

Watchers

In this lesson, we'll learn how to test part of the logic of watchers.

WE'LL COVER THE FOLLOWING



- Testing Watchers
- What Should We Test About the Watch Function?
 - Method 1:
 - Method 2:
- Wrapping Up

There are rare cases where we need to use watchers instead of computed properties. Sometimes watchers are misused as well, which messes things up and results in an unclear data workflow among components. So before using watchers, we should think about their use and suitability.

As we can see in the [Vue.js docs](#), watchers are often used to react to data changes and perform asynchronous operations, such as performing an ajax request.

Testing Watchers

Let's say we want to do something when the `inputValue` from the state changes. We could do an ajax request, but since that's more complicated (as we'll see it in the next lesson) let's just do a `console.log`. Add a `watch` property to the `Form.vue` component options:

```
require('./check-versions')()

process.env.NODE_ENV = 'production'

var ora = require('ora')
var rm = require('rimraf')
var path = require('path')
var chalk = require('chalk')
```

```

var webpack = require('webpack')
var config = require('../config')
var webpackConfig = require('./webpack.prod.conf')

var spinner = ora('building for production...')
spinner.start()

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {
  if (err) throw err
  webpack(webpackConfig, function (err, stats) {
    spinner.stop()
    if (err) throw err
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
      children: false,
      chunks: false,
      chunkModules: false
    }) + '\n\n')

    console.log(chalk.cyan('  Build complete.\n'))
    console.log(chalk.yellow(
      '  Tip: built files are meant to be served over an HTTP server.\n' +
      '  Opening index.html over file:// won\'t work.\n'
    ))
  })
})
})


```

Notice that the `inputValue` watch function matches the state variable name. By convention, Vue will look it up in both `properties` and `data` state by using the watch function name, which in this case is `inputValue`. Vue will find it in `data` and will add the watcher there.

See how a watch function takes the new value as a first parameter and the old one as the second. In this case, we've chosen to log only when it's not empty and the values are different. Usually, we'd like to write a test for each case, depending on the time we have and how critical that code is.

What Should We Test About the Watch Function?

This is also something we'll discuss further in the next lesson when we talk about testing methods, but let's say we just want to know if it calls the `console.log` when it should. So, let's add the bare bones of the watcher's test suite, within `Form.test.js`:

 `Form.test.js`

```
describe("Form.test.js", () => {
```

```

let cmp;

describe("Watchers - inputValue", () => {

  let spy;

  beforeAll(() => {
    spy = jest.spyOn(console, "log");
  });

  afterEach(() => {
    spy.mockClear();
  });

  it("is not called if value is empty (trimmed)", () => {
    // TODO
  });

  it("is not called if values are the same", () => {
    // TODO
  });

  it("is called with the new value in other cases", () => {
    // TODO
  });
});
});

```

We're using a spy on the `console.log` method, initializing before starting any test, and resetting its state after each of them so that they start from a clean spy.

Method 1:

To test a watch function, we just need to change the value of what's being watched; in this case, the `inputValue` state. But there is something curious. Let's start from the last test:

```

it("is called with the new value in other cases", () => {
  cmp.vm.inputValue = "foo";
  expect(spy).toBeCalled();
});

```

We changed the `inputValue`, so the `console.log` spy should be called, right? Well, it won't. But there is an explanation for this: unlike computed properties, watchers are **deferred to the next update cycle** that Vue uses to look for changes. So what's happening here is that `console.log` is indeed called, but only after the test has finished.

Notice that we're changing `inputValue` in a raw way (accessing the `vm` property). If we wanted to do with another way we'd need to use the

property). If we wanted to do with another way, we'd need to use the `vm.$nextTick` function to defer code to the next update cycle:

```
it("is called with the new value in other cases", done => {
  cmp.vm.inputValue = "foo";
  cmp.vm.$nextTick(() => {
    expect(spy).toBeCalled();
    done();
  });
});
```

Notice that we call a `done` function that we receive as a parameter. That's *one way Jest* has to test asynchronous code.

Method 2:

However, there is a **much better way**. The methods that `vue-test-utils` give us, such as `emitted` or `setData`, take care of that under the hood. So the last test can be written in a cleaner way just by using `setData`:

```
it("is called with the new value in other cases", () => {
  cmp.setData({ inputValue: "foo" });
  expect(spy).toBeCalled();
});
```

We can apply the same strategy for the next one, with the difference that the spy shouldn't be called:

```
it("is not called if value is empty (trimmed)", () => {
  cmp.setData({ inputValue: "  " });
  expect(spy).not.toBeCalled();
});
```

Finally, testing that *is not called if values are the same* is a bit more complex. The default internal state is empty, so first, we need to change it, wait for the next tick, then clear the mock to reset the call count, and change it again. Then after the second tick, we can check the spy and finish the test.

This can get simpler if we recreate the component at the beginning, overriding the `data` property. Remember we can override any component option by using the second parameter of the `mount` or `shallowMount` functions:

```
it("is not called if values are the same", () => {
  cmp = shallowMount(Form, {
    data: () => ({ inputValue: "foo" })
  });
```

```
data: () => ({ inputValue: 'foo' })
});
cmp.setData({ inputValue: 'foo' });

expect(spy).not.toBeCalled();
});
```

Wrapping Up

In this chapter, you've learned how to test a part of the logic of Vue components: computed properties and watchers. We've gone through the different test cases we might come across testing them. Hopefully, you've also learned some of the Vue internals such as the `nextTick` update cycles.

Let's test a running project of what we have done so far in the next lesson.