SAMPLE CHAPTER

TESTING Vue.js Applications

Edd Yerburgh

MANNING

Testing Vue.js Applications

by Edd Yerburgh

Sample Chapter 5

brief contents

5 Testing events

This chapter covers

- Testing native DOM events
- Testing custom Vue events
- Testing input elements

If money makes the world go around, then events make web apps go around. Without events, websites would be static HTML pages; with events, websites can become powerful applications that respond to user interactions.

In Vue applications, you will encounter two types of events: native DOM events and Vue custom events. In this chapter you'll learn about both types of events and how to test them.

So far in this book, you've written unit tests for components in a Hacker News application. The Hacker News application is a great example of a real-world app. But there's one problem—it doesn't use any events!

Events are an important part of most Vue applications, and before you can call yourself a testing master, you should know how to test them. In this chapter, you'll take a break from the Hacker News app. Instead, you'll write a pop-up email

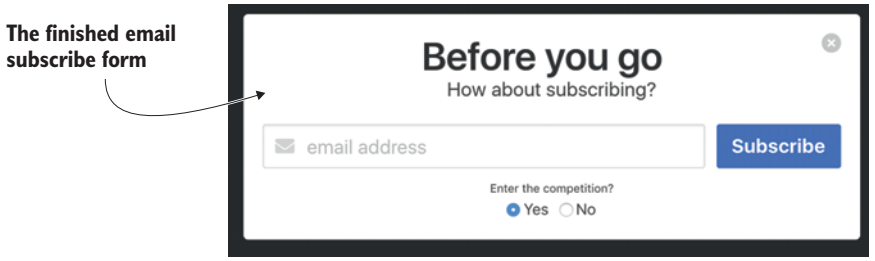

Figure 5.1 The finished pop-up email subscribe form that you'll create in this chapter

subscribe form—you know, the kind of form that appears when your mouse leaves a page and asks you to *subscribe for more content* (figure 5.1).

The sign-up form will be made from three components—a `Modal` component, a `Form` component, and an `App` component. There's already some code for each of these components added in the starter branch of the project.

> **NOTE** To follow this chapter, you need to clone the chapter-5 project and check out the starter branch. You can find instructions to do this in appendix A.

The first section of this chapter is about testing native DOM events. To learn how to test DOM events, you'll write tests for the `Modal` component. In the second section, you'll refactor the `Modal` component to use Vue custom events.

The third section of this chapter focuses on testing input forms. Input forms have some nuances that can be tricky when you first encounter them. You'll write tests for the `Form` component to see how to test text-input elements and radio buttons.

The final section of this chapter is about the limitations of jsdom. So far, jsdom has been working great, but you're going to find that there are some issues with using a pure JavaScript DOM for unit tests.

After you have cloned the project and run `npm install`, you can begin. The first topic you'll learn is how to test native DOM events.

5.1 Testing native DOM events

Native DOM events are how the browser alerts JavaScript code that something interesting has happened. Lots of DOM events are available. For example, clicking an element triggers a `click` event, hovering the cursor over an element triggers a `mouseenter` event, and submitting a form triggers a `submit` event.

> **NOTE** If you aren't familiar with native DOM events, MDN has a great primer you can read at http://mng.bz/mmEa.

In Vue apps you can use event listeners to respond to events with handler functions. Each time an event is triggered on the element, the handler function is called.

For example, to increment a `count` value when a button is clicked, you would use a `v-on` directive with a `click` argument as follows:

```
<button v-on:click="count++">Add 1</button>
```

Vue provides a shorthand notation for the `v-on` directive, which I'll use in this chapter. The following is the same example using the `@` shorthand:

```
<button @click="count++">Add 1</button>
```

Native DOM events are often the input for a component unit test. For example, imagine you wanted to test that clicking a button hides an element. In the test, you would trigger a `click` event on the button element and then assert that the element is removed from the DOM.

In this section you're going to add a test to check that the `Modal` component calls an `onClose` prop when a close button is clicked. To test that a prop is called when a `<button>` element is clicked, you need a way to dispatch a `click` event on the `<button>` element. You can do that with the Vue Test Utils `trigger` method.

5.1.1 Using the Vue Test Utils trigger method

In Vue Test Utils, every wrapper has a `trigger` method that dispatches a *synthetic event* on the wrapped element.

> **DEFINITION** A synthetic event is an event created in JavaScript. In practice, a synthetic event is processed the same way as an event dispatched by a browser. The difference is that native events invoke event handlers asynchronously via the JavaScript event loop; synthetic events invoke event handlers synchronously.

`trigger` takes an `eventType` argument to create the event that is dispatched on the wrapper element. For example, you could simulate a `mouseenter` event on a `<div>` by calling `trigger` on the wrapper of a `<div>` element, as shown in the next listing.

Listing 5.1 Triggering a `mouseenter` event

```
const wrapper = shallowMount(TestComponent)
wrapper.find('div').trigger('mouseenter')
```

> **NOTE** The `trigger` method can be used to simulate any DOM event. For example, it could simulate an `input` event, a `keydown` event, or a `mouseup` event.

The test that you'll write checks that an `onClose` prop is called when a button is clicked. To test this, you need to create a mock function, pass it in as an `onClose` prop, and then dispatch a click event on a button using `trigger`. Then you can assert that the mock function was called correctly.

Add the code from the following listing into the `describe` block in src/components/__tests__/Modal.spec.js.

Listing 5.2 Triggering a test by dispatching a DOM event

```
test('calls onClose when button is clicked', () => {
  const onClose = jest.fn()
  const wrapper = shallowMount(Modal,
    propsData: {
      onClose
    }
  })
  wrapper.find('button').trigger('click')
  expect(onClose).toHaveBeenCalled()
})
```

Creates a mock to pass to the modal component

Shallow-mounts the modal component with an onClose prop

Dispatches a DOM click event on a button element

Asserts that the mock was called

You'll notice there's an existing test in the file, which checks that the modal renders the default slot content. This is part of the `Modal` component's contract: it renders a default slot. You don't need to change this test.

> **NOTE** You might be concerned about using a generic `<button>` tag selector to find a rendered node. Theoretically, you might add an extra `<button>` element to the `Modal` before the close button in the DOM structure, which would cause the test to break. Unfortunately, this is an unavoidable aspect of unit testing Vue components. Some developers add IDs to elements to avoid this coupling, but I find that adding attributes for testing is often unnecessary. In my experience, it's rare that you need to update a test because you decide to change the DOM structure of a component.

To make both tests pass, you need to update the `Modal` component to call the `onClose` prop when the close button is clicked. Add the code from the next listing to src/components/Modal.vue.

Listing 5.3 Calling a prop when button is clicked

```
<template>
  <div>
    <button
      @click="onClose"
    />
    <slot />
  </div>
</template>

<script>
export default {
  props: ['onClose']
}
</script>
```

Adds onClose as a click handler using v-on directive @ shorthand

Run the tests to make sure they pass: `npm run test:unit`. Congratulations—you've learned how to trigger native DOM events in tests.

Most of the time when you test native DOM events, you call `trigger` with the correct event name. Sometimes, though, your code will use values from the event target. You'll learn how to write tests that use the event target later in section 5.3.

Before you learn how to test forms, you'll learn how to test Vue custom events. If you open src/App.vue, you'll see that you're passing a `closeModal` method as the `onClose` prop to the `Modal` component. The `closeModal` method sets `displayModal` to `false`, so the `App` component won't render the `Modal` component. This implementation is fine, but to teach you how to test Vue custom events, I'm going to have you refactor the `App` to listen to a Vue custom event instead.

5.2 Testing custom Vue events

What's better than native DOM events? Vue custom events! Vue has its own custom event system, which is useful for communicating to a parent component.

Custom events are emitted by Vue instances. Just like DOM events, components can listen for Vue events on child components with the `v-on` directive as follows:

```
<my-custom-component @custom-event="logHello" />
```

> **NOTE** if you want to read more about Vue custom events, check out the Vue docs at http://mng.bz/5N4O.

Custom events are useful for communicating from a child component to a parent component. A child component can emit a custom event, and a parent component can decide how to respond to the event.

There are two parts to the Vue custom event system—the parent component that listens to a custom event and the component that emits the event. That means there are two different testing techniques to learn:

- For components that emit events, the emitted event is the component *output*.
- For parent components that listen to custom events, an emitted event is the component *input*.

To learn how to test Vue custom events, you'll refactor the app to use Vue custom events instead of native DOM events in the `Modal` and `App` components. You'll start by rewriting the `Modal` component to emit a custom Vue event.

5.2.1 Testing that components emit custom events

Vue custom events are emitted by a component instance with the Vue instance `$emit` method. For example, to emit a `close-modal` event, make a call like this from the component:

```
this.$emit('close-modal')
```

Emitting a custom event is part of a component's contract. Other components will rely on a child component emitting an event, which means it's important to test that a component emits an event when it's provided the correct input.

You can test that a component emits an event with the Vue Test Utils `emitted` method. Calling `emitted` with an event name returns an array that includes the payload sent in each emitted event.

> **NOTE** You can use `emitted` to test that an event was called in the correct order or with the correct data. Check out the Vue docs to see other examples of testing with emitted—http://mng.bz/6jOe.

You're going to write a new test for the `Modal` component to check that a `close-modal` event is emitted when the close button is clicked. You can do this by dispatching a click on the button and then asserting that a `close-modal` event was emitted by the component instance using the `emitted` method. Open src/components/__tests__/Modal.spec.js. Delete the *calls* `onClose` *when button is clicked* test, and add the test from the following listing.

Listing 5.4 Testing that a component emits an event

```
test('emits on-close when button is clicked', () => {
  const wrapper = shallowMount(Modal)
  wrapper.find('button').trigger('click')
  expect(wrapper.emitted('close-modal')).toHaveLength(1)
})
```

Dispatches a DOM click event on a button element

Asserts that close-modal was emitted once

Run the tests to make sure the new test fails: `npm run test:unit`. You can make the test pass by refactoring the `Modal` component to emit an event when the button is clicked. The `$emit` call will be inline in the template, so you can omit `this`. Open src/components/Modal.vue, and update the `<button>` to emit a `close-modal` event on click as follows:

```
<button @click="$emit('close-modal')" />
```

The component doesn't receive any props now, so you can delete the entire `<script>` block in src/components/Modal.vue. The tests will pass when you run the command `npm run test:unit`. Good, you've converted the `Modal` component to emit a custom event.

How about one more test to check an emitted custom event? In the project, you'll see you have a `Form` component in src/component/Form.vue. This is going to be the form for users to submit their email. You'll add a test to check that the form emits a `form-submitted` event when the `<form>` element is submitted. Create a test file—src/components/__tests__/Form.spec.js—and add the code from the next listing.

Listing 5.5 Testing a Vue custom event is emitted

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

describe('Form.vue', () => {
  test('emits form-submitted when form is submitted', () => {
    const wrapper = shallowMount(Form)
    wrapper.find('button').trigger('submit')
    expect(wrapper.emitted('form-submitted')).toHaveLength(1)
  })
})
```

Dispatches a submit event on a button element

Asserts that the form-submitted custom event was emitted

To make the test pass, you'll add a submit event listener to the form element. In the event handler, you can emit `form-submitted`. Copy the code from the next listing into src/components/Form.vue.

Listing 5.6 Emitting a custom event on form submit

```
<template>
    <form @submit="onSubmit">
       <button />
    </form>
</template>

<script>

export default {
  methods: {
    onSubmit () {
      this.$emit('form-submitted')
    }
  }
}
</script>
```

Adds an onSubmit submit listener with v-bind

Emits a form-submitted event in the onSubmit method

Run the tests to watch them pass: `npm run test:unit`. The application is already listening for the `form-submitted` event in the `App` (src/App.vue) component. When the `Form` component emits a `form-submitted` event, the `App` `closeModal` method will fire and remove the `Modal` component from the page (you can check this by running the dev server: `npm run serve`).

You've written tests for the first part of the custom event system, where an emitted event is the output of a component. Now you need to learn how to write tests for components where an emitted event is the input for the component.

5.2.2 Testing components that listen to Vue custom events

If a Vue component emits an event and no component is listening, does it make a sound? I'm not sure, but you could write a test to see!

Like you saw earlier, components can listen to custom events emitted by their child components and run some code in response, as follows:

```
<modal @close-modal="closeModal" />
```

You just refactored the `Modal` component to emit a `close-modal` event. That changed the component's existing contract—to call an `onClose` prop when the modal is closed—which broke the app. You need to update the `App` component to listen to the `close-modal` event and hide the `Modal` when the `close-modal` event is emitted. Of course, you'll write a test to make sure that it does.

To test that a component responds correctly to an emitted event. you can emit the event from the `Modal` component by getting the `Modal` instance wrapper and accessing the `vm` property as follows:

```
wrapper.find(Modal).vm.$emit('close-modal')
```

In the `App` component test, you'll emit a `close-modal` event from the `Modal` and check that the `Modal` component is removed from the rendered output in response.

Create a test file for `App` in src/__tests__/App.spec.js. Add the code from the following listing to the src/__tests__/App.spec.js file.

Listing 5.7 Testing that the component responds to Vue custom event

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

describe('App.vue', () => {
  test('hides Modal when Modal emits close-modal', () => {
    const wrapper = shallowMount(App)
    wrapper.find(Modal).vm.$emit('close-modal')
    expect(wrapper.find(Modal).exists()).toBeFalsy()
  })
})
```

Shallow-mounts the App component

Emits a close-modal event from the Modal instance

Asserts that the Modal isn't rendered anymore using the toBeFalsy matcher

You can pass the test by updating the `App` component to listen to the `close-modal` event on the `Modal` component. Open src/App.vue, and replace the `Modal` start tag with the following code:

```
<modal
  v-if="displayModal"
  @close-modal="closeModal"
>
```

And just like that, the tests will pass: `npm run test:unit`.

You've seen how to write unit tests for components that use DOM events and Vue custom events. You'll write tests like these often; triggering an event is a common

input for components. The principle is simple: you need to trigger or emit an event in the test, and then assert that the tested component responds correctly.

Another common element that uses events is an input form. Input forms often use the value of form elements in an event handler to do something interesting with, like validating a password. Because input forms are so common, they deserve their own section.

5.3 Testing input forms

From contact forms, to sign-up forms, to login forms, input forms are everywhere! Input forms can contain a lot of logic to handle validation and perform actions with input values, and that logic needs to be tested.

In this section you'll learn how to test forms by writing tests for a `Form` component. The form will have an email input for users to enter their email address, two radio buttons for users to select whether they want to enter a competition, and a Subscribe button (figure 5.2).

Figure 5.2 The finished form

When a user submits the form, the component should send a POST request to an API with the email address the user entered and the value of the radio buttons. To keep things simple, it won't include any validation logic.

So the specs for the form are

- It should POST the value of email input on submit.
- It should POST the value of the *enter competition* radio buttons on submit.

The first test to write will check that you send a POST request with the email entered by the user. To do this, you need to learn how to test text-input values.

5.3.1 Testing text control inputs

Input elements are used to collect data entered by the user. Often, applications use this data to perform an action, like sending the data to an external API.

An interesting thing to note about input elements is that they have their own state. Different element types store their state in different properties. Text control inputs, like `text`, `email`, and `address`, store their state in a `value` property.

To test that event handlers use a `value` correctly, you need to be able to control the `value` property of an input in your tests. A lot of people get confused about how to set the value of an input form. A common misconception is that simulating a `keydown` event with a `key` property changes the element value. This is incorrect. To change the `value` property of an input in JavaScript, you need to set the `value` property on the element directly, as follows:

```
document.querySelector('input[type="text"]').value = 'some value'
```

When you write a test that uses an input `value`, you must set the `value` manually before triggering the test input, like so:

```
wrapper.find('input[type="text"]').value = 'Edd'
wrapper.find('input[type="text"]').trigger('change')
expect(wrapper.text()).toContain('Edd')
```

In Vue, it's common to use the `v-model` directive to create a two-way binding between an input `value` and component instance data. For a bound value, any changes a user makes to the form value will update the component instance data value, and any changes to the instance property value is applied to the input `value` property, as shown in the next listing.

NOTE If you aren't familiar with the `v-model` directive, you can read about it in the Vue docs—https://vuejs.org/v2/api/#v-model.

Listing 5.8 Using `v-model` to bind data

```
new Vue({
  el: '#app',
  data: {
    message: 'initial message'
  },
  template: '<input type="text" v-model="message" />',
  mounted() {
     setTimeout(() => this.message = '2 seconds', 2000)
  }
})
```

The initial value of message

Binds the input element to message data. The initial value of the input element will be the initial message.

Causes the input element value to be updated to 2 seconds, after 2,000 ms

Unfortunately, setting the input `value` property directly won't update the bound value. To update the `v-model` of a text input, you need to set the `value` on the element and then trigger a `change` event on the element to force the bound value to update. This is due to the implementation of `v-model` in Vue core and is liable to change in the future. Rather than relying on the internal implementation of `v-model`,

you can use the wrapper `setValue` method, which sets a value on an input and updates the bound data to use the new value, shown in the next listing.

Listing 5.9 Updating the value and `v-model` value of an input in a test

```
const wrapper = shallowMount(Form)
const input = wrapper.find('input[type="email"]')
input.setValue('email@gmail.com')
```

Gets a wrapper of an input element

Sets the value of the input element and updates the bound data

In the test that you're writing, you need to set the value of an input element with `setValue`, trigger a form submit, and check that the input element value is sent as part of a POST request. You know how to set the value and dispatch an event, but to assert that a POST request is sent, you need to decide how the POST request will be made by your component.

A common way to make HTTP requests is to use a library, like the axios library. With axios you can send a POST request using the axios `post` method, which takes a URL and an optional data object as arguments, as follows:

```
axios.post('https://google.com', { data: 'some data' })
```

> **NOTE** axios is a library for making HTTP requests, similar to the native `fetch` method. There's no special reason to use this library over another HTTP library; I'm just using it as an example.

The application you're working on is already set up to use axios. It uses the vue-axios library to add an `axios` Vue instance property (you can see this in src/main.js). That means you can call axios from a component as follows:

```
this.axios.post('https://google.com', { data: 'some data' })
```

Now that you know how you'll make a POST request, you can write an assertion that will check that you call the `axios.post` method. You do that by creating a mock `axios` object as an instance property and checking that it was called with the correct arguments using the Jest `toHaveBeenCalledWith` matcher.

The `toHaveBeenCalledWith` matcher asserts that a mock was called with the arguments that it's passed. In the following test, you're checking that the `axios.post` was called with the correct URL and an object containing the `email` property:

```
expect(axios.post).toHaveBeenCalledWith(url, {
  email: 'email@gmail.com'
})
```

The problem is, if you add extra properties to the axios data in later tests, the test will fail, because the argument objects do not equal each other. You can future-proof this test by using the Jest `expect.objectContaining` function. This helper, shown in

the next listing, is used to match *some* properties in the data object, rather than testing that an object matches exactly.

Listing 5.10 Using `objectContaining`

```
const data = expect.objectContaining({
  email: 'email@gmail.com'
})
expect(axios.post).toHaveBeenCalledWith(url, data)
```

Now the test will always pass as long as the `email` property is sent with the correct value.

It's time to add the test. It looks quite big, but if you break it down it's really a lot of setup before you trigger the `submit` event. Add the code from the next listing to src/components/__tests__/Form.spec.js.

Listing 5.11 Testing a mock was called with a `v-model` bound input form value

```
test('sends post request with email on submit', () => {
  const axios =
    post: jest.fn()
  }
  const wrapper = shallowMount(Form, {
    mocks: {
      axios
    }
   })
  const input = wrapper.find('input[type="email"]')
  input.setValue('email@gmail.com')
  wrapper.find('button').trigger('submit')
  const url = 'http://demo7437963.mockable.io/validate'
  const expectedData = expect.objectContaining({
    email: 'email@gmail.com'
  })
  expect(axios.post).toHaveBeenCalledWith(url, expectedData)
})
```

Creates a mock axios object with a post property

Shallow-mounts the form with the axios mock as an instance property

Sets the value of the input

Submits the form

Asserts that axios.post was called with the correct URL value as the first argument

Before you run the tests, you need to update the previous test. Currently the previous test will error because the form component will try to call `axios.post`, which is `undefined`—the leaky bucket problem in practice. The instance property dependency doesn't exist, so you need to mock the instance property to avoid errors in the test.

In src/components/__tests__/Form.spec.js, replace the code to create the wrapper in the *emits form-submitted when form is submitted* test with the following code snippets:

```
const wrapper = shallowMount(Form, {
  mocks: { axios: { post: jest.fn() } }
})
```

Now update the component with the code from the next listing.

Listing 5.12 `Form` component

```
<template>
  <form name="email-form" @submit="onSubmit">
    <input type="email" v-model="email" />
    <button type="submit">Submit</button>
  </form>
</template>

<script>
export default {
  data: () => ({
    email: null
  }),
  methods: {
    onSubmit (event) {
      this.axios.post('http://demo7437963.mockable.io/validate', {
        email: this.email
      })
      this.$emit('form-submitted')
    }
  }
}
</script>
```

Binds the input to the component email property with the v-model directive

Calls the axios.post method

You've just seen how to write a test for components that use an input element's value in the assertion. You can use `setValue` for all input elements that use a text control, like `text`, `textarea`, and `email`. But you need to use a different method for other input types, like radio buttons.

5.3.2 Testing radio buttons

Radio buttons are buttons you can select. You can select only one button from a radio group at a time. Testing radio buttons is slightly different from testing a text-input element.

Your website is having a competition! Everybody who sees the sign-up modal will have the chance to enter the competition. When the form is submitted, you'll send the user's selection (using the radio button's value) in the POST request to the API. I sense another test to write!

Testing radio buttons is similar to testing input forms. Instead of the internal state being `value`, the internal state of radio buttons is `checked`. To change the selected radio button, you need to set the `checked` property of a radio button input directly, as shown in listing 5.13.

> **NOTE** The `checked` property is like the `value` property. It's the state of a radio button that's changed by a user interacting with the radio input.

Listing 5.13 Updating the value and `v-model` value of a radio button input in a test

```
const wrapper = shallowMount(Form)
const radioInput = wrapper.find('input[type="radio"]')
radioInput.element.checked = true
```

Gets a wrapper of a radioInput element

Sets the checked property of the radioInput element directly

Setting the `checked` value directly suffers the same problem as setting a text-control value directly: the `v-model` isn't updated. Instead, you should use the `setChecked` method as follows:

```
wrapper.find('input[type="radio"]').setChecked()
```

You should really write two tests. The first test will check that the form sends `enterCompetition` as `true` by default, because the Yes check box is selected by default. For brevity, I won't show you how to write that test. The test that you'll write instead will select the No radio button, submit the form, and assert that `enterCompetition` is false.

This is a big old test, but once again, it's mainly setup. You can see the same technique of putting an input element into the correct state using `setSelected` before dispatching an event to trigger the submit event handler. Add the code from the next listing to the `describe` block in src/components/__tests__/Form.spec.js.

Listing 5.14 Testing a component is called with the correct values

```
test('sends post request with enterCompetition checkbox value on submit', ()
     => {
  const axios = {
    post: jest.fn()
  }
  const wrapper = shallowMount(Form, {
    mocks: {
      axios
    }
  })
  const url = 'http://demo7437963.mockable.io/validate'

  wrapper.find('input[value="no"]').setChecked()
  wrapper.find('button').trigger('submit')

  expect(axios.post).toHaveBeenCalledWith(url, expect.objectContaining({ //
    enterCompetition: false
  }))
})
```

Shallow-mounts the Form component with an axios mock object

Sets the No radio button as checked

Submits the form

Asserts that axios.post was called with the correct enterCompetition value

To make the tests pass, you need to add the radio inputs and update the `onSubmit` method to add the `enterCompetition` value to the data object sent with `axios.post`.

Add the following radio inputs to the `<template>` block in src/components/Form .vue:

```
<input
  v-model="enterCompetition"
  value="yes"
  type="radio"
  name="enterCompetition"
/>
<input
  v-model="enterCompetition"
  value="no"
  type="radio"
  name="enterCompetition"
/>
Add enterCompetition to the default object:
data: () => ({
  email: null,
  enterCompetition: 'yes'
}),
```

Finally, update the axios call to send an `enterCompetition` property. The test expects a Boolean value, but the values of the radio buttons are strings, so you can use the strict equals operator to set `enterCompetition` as a Boolean value as follows:

```
this.axios.post('http://demo7437963.mockable.io/validate', {
  email: this.email,
  enterCompetition: this.enterCompetition === 'yes'
})
```

Run the unit tests to watch them pass: `npm run test:unit`. You've added all the tests that you can to test the form functionality.

Ideally you would add one more test to check that submitting the form doesn't cause a reload, but using jsdom it's not possible to write. Every parent dreads the inevitable birds-and-the-bees conversation; I always dread the inevitable limitations-of-jsdom conversation.

5.4 Understanding the limitations of jsdom

To run Vue unit tests in Node, you need to use jsdom to simulate a DOM environment. Most of the time this works great, but sometimes you will run into issues with unimplemented features.

In jsdom, the two large unimplemented parts of the web platform are

- Layout
- Navigation

Layout is about calculating element positions. DOM methods like `Element.getBoundingClientRects` won't behave as expected. You don't encounter any problems with this in this book, but you can run into it if you're using the position of elements to calculate style in your components.

The other unimplemented part is navigation. jsdom doesn't have the concept of pages, so you can't make requests and navigate to other pages. This means `submit` events don't behave like they do in a browser. In a browser, by default a `submit` event makes a GET request, which causes the page to reload. This behavior is almost never desired, so you need to write code to prevent the event from making a GET request to reload the page.

Ideally, you would write a unit test to check that you prevent a page reload. With jsdom, you can't do that without extreme mocking, which isn't worth the time investment.

So instead of writing a unit test, you would need to write an end-to-end test to check that a form submission doesn't reload the page. You'll learn how to write end-to-end tests in chapter 14. For now, you'll add the code without a test.

To stop the page from reloading, you can add an event modifier to the `v-bind` directive. Open src/components/Form.vue and add a `.prevent` event modifier to the submit `v-bind` as follows:

```
<form name="email-form" @submit.prevent="onSubmit">
```

The modifier calls `event.preventDefault`, which will stop the page from reloading on submit.

As I said earlier, the two parts of jsdom that aren't implemented are navigation and layout. It's important to understand these limitations, so you can guard against them. When you encounter the limitations, instead of mocking, you should supplement your unit tests with end-to-end tests that check functionality that relies on unimplemented jsdom features.

Now that you're preventing default, you have a fully functioning form. You can open the dev server and have a look: `npm run serve`. Obviously this form is nowhere near ready for public consumption. It has no styling and is incredibly ugly. The point is, you now have a suite of unit tests that check the core functionality and can freely add style without being slowed down by unit tests.

> **NOTE** To see what the finished application looks like, you can go to http://mng.bz/oN4Z.

In the next chapter, you're going to learn about Vuex. The chapter is intended for readers who don't have experience with Vuex; if you have used it before, then you can skip ahead to chapter 7 to learn how to test Vuex.

Summary

- You can trigger native DOM events with the wrapper `trigger` method.
- You can test that a component responds to emitted events by calling `$emit` on a child component instance.
- You can test that a component emitted a Vue custom event with the wrapper `emitted` method.
- jsdom doesn't implement navigation or layout.

Exercises

1. How do you simulate a native DOM event in tests?
2. How would you test that a parent component responds to a child component emitting an event?

JAVASCRIPT/WEB DEVELOPMENT

Testing Vue.js Applications

Edd Yerburgh

Web developers who use the Vue framework love its reliability, speed, small footprint, and versatility. Vue's component-based approach and use of DOM methods require you to adapt your app-testing practices. Learning Vue-specific testing tools and strategies will ensure your apps run like they should.

With **Testing Vue.js Applications**, you'll discover effective testing methods for Vue applications. You'll enjoy author Edd Yerburgh's engaging style and fun real-world examples as you learn to use the Jest framework to run tests for a Hacker News application built with Vue, Vuex, and Vue Router. This comprehensive guide teaches the best testing practices in Vue along with an evergreen methodology that applies to any web dev process.

What's Inside

- Unit tests, snapshot tests, and end-to-end tests
- Writing unit tests for Vue components
- Writing tests for Vue mixins, Vuex, and Vue Router
- Advanced testing techniques, like mocking

Written for Vue developers at any level.

Edd Yerburgh is an experienced JavaScript developer, a Vue core contributor, and the main author of the official Vue test library.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/testing-vue-js-applications

"A comprehensive guide to Vue.js testing by the author of the official testing utility."
—Evan You, creator of Vue.js

"Totally on point—your one-stop shop for testing Vue.js applications. Best of breed for the latest tech."
—Clive Harber, Distorted Thinking

"Edd's book on testing Vue is the richest guide I have found so far. I will be pointing others to it as the way to learn testing that works for Vue."
—John Farrar, Active On-Demand

"Straightforward and easy to read. The book instructs, demonstrates, and provides the techniques necessary to successfully test all levels of a Vue.js application."
—Jim Schmehil
National Heritage Academies

ISBN-13: 978-1-61729-524-9
ISBN-10: 1-61729-524-8
54499
9 781617 295249

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]