Properties

In this lesson, we will be testing Component Properties.

> **Properties** are custom attributes passed from parent to child components.

> **Custom events** solve just the opposite; they send data out to the direct parent via an event.

Combined, they are the wires of interaction and communication in Vue.js components.

There are different ways to test properties, events and custom events.

In Unit Testing, testing **ins** and **outs** (properties and custom events) means testing how a component behaves when it receives and sends data in isolation.

Testing Component Properties

When we are testing component properties, we can test how the components behave when we pass them certain properties. But before proceeding, there is an important note:

> To pass properties to components, use `propsData,` and not `props.` The latter is used to define properties, not to pass the data.

First, create a `Message.test.js` file and add the following code:

```
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'
    ))
  })
})
```

Note: `Message.test.js` will fail for now with the error: `Your test suite must contain at least one test.`

We can group test cases within a `describe` expression, and they can be nested. We can use this strategy to group the tests for properties and events separately.

Then we'll create a helper factory function to create a message component, giving it some properties, i.e. `propsData`:

```
const createCmp = propsData => mount(Message, { propsData });
```

Testing Property Existence

One obvious thing we can test is whether a property exists or not. Recall that the `Message.vue` component has a `message` property, so, let's assert that it receives the property correctly. `vue-test-utils` comes with a `hasProp(prop, value)` function, which is very handy for this case:

Message.vue

```
it("has a message property", () => {
  cmp = createCmp({ message: "hey" });
  expect(cmp.hasProp("message", "hey")).toBeTruthy();
});
```

The properties behave in such a way that they will be received only if they're declared in the component. This means that if we pass a **property that is not defined, it won't be received.** So to check for the non-existence of a property, you could use a non-existing property:

```
it("has no cat property", () => {
  cmp = createCmp({ cat: "hey" });
  expect(cmp.hasProp("cat", "hey")).toBeFalsy();
});
```

However, in this case, the test will fail because Vue has non-props attributes which sets it to the root of the `Messagecomponent`, thus being recognized as a prop. Hence, the test will return `true`. Changing it to `toBeTruthy` will make it pass for this example:

```
it("has no cat property", () => {
  cmp = createCmp({ cat: "hey" });
  expect(cmp.hasProp("cat", "hey")).toBeTruthy();
});
```

We can test the **default value** as well. Go to `Message.vue` and change the props as follows:

Message.vue

```
props: {
  message: String,
  author: {
    type: String,
    default: 'Paco'
  }
},
```

Then the test could be:

```
it("Paco is the default author", () => {
  cmp = createCmp({ message: "hey" });
  expect(cmp.hasProp("author", "Paco")).toBeTruthy();
});
```

Asserting Property Validation

Properties can have validation rules, ensuring that a property is required or is of a determined type. Let’s write the `message` property as follows:

```
props: {
  message: {
    type: String,
    required: true,
    validator: message => message.length > 1
  }
}
```

Going further, you could use custom constructor types or custom validation rules, as you can see in the docs. Let’s consider this example:

```
class Message {}
...
props: {
  message: {
    type: Message,
    ...
    }
  }
}
```

Whenever a validation rule is not fulfilled, Vue shows a console.error. For example, for `createCmp({ message: 1 })`, the following error will be shown:

> ```
> [Vue warn]: Invalid prop: type check failed for prop >"message". Expected String, got Number.
> (found in <Root>)
> ```

`vue-test-utils` doesn’t have any utility to test this. We could use `jest.spyOn` to test it:

```
it("message is of type string", () => {
  let spy = jest.spyOn(console, "error");
  cmp = createCmp({ message: 1 });
  expect(spy).toBeCalledWith(
    expect.stringContaining("[Vue warn]: Invalid prop")
  );

  spy.mockReset(); // or mockRestore() to completely remove the mock
});
```

Here, we’re spying on the console.error function, and checking if it shows a message containing a specific string. This is not an ideal way to check since we’re spying on global objects and relying on side effects.

Fortunately, there is an easier way to do it, which is by checking `vm.$options`. This is where Vue stores the component options “expanded”. With expanded, you can define your properties in a different way:

```
props: ["message"];

// or

props: {
  message: String;
}

// or

props: {
  message: {
    type: String;
  }
}
```

But due to this, all components will end up in the most expanded object form (like the last one). So if we check `cmp.vm.$option.props.message`, they all will be in the `{ type: X }` format (though for the first example it will be `{ type: null }`) With this in mind, we could write a test suite to test if the `message` property has the expected validation rules:

```
describe('Message.test.js', () => {
  ...
  describe('Properties', () => {
    ...

    describe('Validation', () => {
      const message = createCmp().vm.$options.props.message

      it('message is of type string', () => {
        expect(message.type).toBe(String)
      })

      it('message is required', () => {
        expect(message.required).toBeTruthy()
      })

      it('message has at least length 2', () => {
        expect(message.validator && message.validator('a')).toBeFalsy()
        expect(message.validator && message.validator('aa')).toBeTruthy()
      })
    })
```

In the next lesson, we’ll learn about how to test Custom Events.