SAMPLE CHAPTER

TESTING Vue.js Applications

Edd Yerburgh

MANNING

Testing Vue.js Applications

by Edd Yerburgh

Sample Chapter 3

brief contents

3 Testing rendered component output

This chapter covers

- Testing component output
- Writing a static Hacker News feed

If a tree falls in a forest and no one is around to hear it, does it make a sound? More importantly, if a component is mounted and no one writes a test for it, did it generate output?

Testing is about input and output. In a test you supply an input, receive an output, and assert that the output is correct. The most common output of components is the rendered output—the stuff that render functions generate. That's what this chapter is about: testing rendered component output.

To learn how to test component output, you'll write a static Hacker News feed. The news feed is simple, but it gives you the opportunity to test different forms of output and get acquainted with the Vue Test Utils API.

This book is about testing from start to finish, and part of the testing process is converting murky requirements into specifications. The first section of this chapter is about creating specifications from requirements.

After you have the specifications for the static news feed, you'll write tests for the specs. By doing that, you'll learn how to use different Vue Test Utils methods to test the rendered output of Vue components in unit tests. After you've written the news feed, you'll create a progress bar component, a task through which you'll learn how to test styles and classes.

The first thing to do is create the specifications for the static news feed.

3.1 Creating test specifications

A construction company doesn't begin to construct a skyscraper until it has the blueprints. As a programmer, you shouldn't write tests until you've got the specifications.

Deciding what tests you should write can be difficult at first, but it's a vital skill to learn. To help, you can use the following process for Vue applications:

- Agree on requirements.
- Answer questions on details to get high-level specifications and design.
- Break up design into components.
- Write component-level specifications.

You already know the requirement: you should *write a static Hacker News feed*. There's a lot of room for interpretation there, so you need to hammer down some of the details.

3.1.1 High-level specifications

Requirements are fuzzy descriptions of an application from a user perspective. You need to take requirements and interrogate them until you have technical specifications for how the product should work.

The requirement you've been given is to create a static Hacker News feed. The first question you should have is: what will it look like? Words can mean different things to different people, but it's difficult to misinterpret an image. You can see a design for the Hacker News app in figure 3.1.

The design answers a lot of questions. I have only two more questions:

- How many items should be displayed?
- How do you get the data?

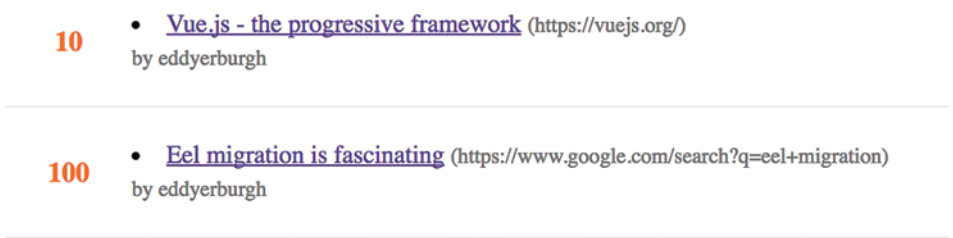

Figure 3.1 Two items from the finished feed

The first question is easy. You're going to display every item that's returned by the data. The next question to answer is *how do you get the data*. For now, you're going to fetch the data before the app is mounted, and then set it as a property on the `window` object. The `window` object is a global variable in a browser environment, so components in the app will be able to access the items.

> **NOTE** If you've been developing for a while, your alarm bell is probably sounding. Adding properties to the `window` object is bad practice. Don't worry—you'll refactor the data fetching in chapter 4.

The code is already written in the src/main.js entry file. It fetches the top news items from the Hacker News API and set them as `items` properties on the `window` object before creating the Vue instance. You can see the data being fetched and added to `window` in listing 3.1.

> **NOTE** This chapter follows on from the app you created in chapter 2. If you don't have the app, you can check out the chapter-3 Git branch by following the instructions in appendix A.

Listing 3.1 Instantiating Vue after fetching data

```
fetchListData('top')
  .then((items) => {
    window.items = items
    new Vue({
      el: '#app',
      render: h => h(App)
    })
  })
```

callback function with the items returned from fetchListData

Sets window.items to items returned by fetchListData

Mounts the application when the data is added to window.items

> **NOTE** The code in main.js uses a promise to make sure the data is loaded before mounting the app. If you aren't familiar with promises, read about them on the MDN page at https://mzl.la/2j7Nq1C.

The high-level specifications are

- Create a feed using the design in figure 3.1.
- Use the data in `window.items` to render the feed.
- Display all the items in the data.

Congratulations, you've turned a murky requirement into clear high-level specifications for how the feed will work. Now you need to think about component design.

3.1.2 Creating component-level specifications

When you have high-level specifications for an application, you need to think about how you're going to implement them. With Vue applications, that involves deciding how to represent UI elements as components.

At a high level, a feed is simply a list of items. You could represent the Hacker News feed as an `ItemList` component that renders an `Item` component for each item in the data (figure 3.2).

The feed will be made from an `ItemList` component and an `Item` component. Now you need to think about what each of these components should do.

The `ItemList` component is responsible for rendering `Item` components with the correct data. You could write the following specs for `ItemList`:

- Render an `Item` component for each item in `window.items`.
- Pass correct data to each `Item`.

The component design for a Hacker News feed

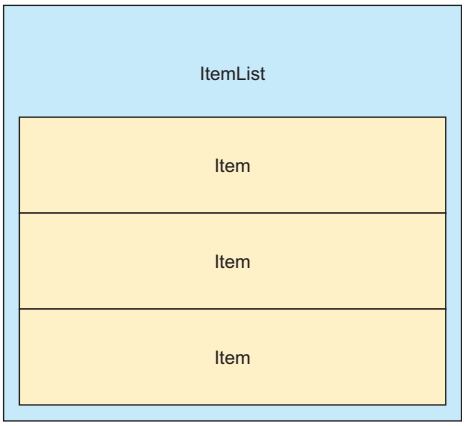

Figure 3.2 `ItemList` containing `Item` components

The `Item` component will be responsible for rendering the correct data. In chapter 1, I spoke about the concept of a component contract. A component contract is like a component API. The API for the `Item` component is that it receives an item prop and uses it to render the data. You could write this as specs as follows:

- Render a URL, an author, and a score using the data it receives as an item prop.
- Render a link to the `item.url` with `item.title` as the text.

Now that you've got your specifications, it's time to write tests that implement them. You'll start by writing tests for the `Item` component. To do that, you need to learn how to test rendered text.

3.2 Testing rendered text

Often you need to test that a component renders some text. I find myself doing it all the time. In this section, you'll learn how to test that components render text and how to test that a specific DOM element renders the correct text.

To learn how to test text, you'll write tests for the `Item` component. The `Item` component has the following specs:

- Render a URL, an author, and a score using the data it receives as an `item` prop.
- Render a link to the `item.url` with `item.title` as the text.

The first spec doesn't specify an element that the URL, author, or score should be rendered in. Therefore, the test should just check that they are rendered *somewhere* in the component output. The second test specifies that the test should be rendered in a link (also known as an `<a>` tag). That test should check that the component renders the text in an `<a>` tag.

To write these tests, you need to provide the `Item` component with props data when you mount it. You can provide props data to components using Vue Test Utils.

3.2.1 Passing props to a component

When you write a unit test for a component, you need provide the component with the input it will receive in production. If the component receives a prop in production, you need to provide the prop to the component when you mount it in the test.

You can pass props to components in an options object when you mount a component with Vue Test Utils as shown next.

Listing 3.2 Passing props to a component with mounting options

```
const wrapper = shallowMount(Item, {
  propsData: {
    item: {}
  }
}
```

As well as `propsData`, the options object accepts any options that you normally pass when you create a Vue instance. You'll use the options object a lot throughout this book.

Now that you know how to pass props to a component, you can write a test to check that the `Item` component uses this data to render text.

3.2.2 Testing the text content of a component

Sometimes you need to test that a component renders some text. It doesn't matter what element renders the text, just that the text is rendered *somewhere* in the rendered output of a component.

You can test that a component contains text using the Vue Test Utils `text` method:

```
expect(wrapper.text()).toBe('Hello, World!')
```

There's a problem, though. Calling `text` on a component wrapper returns *all* the text rendered by the component. A `toBe` matcher would check that all the text in a component strictly equals the expected value. If you decided to add extra text to the component, the test would break.

A principle of tests is that *a test should not break if the functionality it tests does not change*. It's difficult to write tests that follow this principle, but you should always aim to make your tests future proof. Using a `toBe` matcher to check all rendered text of a component violates this principle.

You can solve this problem by using the `toContain` matcher. The `toContain` matcher checks that a value is contained *somewhere* in the string it is checking. It's a bit like the `string.prototype.includes` method. You could write a test that checks that a component renders some text, such as the following:

```
expect(wrapper.text()).toContain('Hello, World!')
```

Now the assertion will fail only if "Hello, World" isn't rendered at all. You could add as much extra text to the component as you wanted, and the test would still pass. You'll use `toContain` a lot in this book to check that rendered component output contains a value.

The first spec for the `Item` component checks that it *renders a URL, an author, and a score using the data it receives as a prop*. These are each independent features of the component, so you should split them into three unit tests.

Each test will mount an `Item` component with an item object passed down as an `item` prop and then assert that the rendered output contains the correct text. You'll use the `text` method and the `toContain` matcher to write the test.

Add the code shown next to src/components/__tests__/Item.spec.js, replacing the existing code.

Listing 3.3 Passing props to components in a test

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

describe('Item.vue', () => {
  test('renders item.url', () => {
    const item = {
      url: 10
    }
    const wrapper = shallowMount(Item, {
      propsData: { item }
    })
    expect(wrapper.text()).toContain(item.url)
  })
})
```

Passes the item object as props to Item

Uses toContain to assert the item.url exists in rendered component text

Watch the test fail by running the following test script: `npm run test:unit`. It should fail with an assertion error telling you that the string did not contain the correct text.

You can make the test pass by updating the component to receive an `item` prop and render the score. Replace the code in src/components/Item.vue with the following code:

```
<template>
  <li>
      {{ item.url }}
  </li>
</template>

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

You can check the test passes by running `npm run test:unit`. The other two tests for this component are very similar to the tests you just wrote, so I won't show you how

to write them here. In the exercises at the end of this chapter, you can implement them yourself, or you can check out the chapter-4 Git branch to see the finished tests.

For the next spec, you need to check that the `Item` component renders a link to the `item.url` with `item.title` as the text. To test that `Item` renders an `<a>` element with the correct text, you need to access the `<a>` element in a components rendered output.

3.2.3 Using find

With Vue applications, it's components all the way down. With Vue Test Utils, it's wrappers all the way down. You interact with the rendered output of components through the wrapper interface.

You can get wrappers for each node in the rendered output using the `find` method. `find` searches the rendered output for the first node that matches a *selector* and returns a wrapper containing the matching node (figure 3.3).

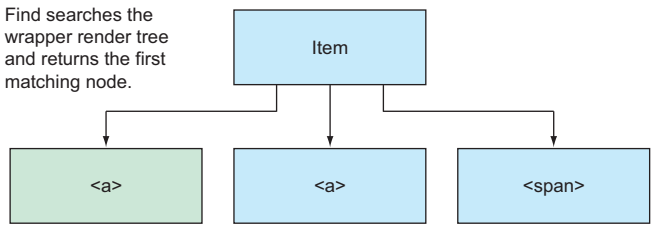

Figure 3.3 `find` searching the render tree

For example, you could get the text of an `<a>` element by using an `<a>` selector with `find`, and calling the text on the returned wrapper as follows:

```
wrapper.find('a').text()
```

You'll use this to test that your component renders an `<a>` element with the correct text content.

3.2.4 Testing the text content of an element

Sometimes in tests a component must render text somewhere in the component. Other times you need to be more specific and test that a component renders text in a particular element.

The `Item` component should *render a link to the* `item.url` *with* `item.title` *as the text*. This test needs to be more specific than checking that the text is rendered *somewhere* in the component. This text *must* be rendered in an `<a>` element.

> **NOTE** You'll check that the `<a>` element has the correct `href` in the next section—testing DOM attributes.

The test will use `find` to get a wrapper containing an `<a>` element and then call the `text` method to retrieve the text content of the element. Add the following test to the `describe` block in src/components/__tests__/Item.spec.js.

Listing 3.4 Testing component text

```
test('renders a link to the item.url with item.title as text',
  () => {  const item = {
    title: 'some title'
  }
  const wrapper = shallowMount(Item, {
    propsData: { item }
  })
  expect(wrapper.find('a').text()).toBe(item.title)
})
```

Creates a mock item to pass in as prop data

Passes prop data

Finds an <a> element and checks the text rendered is item.title

Run the unit test script to make sure the test fails for the right reason: `npm run test:unit`. You'll get a Vue Test Utils error that tells you an `<a>` element couldn't be found.

A Vue Test Utils error means the test is *almost* failing for the right reason, but not quite. You can check that the test fails for the right reason after add the `<a>` tag and make the test pass.

To make the test pass, you need to render `item.title` in the `<a>` tag. Open src/components/Item.vue, and replace the `<template>` block with the following code:

```
<template>
  <li>
      <a>{{ item.title }}</a>
      {{ item.url }}
  </li>
</template>
```

Run the unit script again: `npm run test:unit`. It passes, but you never saw it fail for the right reason. If you want to be extra careful (which I always am), you can remove the text from the `<a>` tag to see an assertion error. Be sure to add the text again once you've verified it fails for the correct reason.

In line with TDD, you added the bare minimum source code to pass this test. You're just rendering a title inside an `<a>` element, which is not a full functioning link. The next step is to make it a link by adding a test to check that the `<a>` element has an `href` value!

3.3 Testing DOM attributes

I don't always write components that render DOM attributes as part of the component contract, but when I do I write a test for them. Luckily, it's easy to test DOM attributes with Vue Test Utils.

The specification you're working on is that the `Item` component *renders a link to the* `item.url` *with* `item.title` *as the text*. You've rendered it with the title text, so now

you need to give it an `href` property using the `item.url` value. To do that, you need a way to access the `href` attribute in the test.

A Vue Test Utils wrapper has an `attributes` method that returns an object of the component attributes. You can use the object to test the value of an attribute as follows:

```
expect(wrapper.attributes().href).toBe('http://google.com')
```

You'll use `attributes` in your test to check that the `<a>` element has the correct `href` value. You can `find` a wrapper containing the `<a>` element and then call the `attributes` method to access the `href` value. Replace the *renders a link to the* `item.url` *with* `item.title` *as text* test with the code from the next sample into src/components/__tests__/Item.spec.js.

Listing 3.5 Testing DOM attributes

```
test('renders a link to the item.url with item.title as text', () => {
  const item = {
    url: 'http://some-url.com',
    title: 'some-title'
  }
  const wrapper = shallowMount(Item, {
    propsData: { item }
  })
  const a = wrapper.find('a')
  expect(a.text()).toBe(item.title)
  expect(a.attributes().href === item.url).toBe(true)
})
```

Asserts that an <a> element has an href attribute with value of item.url

Run the unit test again: `npm run test:unit`. There isn't an `href` attribute on the `<a>` element, so the test fails.

Take look at the error message: "expected value to be true, received false." This message isn't very useful. Is the test failing because the `href` value is incorrect, or is it failing because the `<a>` element doesn't have an `href` at all?

This kind of assertion error is caused by a *Boolean assertion*. You should avoid Boolean assertions like I avoided long-distance running at school.

3.3.1 Avoiding Boolean assertions

Boolean assertions are assertions that compare Boolean values. When they fail, the assertion error isn't clear about why the test failed: "expected false to equal true."

With a Boolean assertion, you're left wondering: *Did the element contain an incorrect attribute value? Was the attribute even rendered?* The Boolean assertion error refuses to tell you.

The alternative to Boolean assertions are expressive *value assertions*. Like the name suggests, a value assertion is an assertion that compares one value against another value.

When a test fails with a value assertion, you get a descriptive error message—"expected 'some value' to equal 'somevalue'." Aha! You look at the test code and see that a component was expecting a prop of `some value`, but someone accidentally

deleted a space in the code. When a unit test throws a value assertion error, you get a useful clue to start you on your debugging trail.

You can rewrite the test from earlier to use a value assertion. In src/components/__tests__/Item.spec.js, replace the Boolean assertion in the *renders a link to the* `item.url` *with* `item.title` *as text* test with the with the following line:

```
expect(a.attributes().href).toBe(item.url)
```

Run the tests again: `npm run test:unit`. The assertion error is much clearer now. You can see that the test is failing because the `<a>` element `href` attribute is `undefined`.

Armed with useful information, you can make the test pass. In src/components/Item.vue, add an `href` prop that takes `item.url`. You need to *bind* the `href` to the <a> tag using with the v-bind directive colon (`:`) shorthand, because you're using a dynamic value. Edit the <a> tag to look like this:

```
<a :href="item.url">{{ item.title }}</a>
```

> **NOTE** You need to use the v-bind directive to pass dynamic data as an HTML attribute. To learn more about the v-bind directive, see the Vue documentation at https://vuejs.org/v2/api/#v-bind.

Now run the test script: `npm run test:unit`. If you added the `href` correctly, the test will pass. Great, you've got the functionality for the `Item` component written and tested—the component fulfills its contract.

The next component to test is the `ItemList` component. The first test that you write will check that it renders an `Item` component for each item in the `window.items` array. To write this test, you need to learn how to test how many components are rendered by a parent component.

3.4 Testing how many components are rendered

In this book, you've been using the `shallowMount` method to mount components. `shallowMount` doesn't render child components, so the unit tests you write are testing only the root-level component. Doing so makes unit tests more focused and easier to understand. But what if you need to check that a root component renders child components?

This is a common scenario. I find myself writing lots of tests that check that components are rendered. Previously, you used the `find` method to access an element, but `find` returns the first matching node, so it's no good if you want to check the total number of nodes rendered. To do that, you can use the `findAll` method.

3.4.1 Using findAll

The `findAll` method is to `document.querySelectorAll` as `find` is to `document.querySelector`. `findAll` searches the rendered output for nodes that match a selector and returns an arraylike object containing wrappers of matching nodes.

The arraylike object is known as a *wrapper array*. Like a JavaScript array, a wrapper array has a `length` property. You can use the wrapper array `length` property to check how many elements exist in the component tree, as shown next.

Listing 3.6 Using the wrapper array `length` property

```
const wrapper = mount(ItemList)
wrapper.findAll('div').length
```

Length equals the number of elements rendered by TestComponent.

`findAll` uses a selector to match nodes in the rendered output. If you use a Vue component as a selector, `findAll` matches instances of the component, as shown in the next code sample.

Listing 3.7 Using a component as a selector

```
import Item from '../src/Item.vue'

const wrapper = mount(ItemList)
wrapper.findAll(Item).length
```

Length equals the number of Item instances.

You'll use this technique to write your test. Remember the specification: ItemList *should render an* `Item` *component for each item in* `window.items`.

In the test, you'll set the `window.items` property to be an array with some objects in it. Then you'll mount the component and check that ItemList renders the same number of `Item` components as the objects in the items array.

Remember that a good unit test has an expressive assertion error. Assertion errors are more expressive when you use matchers suitable for the assertion. When you test that an array, or an arraylike object, has a `length` property, you can use the `toHaveLength` matcher.

Create a new test file src/views/__tests__/ItemList.spec.js. Copy the following code into src/views/__tests__/ItemList.spec.js.

Listing 3.8 Testing child components

```
import { shallowMount } from '@vue/test-utils'
import ItemList from '../ItemList.vue'
import Item from '../../components/Item.vue'

describe('ItemList.vue', () => {
  test('renders an Item for each item in window.items', () => {
    window.items = [{},{},{}]
    const wrapper = shallowMount(ItemList)
    expect(wrapper.findAll(Item))
      .toHaveLength(window.items.length)
  })
})
```

Sets items data for the component to use

Mounts ItemList

Uses a WrapperArray length property to check that an Item is rendered for each item in window.items

Now run the unit tests with the command `npm run test:unit`. This will give you a useful assertion error—"Expected value to equal: 3 Received: 1."

> **The principle of the smallest possible mocks**
>
> Often, in tests, you need to pass mocked data to a component or function. In production, this data might be large objects with tons of properties.
>
> Large objects makes tests more complicated to read. You should always pass the least amount of data that's required for the test to work.

To make the test pass, add the next code to src/views/ItemList.vue.

Listing 3.9 Using `v-for` to render items based on an array

```
<template>
    <div class="item-list">
        <item v-for="item in displayItems" :key="item.id"></item>
  </div>
</template>

<script>
import Item from '../components/Item.vue'

export default {
  components: {
    Item
  },
  data () {
    return {
      displayItems: window.items
    }
  }
}
</script>
```

Renders an Item for each object in the displayItems array

Makes the items from the data/items array available to the component as displayItems

Run the tests again: `npm run test:unit`. The test should pass. Great—that's the first `ItemList` spec done.

Now you need to write a test for the second spec: *each* `Item` *should receive the correct data to render*. You need to test that `ItemList` passes the correct data to each `Item`. To do that, you need to learn how to test component props.

3.5 Testing props

For components that take props, it's vital that they receive the correct prop to behave correctly. You can write tests that check that a component instance has received the correct props using Vue Test Utils.

The second spec for `ItemList` is that *each* `Item` *should receive the correct data to render*. It's important that each `Item` component receives the correct data as a prop. Part

of the `Item` component contract is that it receives the correct data. You can't expect an employee to work if you don't supply a salary, and you can't expect an `Item` to render correctly if you don't supply an `item` prop.

To test that a component receives a prop, you can use the Vue Test Utils `props` method.

3.5.1 Using the Vue Test Utils props method

`props` is a Vue Test Utils wrapper method. It returns an object containing the props of a wrappr component instance and their values, shown in the following listing.

Listing 3.10 Testing props

```
const wrapper = shallowMount(TestComponent)
expect(wrapper.find(ChildComponent).props()
➥ .propA).toBe('example prop')
```

Calls the props method to get an object of ChildComponent props

You can use the `props` method to assert that each `Item` component receives the correct `item` prop. Instead of adding a new test, you should update the previous test description and assertion.

In src/views/__tests__/ItemList.spec.js, replace the *renders an* `Item` *for each item in* `window.items` with the following code.

Listing 3.11 Testing props using the `props` method

```
test('renders an Item with data for each item in window.items', () => {
  window.items = [{}, {}, {}]
  const wrapper = shallowMount(ItemList)
  const items = wrapper.findAll(Item)
  expect(items).toHaveLength(window.items.length)
  items.wrappers.forEach((wrapper, i) => {
    expect(wrapper.props().item).toBe(window.items[i])
  })
})
```

Creates a WrapperArray of Item components

Loops through each Item

Asserts that the Item at index i has a prop item with a value matching the item at index i

If you run the tests with `npm run test:unit` you'll see that `item` is `undefined`. This is expected, because you aren't passing any data to the `Item` components yet. To make the test pass, you need to pass an `item` prop to each `Item` component that you render.

Open src/views/ItemList.vue, and replace the `<template>` block with the next code.

Listing 3.12 Passing props to a child component

```
<template>
  <div class="item-list">
  <item
    v-for="item in displayItems"
    :key="item.id"
```

Loops through each item object in the displayItems array

Gives each Item component a unique key; notice you have access to an item object in the loop

```
    :item="item"                          Passes the item as an item
  />                                      prop to the Item component
  </div>
</template>
```

Now run the test script again: `npm run test:unit`. You'll see the code pass. Congratulations—you've written tests and code for the news feed.

Before you move on to the next component, I want to take a moment to talk about a common gotcha when testing props. If you're not careful, this one will trip you up.

3.5.2 Avoiding gotchas when testing props

One big gotcha can catch you out when you test component props. If a component does not declare that it will receive a prop, the prop will not be picked up and added to the Vue instance.

For a component to receive a prop, *the component must declare that it will receive the prop*. This can catch you off guard if you're following TDD and you write tests for the parent before you finish the child component.

Listing 3.13 Declaring a prop in a single-file component

```
<script>
export default {
  props: ['my-prop']                      Declaring that the component
}                                         receives a prop named my-prop
</script>
```

NOTE You can read in detail how to declare received component props in the Vue docs at http://mng.bz/ZZwP.

For demonstration purposes, open src/components/Item.vue, and remove the `props` property. If you run the test again, the `ItemList` test will fail. Make sure to add `props` back in to pass the test before moving on.

Now you've completed the specs for the `Item` and `ItemList` components. But if you run the dev server (`npm run serve`), you'll see that the components are far from finished. It would be embarrassing to show this to your boss and say, "There you go, I've finished all the specs!" The application is missing the style and pizzazz that it deserves.

To style the components, you need to add some static HTML and CSS. The thing is, HTML and CSS don't work well with unit tests. Adding presentational HTML is an iterative process, and unit tests can really slow this process down. Another key part of styling is manual testing. Unless you're a CSS superstar, you need to manually test that the HTML and CSS style your application correctly. Unit tests get in the way of this process.

Later in this book, you'll learn how to capture manual testing for static HTML with snapshot tests. For now, you can add style to your components without any tests. The big takeaway here is that *you don't need to unit test presentational HTML*.

You've finished writing the news feed. There are no more unit tests to write. Before you move on to the next chapter, you'll learn how to test classes and styles by writing tests for a progress bar component.

3.6 Testing classes

One of the questions that developers new to frontend testing have is whether they should test element classes. Frustratingly, the answer is that it depends. Let's look at an example where you should test element classes.

Your Hacker News application is going to render a progress bar. The `ProgressBar` component will indicate that a page is loading. You can see an example in figure 3.4.

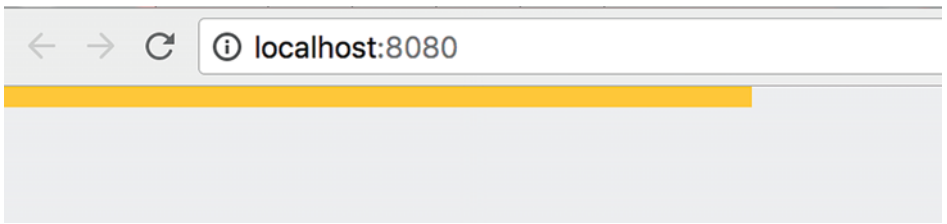

Figure 3.4 The progress bar in progress

I've done the hard work of getting the following specs and requirements for you:

- The ProgressBar should be hidden by default.
- The ProgressBar should initialize with 0% width.

The root element in the `ProgressBar` component should be hidden by default. You can hide the `ProgressBar` with a `hidden` class, which will apply a CSS rule to hide the element.

So to check that the element is hidden, you will test that the component root element has a class of `show`. You can do that with the wrapper `classes` method.

3.6.1 Using the classes method

The Vue Test Utils `classes` wrapper method returns an array of classes on the wrapper root element. You can assert against the array to see whether an element has a class. The test will shallow-mount the ProgressBar and check that the array returned by the `classes` method contains `hidden`.

The `classes` method returns an array. Earlier, you used the `toContain` matcher to check that one string contained another. That `toContain` matcher is more versatile than Leonardo DiCaprio. Not only can it compare string values, it can also compare values in an array. Add the code from the next listing to src/components/__tests__/ProgressBar.spec.js.

Listing 3.14 Testing a class with the `classes` method

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

describe('ProgressBar.vue', () => {
  test('is hidden on initial render', () => {
    const wrapper = shallowMount(ProgressBar)
    expect(wrapper.classes()).toContain('hidden')
  })
})
```

Checks that the root element has a class name including hidden

Before you run the test for the `ProgressBar` component, you should add the component file and a simple `<template>` block. That way, you can mount the component in the test and get an assertion error. If a file doesn't exist, then the test will fail when it tries to import the file. You want the test to fail with an assertion error, so you should always create a minimal file before running a unit test.

Create a file in src/components/ProgressBar.vue, and add the following empty `<template>` block:

```
<template></template>
```

Now run the tests: `npm run test:unit`. The test will fail with a friendly assertion error. To make it pass, you need to update the component template. Add the following code to src/components/ProgressBar.vue:

```
<template> <div class="hidden" /> </template>
```

Check that the test passes before moving on: `npm run test:unit`. You have only one spec for the `ProgressBar` left—the `ProgressBar` *should initialize with 0% width*. The width will be added as an inline style, so you need to learn how to test inline styles to write this spec.

3.7 Testing style

Sometimes you don't need to test style—not Italian suit kind of style, but inline CSS styles. Normally testing style isn't valuable, but you *should* write tests for some cases of inline styles: for example, if you add an inline style dynamically.

The `ProgressBar` component you're going to write needs to have 0% width when it initializes; it will increase over time, which makes it appear to load. In the next chapter, you'll add methods to control the component, like `start` and `finish`. For now, you'll just test that it's initialized with a width style of 0%.

To test an inline style, you need to access the wrapper element directly and get the style value.

3.7.1 Accessing a wrapper element

The DOM has a notoriously ugly API. Often you will use libraries to abstract over it and make code more expressive—that's one of the benefits of Vue Test Utils—but sometimes you should use the DOM API directly.

To use the DOM API with Vue Test Utils, you need to access a DOM node. Every wrapper contains an `element` property, which is a reference to the root DOM node that the wrapper contains. You can use the `element` property to access the element's inline styles as follows:

```
wrapper.element.style.color
```

The test you write will check that the wrapper root element has a `width` style value of 0%. To test that, you'll shallow-mount the component and access the element's `style` property. Open src/components/__tests__/ProgressBar.spec.js, and add the test from the following listing to the `describe` block.

Listing 3.15 Testing style by accessing wrapper element

```
test('initializes with 0% width', () => {
  const wrapper = shallowMount(ProgressBar)
  expect(wrapper.element.style.width).toBe('0%')
})
```

Checks the wrapper element's inline width property

Now run the test script – `npm run test:unit`. It will fail because the root element has no `width` value. To make the test pass, copy the following code into the `<template>` block:

```
<template>
  <div
    class="hidden"
    :style="{
    'width': '0%'
  }" />
</template>
```

If you run the tests again, they'll pass. Great—those are all the tests that you'll write in this chapter. Now is a good time to talk about styling an application.

3.7.2 Adding style to an application

Style is an important part of frontend development. You could write the greatest HTML in the world, but without some CSS your application is going to look bad.

The process of adding style involves manual testing. After you've written CSS, you need to check in a browser that the styles have been applied correctly. If you're a great developer, then you'll probably test it on multiple devices and browsers.

Because styling applications involves manual testing, unit tests that check only static presentational elements are not valuable. One benefit of unit tests is that they save you time, because you can run them without checking the code manually. Unit tests for static elements often take longer to write than the time they save. Save yourself time, and don't write unit tests when you're styling!

This book is about automated testing, so styling that should be manually tested won't be included here. That said, style is important for the Hacker News application,

so the chapter Git branches contain fully styled components. At the beginning of each chapter, you can switch to the chapter branch to see the styles for the code from the previous chapter.

Now you have all the specs written for this chapter. You've learned *how* to test component output. Before you move on to the next chapter, I want to take a minute to talk about *when* you should write tests for rendered component output.

3.8 When to test rendered component output

In testing, less is more. Every extra unit test you write *couples* your test code to your source code. When you write unit tests, you need to be more miserly than Scrooge McDuck.

> **DEFINITION** Coupling is the interdependence between modules of code. If test code is coupled to source code, it means the test code is dependent on the details of your source code, rather than the functionality of the code.

Tightly coupled code makes it difficult to refactor, because you can break tens of tests in a file when you decide to change the implementation. To avoid this, remember the following principles when testing component output:

- Test only output that is dynamically generated.
- Test only output that is part of the component contract.

Generally, you should test only output that's *dynamically generated*. Dynamically generated sounds very formal, but what it means is that a value is generated in the component using JavaScript. For example, an `Item` component at index 2 might have the class `item-2`, generated using the component index. You should write a test for this, because you're adding logic to generate the prop, and logic is prone to errors.

You should also test output that's part of the component contract. If it's part of the contract, then it's important enough to sacrifice the coupling of code.

The unit testing Goldilocks rule

Writing unit tests is a constant struggle between writing enough tests and not writing too many. I call this the unit testing Goldilocks rule—not too many, not too few, but just enough. Thousands of tests for a small application can be as damaging to development time as no tests.

In this book, I'll give you examples when you should write a test and when you shouldn't. The rules I lay out in this book aren't set in stone—they're general principles. You should decide on a test-by-test basis whether you should write a test for your component.

If you follow these rules, you will never write tests for presentational elements and static CSS classes. You should add presentational style without writing unit tests.

If you want to see what you built, you can run the dev server: `npm run serve`. Go to http://localhost:8080 in your browser. You should see a great Hacker News application, using your static data. In the next chapter, you'll flesh out the application by learning to test methods.

Summary

- You can test DOM attributes, component props, text, and classes using Vue Test Utils Wrapper methods.
- `find` and `findAll` return wrappers of nodes in the rendered output of a component mounted with Vue Test Utils.
- You should only test component output if the output is generated dynamically or the output is part of the component contract

Exercises

1. Write a test in src/components/__tests__/Item.spec.js to test that `Item` renders the `item.score` and `item.author` values. When you have written the tests, make them pass by adding code to src/components/Item.vue.
2. Write a test to check that the following component renders the `Child` component with the correct `test-prop` value of `some-value`:

```
// TestComponent.vue
<template>
  <div>
    <child testProp="some-value" />
  </div>
</template>

<script>
  import Child from './Child.vue'

  export default {
    components: { Child }
  }
</script>
// Child.vue
<script>
  export default {
    props: ['testProp']
  }
</script>
```

3. Write a test to check that the `<a>` tag has an `href` with the value of https://google.com:

```
// TestComponent.vue
<template>
  <div>
    <a href="https://google.com">Link</a>
  </div>
</template>
```

4 Write a test to check that the <p> tag has a color style with the value of red:

```
// TestComponent.vue
<template>
  <div>
    <p style="color: red">Paragraph</p>
  </div>
</template>
```

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]