

# What to test

Have you ever deployed a new feature in your app and crossed your fingers, hoping you don't wake up to the news that you've accidentally introduced a brand new bug? You can greatly reduce these kinds of concerns, and strengthen your Vue apps, by testing your application.

A thoroughly tested application generally consists of a well implemented combination of several kinds of testing, including End to End (E2E) testing, sometimes Integration testing, Snapshot testing, and Unit Testing. This course specifically serves as a beginner's guide to unit testing in Vue. As you'll see throughout the course, unit tests serve as the foundation of a well tested app.

In this course, we'll be using the popular [Jest](#) JavaScript testing library to run our tests, along with [Vue Test Utils](#), the official unit testing utility library for Vue.

## Goals of writing tests

In the following lesson, we'll write our first test. But first, let's gain a clear understanding of the benefits of testing your apps. What do we hope to gain from all of this added work of writing more code to test our code?

### Boosted Confidence

Beyond sleeping better at night after a new feature deployment, testing helps bring everyone in your team onto the same page. If you're new to a repository, seeing a suite of tests is like having an experienced developer sitting right next to you and watching you code, making sure you are staying within the proper lanes of what the code ought to do. With these tests in place, you can feel confident you aren't breaking anything when adding new functionality or changing existing code.

### Quality Code

When you write your components with testing in mind, you end up creating isolated, more reusable components. If you start writing tests for your components and you notice they aren't easy to test, this is a clear sign that you can probably refactor your components, which can end up improving them.

### Better Documentation

As alluded to in the first point, another result of testing is that it can end up producing good documentation for your development team. When someone is new to a code base, they can look to the tests for guidance, which can provide insight around the intentions for how the components should work, and provide clues to edge cases that may be tested for.

## Identifying what to test

So we know that testing is valuable, but *what exactly* should we be testing in our applications? It's easy to go overboard and test things you don't have to, which unnecessarily slows down development time. So what do we test in a Vue.js app? The answer is actually quite simple: components. Since a Vue app is merely a puzzle of interlocking components, we need to be testing their individual behavior to make sure they're working correctly.

### The Component Contract

When first learning about Vue unit testing myself, I found it helpful when Ed Yerburch (who literally wrote [the book](#) on Testing Vue.js Applications) spoke about thinking through the *component contract*. By that, we're referring to the agreement between a component and the rest of the application.

For example, imagine a component that takes in a `min` and `max` prop and generates a random number between those prop values, which it renders to the DOM. The component contract says: I will accept two props and use them to produce a random number. Inherent within this contract is the concept of **inputs** and **outputs**. The component agrees to receive the `min` and `max` props as **inputs** and to deliver a random number as the **output**. As such, we can start to pick apart what we should be testing by thinking through the component contract, and identifying the inputs and outputs.

At a high level, common **inputs** and **outputs** are as follows:

#### Inputs

- Component Data
  - Component Props
    - User Interaction
      - Ex: user clicks a button
- Lifecycle Methods
  - `mounted()`, `created()`, etc.
- Vuex Store
- Route Params

#### Outputs

- What is rendered to the DOM
- External function calls
- Events emitted by the component
- Route Changes
- Updates to the Vuex Store
- Connection with children
  - i.e. changes in child components

By focusing on these specifically, you avoid focusing on the internal business logic. In other words, we shouldn't be getting bogged down by worrying about how every single line of code works. It may seem counterintuitive, but the goal of unit testing is purely to ensure your components are producing the expected results. We aren't concerned here about *how* it arrived at that result. We may even change the way we're logically arriving at that result later on, so we don't want our tests to be unnecessarily prescriptive about how that ought to be achieved. It's up to your team to figure out the most efficient path to that result. That's not the job of testing. As far as a unit test is concerned, if it works, it works.

Now that we know what we should be testing for, let's look at a couple basic examples and identify what we might test in each.

## Example: AppHeader Component

In this example, we have a component that will display a logout button if the `loggedIn` property is `true`.

### AppHeader.vue

```
<template>
<div>
  <button v-show="loggedIn">Logout</button>
</div>
</template>

<script>
export default {
  data() {
    return {
      loggedIn: false
    }
  }
}
</script>
```

To get clear on what part of this component we ought to test, our first step is to identify the component's inputs and outputs.

#### Inputs

- Data ( `loggedIn` )
  - This data property determines if the button shows or not, so this is an input that we should be testing

#### Outputs

- Rendered Output ( `button` )
  - Based on the inputs ( `loggedIn` ), is our button being displayed in the DOM when it should be?

With more complex components, there will be more aspects of it to test, but the same general approach applies. While getting a feel for what you ought to be testing is certainly helpful, so is knowing what you should **not** be testing. So let's unpack that now.

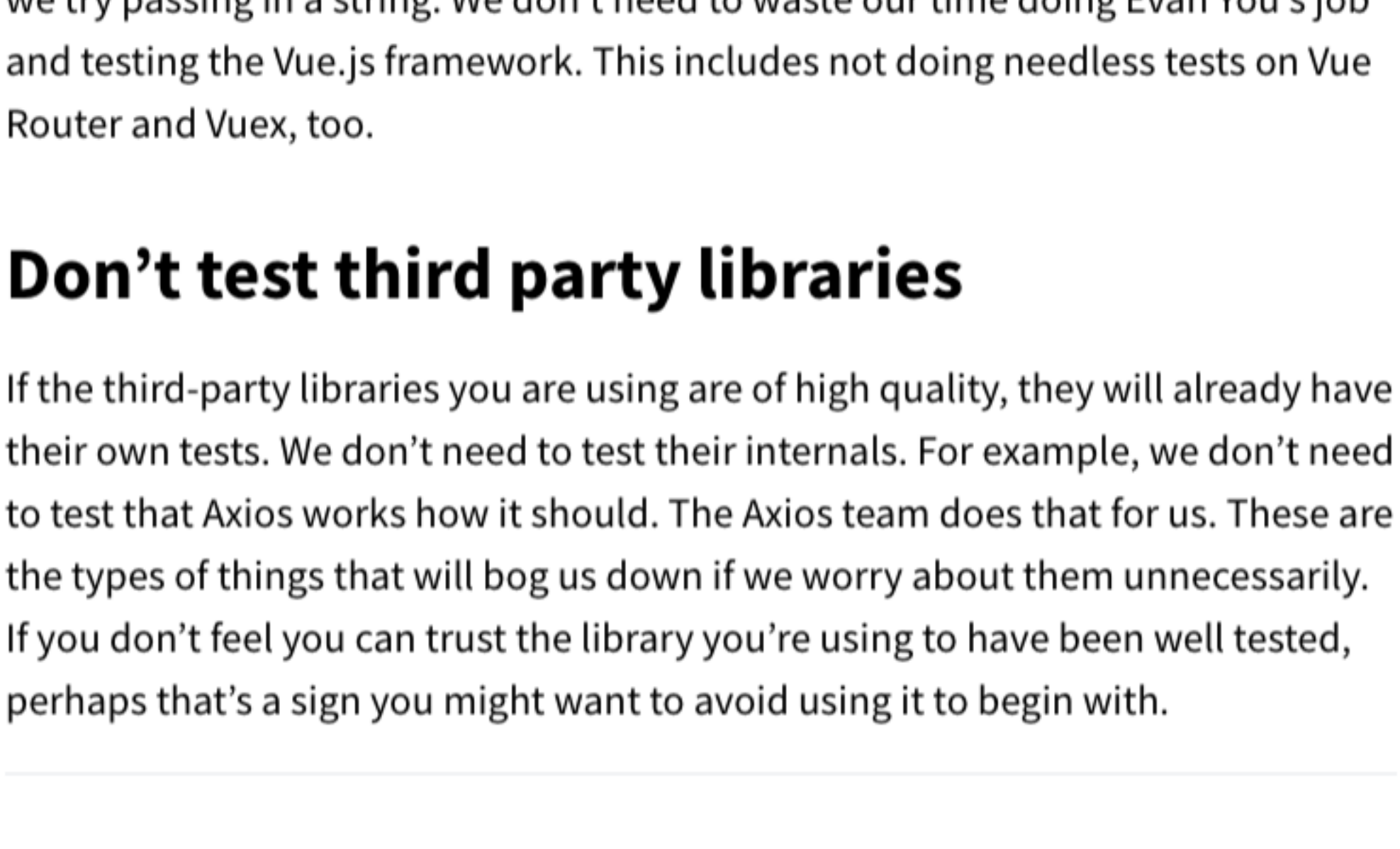
## What NOT to test

Understanding what doesn't need to be tested is an important part of the testing story that many developers don't think about, and in turn it costs them a lot of time that could be spent elsewhere.

Let's look again at the example from earlier, where we have a component that takes in a `min` and `max` prop and outputs a random number within that range. We already know we **should** be testing the the output that is rendered to the DOM, and to do so we'll need to be considering the `min` and `max` props in our test. But what about the actual method that is generating the random number? Do we need to test that?

The answer is no. Why? Because we don't need to get lost in the implementation details.

What not to test



The diagram illustrates the concept of 'What not to test'. It shows a funnel at the top with two input boxes containing the numbers '1' and '10'. An arrow points from these inputs into a central box. This central box is divided into two sections: the top section is labeled 'Implementation details' and has a red 'X' icon next to it, indicating that this is what should not be tested. The bottom section of the central box contains a circular arrow icon, representing the internal logic or implementation. An arrow points from the bottom of the central box to an output box containing the number '6', which has a green checkmark icon next to it, indicating that this is what should be tested.

## Don't test implementation details

When unit-testing we don't need to fuss over *how* certain things work, just that they *do* work. We don't need to set up a test that calls the function that generates our random number, making sure it behaves in a certain way. We don't care about the internals here. We just care that the component produced the output we are expecting. This way, we can always come back later and replace the implementation logic (with a third-party library for generating random numbers, for example).

## Don't test the framework Itself

Developers often try to test too much, including the inner workings of the framework itself. But the framework authors already have tests set up to do that. For example, if we set up some prop validation for our `min` and `max` props, specifying they need to be a `Number`, we can trust that Vue will throw an error if we try passing in a string. We don't need to waste our time doing Evan You's job and testing the Vue.js framework. This includes not doing needless tests on Vue Router and Vuex, too.

## Don't test third party libraries

If the third-party libraries you are using are of high quality, they will already have their own tests. We don't need to test their internals. For example, we don't need to test that Axios works how it should. The Axios team does that for us. These are the types of things that will bog us down if we worry about them unnecessarily. If you don't feel you can trust the library you're using to have been well tested, perhaps that's a sign you might want to avoid using it to begin with.

## Let's ReVue

In this lesson, we took an important first step before writing effective unit tests: identifying what you should and shouldn't be testing in components. With this approach, we can focus our time wisely on testing the pieces that *need* to be tested. In the next lesson, we'll take this knowledge and write our first unit test.



# Writing a Unit Test with Jest

Write your first unit test using Jest and Vue Test Utils.

**Create a free account to unlock it.**



Unlock Content

# Testing Emitted Events

Learn how to test that your component's custom events were emitted with the `expectToEmit` method.

**Unlock this lesson by subscribing to a plan.**



Unlock Content

# Testing API Calls

Learn how to test your data-fetching components.

**Unlock this lesson by subscribing to a plan.**



Unlock Content

# Stubbing Child Components

How do we test parent components whose children have a lot of baggage?

**Unlock this lesson by subscribing to a plan.**



Unlock Content