# Testing Vue.js components with Vue Test Utils

July 23, 2021 · 8 min read

I know, testing can seem like a waste of time for many developers. You hate it, right? But, should you? Should you test your components if you want to create a reliable application?

I will tell you my thoughts: testing your components (and, importantly, doing it the right way) is one of the best investments you can make if you are building something for the long run. But why?

In this guide, I'll answer these questions and summarize the benefits of testing your Vue.js components using Vue Test Utils by sharing a few stories that happened to me. 🤔

We'll cover the following:


- Why should you test Vue.js components?
- How should you test Vue.js components?
- What is Vue Test Utils?
- Testing Vue.js components with Vue Test Utils


## Why should you test Vue.js components?

When you're pushing code to production, you don't want to introduce bugs. If you're a talented developer who knows your codebase inside out, you might not (that said, I have seen plenty of excellent, self-assured engineers introduce rare conditions they did not see coming).

But what happens when you're overwhelmed with a lot of work because your company is growing and you need to hire some juniors to keep improving the product? Will they introduce bugs? Probably more often than you think.

When a junior developer pushes something that is breaking one of your most essential features, in a perfect world, you'd want to be notified before it hits the production servers. If your codebase is correctly tested, one of the tests will fail and you'll be able to fix the issue before any damage is done.

This is an important reason why you should test your codebase if you are building a project for the long run: developers work in a team and must protect each other. Some companies have even changed the way they code by introducing a methodology such as test-driven development (TDD) in their workflow. In short, this means you write the tests (i.e., the specs) before coding the business logic.

Another reason why you should test that your components are working correctly is that doing so provides documentation for each one. By reading the tests (which we will demonstrate in the coming sections), we can see what output we can expect for a given input (a prop, an event, etc.). And, as you probably already know, excellent documentation leads to easier debugging. 😃 📖

But if you ask me what I love the most about testing, it's how productive refactoring can become. When I started my unique path of becoming a web developer a few years ago, I quickly learned that a codebase is not static and changes a lot over time. In other words, you must refactor some part of it every week.

I remember when the product manager asked me to introduce a subfeature in one of the most critical interfaces. Unfortunately for me, it needed a complete refactoring of many components to make it work. I was afraid to break something, but this fear quickly disappeared. After I finished the refactoring, I was so happy to see that all tests passed through without triggering any error.

Confidence is key! In fact, this is yet another benefit of testing your Vue.js components. When you're confident that your code is working properly, you can be confident that you aren't shipping broken software. 😇

If you're still not convinced, here's more food for thought: fixing issues is usually far more costly than preventing them. The time it takes to write your tests is worth it.

## How should you test Vue.js components?

It's essential to talk about what we should be testing. For UI components, I don't recommend aiming to test every single line of code. This can lead to too much focus on the internal implementation of the component (i.e., reaching 100 percent test coverage).

Instead, we should write tests that assert the component public interface and treats it as an internal black box. A single test case would assert that some input (user actions, props, store) provided to the component results in the expected output (component rendering, vue events, function calls, etc.).

Also, last year, I watched a great talk by Sarah Dayan titled "Test-Driven Development with Vue.js" at Vue Amsterdam. In one of her slides, she said that to determine whether you should test one of your components (or a feature inside it), you must ask yourself: do I care about this if it changes? In other words, is it a feature that can cause issues in the interfaces if someone breaks it? If so, you should write a test to strengthen your code.

## What is Vue Test Utils?

Let's talk about the elephant in the room now. What is Vue Test Utils? 🤔

Vue Test Utils is an official library of helper functions to help users test their Vue.js components. It provides some methods to mount and interact with Vue.js components in an isolated manner. We refer to this as a wrapper. But what is a wrapper?

A wrapper is an abstraction of the mounted component. It provides some utility functions that make our lives easier, such as when we want to trigger a click or an event. We'll use this to execute some input (user actions, props,

store changes, etc.) so we can check that the output is correct (component rendering, Vue events, function calls, etc.).

What's remarkable is that you can grab the Vue instance with `wrapper.vm` if you don't have what you need on the wrapper. So you have a lot of flexibility.

You can find all the properties and methods available on the wrapper in the official Vue Test Utils documentation.

Vue Test Utils also allows mocking and stub components to be rendered with `shallowMount` or individual stubs, but we'll get to that later. So yes, this is a very complete and reliable library you will love. 😍

# Testing Vue.js components with Vue Test Utils

Now it's time to get our hands dirty and start testing our components with Vue Test Utils.

## Set up the infrastructure

You can choose between two test runners: Jest or Mocha and Chai. We'll go with Jest for this tutorial because it is recommended to use Vue Test Utils with Jest.

If you're not familiar with Jest, it's a test runner developed by Facebook. It aims to deliver a batteries-included unit testing solution.

If you're using the Vue CLI to build your project, here is how you can set up Vue Test Utils in your current Vue app.

> *For a manual installation, follow the **installation guide** in the docs.*

```
vue add unit-jest
npm install --save-dev @vue/test-utils
```

You should now see a new command added to `package.json` that we will use to run our tests.

```
{
  "scripts": {
    "test:unit": "vue-cli-service test:unit"
  }
}
```

## Testing our `HabitComponent`

It is now time to create our first suite of tests. For our example, we'll create a habit tracker. It will be composed of a single component, which we'll name `Habit.vue` , that we will tick every time we complete the habit. Inside your components folder, copy/paste the code below:

```
  margin-right: 20px;
}
.habit__box {
  width: 56px;
  height: 56px;
  display: flex;
  align-items: center;
  justify-content: center;
  border: 4px solid #cbd5e0;
  background-color: #ffffff;
  font-size: 40px;
  cursor: pointer;
  border-radius: 10px;
}
.habit__box--done {
  border-color: #22543d;
  background-color: #2f855a;
  color: white;
}
</style>
```

The component accepts a single prop (the title of the habit) and includes a box that becomes green when we click on it (i.e., the habit is complete).

In the `tests` folder at the root of your project, create a `Habit.spec.js`. We will write all our tests inside it.

Let's start by creating the wrapper object and write our first test.

```
import { mount } from "@vue/test-utils";
import Habit from "@/components/Habit";
describe("Habit", () => {
  it("makes sure the habit name is rendered", () => {
    const habitName = "Learn something new";
    const wrapper = mount(Habit, {
      propsData: {
        name: habitName,
      },
    });
    expect(wrapper.props().name).toBe(habitName);
    expect(wrapper.text()).toContain(habitName);
  });
});
```

If you run `npm run test:unit`, you should see that all tests are successful.

```
> vue-cli-service test:unit
 PASS  tests/unit/Habit.spec.js
  Habit
    ✓ makes sure the habit name is rendered (11ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.135s
Ran all test suites.
```

Now let's make sure our habit is checked when we click on the box.

```
it("marks the habit as completed", async () => {
  const wrapper = mount(Habit, {
    propsData: {
      name: "Learn something new",
    },
  });
  const box = wrapper.find(".habit__box");
  await box.trigger("click");
  expect(box.text()).toContain("✓");
});
```

Notice how the test must be async and that trigger needs to be awaited. Check out the "Testing Asynchronous Behavior" article in the Vue Test Utils docs to understand why this is necessary and other things to consider when testing asynchronous scenarios.

```
> vue-cli-service test:unit
 PASS  tests/unit/Habit.spec.js
  Habit
    ✓ makes sure the habit name is rendered (11ms)
    ✓ marks the habit as completed (10ms)
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.135s
Ran all test suites.
```

We can also verify that the `onHabitDone` method is called when we click on it.

```
it("calls the onHabitDone method", async () => {
  const wrapper = mount(Habit, {
    propsData: {
      name: "Learn something new",
    },
  });
  wrapper.setMethods({
    onHabitDone: jest.fn(),
  });
  const box = wrapper.find(".habit__box");
  await box.trigger("click");
  expect(wrapper.vm.onHabitDone).toHaveBeenCalled();
});
```

Run `npm run test:unit` and everything should be green.

Here's what you should see in your terminal:

```
> vue-cli-service test:unit
 PASS  tests/unit/Habit.spec.js
  Habit
    ✓ makes sure the habit name is rendered (11ms)
    ✓ marks the habit as completed (10ms)
    ✓ calls the onHabitDone method (2ms)
Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.135s
Ran all test suites.
```

We can even check that the component behaves as expected when we change a prop.

```
it("updates the habit method", async () => {
  const wrapper = mount(Habit, {
    propsData: {
      name: "Learn something new",
    },
  });
  const newHabitName = "Brush my teeth";
  await wrapper.setProps({
    name: newHabitName,
  });
  expect(wrapper.props().name).toBe(newHabitName);
});
```

Here's what you should see in your terminal:

```
> vue-cli-service test:unit
 PASS  tests/unit/Habit.spec.js
  Habit
    ✓ makes sure the habit name is rendered (11ms)
    ✓ marks the habit as completed (10ms)
    ✓ calls the onHabitDone method (2ms)
    ✓ updates the habit method (2ms)
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.135s
Ran all test suites.
```

To help you code faster, here are the wrapper methods I use the most:

- `wrapper.attributes()` : Returns Wrapper DOM node attribute object
- `wrapper.classes()` : Return Wrapper DOM node classes
- `wrapper.destroy()` : Destroys a Vue component instance
- `wrapper.emitted()` : Return an object containing custom events emitted by the Wrapper vm
- `wrapper.find()` : Returns Wrapper of first DOM node or Vue component matching selector

- `wrapper.findAll()` : Returns a WrapperArray
- `wrapper.html()` : Returns HTML of Wrapper DOM node as a string
- `wrapper.isVisible()` : Assert Wrapper is visible
- `wrapper.setData()` : Sets Wrapper vm data
- `wrapper.setProps()` : Sets Wrapper vm props and forces update
- `wrapper.text()` : Returns text content of Wrapper
- `wrapper.trigger()` : Triggers an event asynchronously on the Wrapper DOM node

## Using `fetch`

If you use the `fetch` method inside your component to call an API, you will get an error. Here's how you can make sure `fetch` is defined during your tests.

```
npm install -D isomorphic-fetch
```

Then update your `package.json` .

```
{
  "scripts": {
    "test:unit": "vue-cli-service test:unit --require isomorphic-fetch"
  }
}
```

## `mount` vs. `shallowMount`

You may find that some people are using `shallowMount` instead of `mount` . The reason is that, like `mount` , it creates a wrapper that contains the mounted and rendered Vue.js component, but with stubbed child components.

This means the component will be rendered faster because all its children components will not be computed. Be careful, though; this approach can lead to some trouble if you're trying to test something linked to a child's

component.

## Where do we go from here?

The Vue Test Utils documentation is a great resource to help you get started — especially the guides, which are updated every month. The page with all the wrapper methods and the Jest API are both excellent resources you should bookmark as well.

Remember, practicing and writing your tests for your project is the best way to start learning. I hope this guide helps you grasp how robust testing your components can be. And that this is not very hard. 😃

We will end this guide with a quote by renowned computer scientist Donald Knuth: "Computers are good at following instructions, but not at reading your mind."

I'd be happy to read your comments and your Twitter messages @RifkiNada. And in case you are curious about my work, you can check it out at NadaRifki.com.

## Experience your Vue apps exactly how a user does

Debugging Vue.js applications can be difficult, especially when there are dozens, if not hundreds of mutations during a user session. If you're interested in monitoring and tracking Vue mutations for all of your users in production, try LogRocket. https://logrocket.com/signup/

LogRocket is like a DVR for web and mobile apps, recording literally everything that happens in your Vue apps including network requests, JavaScript errors, performance problems, and much more. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred.

The LogRocket Vuex plugin logs Vuex mutations to the LogRocket console, giving you context around what led to an error, and what state the application was in when an issue occurred.

Modernize how you debug your Vue apps - Start monitoring for free.

Nada Rifki Follow

Nada is a JavaScript developer who likes to play with UI components to create interfaces with great UX. She specializes in Vue.js and loves sharing anything and everything that could help her fellow frontend web developers. Nada also dabbles in digital marketing, dance, and Chinese.

#vue

**Leave a Reply**

Enter your comment here...