol de componentes. De la misma manera, si hubiéramos añadido algún `created()` o algún otro hook en los componentes hijos, no se habrían llamado tampoco .

Si se te despierta la curiosidad sobre **cómo el *shallow render* se implementa**, puedes ver el código fuente[9] y verás que básicamente está sustituyendo las claves de los componentes.

Asimismo, puedes implementar `ListaMensajes.test.js` como sigue:

```
import { mount } from '@vue/test-utils'
import ListaMensajes from '../src/components/ListaMensajes'

describe('ListaMensajes.test.js', () => {
  let cmp

  beforeEach(() => {
    cmp = mount(ListaMensajes, {
      // Ojo: las `props` se sobreescriben con ``propsData`
      propsData: {
        mensajes: ['Gatito']
      }
    })
  })

  it('ha recibido ["Gatito"] como la propiedad mensajes', () => {
    expect(cmp.vm.mensajes).toEqual(['Gatito'])
```

[9] https://github.com/vuejs/vue-test-utils/blob/dev/packages/shared/stub-components.js

```
  })

  it('tiene la estructura html esperada', () => {
    expect(cmp.element).toMatchSnapshot()
  })
})
```

Encuentra los ejemplos completos en github[10].

[10] https://github.com/alexjoverm/vue-testing-series/tree/lesson-1