Testing

Why Test?

Automated tests help you and your team build complex Vue applications quickly and confidently by preventing regressions and encouraging you to break apart your application into testable functions, modules, classes, and components. As with any application, your new Vue app can break in many ways, and it's important that you can catch these issues and fix them before releasing.

In this guide, we'll cover basic terminology and provide our recommendations on which tools to choose for your Vue 3 application.

There is one Vue-specific section covering composables. See Testing Composables below for more details.

When to Test

Start testing early! We recommend you begin writing tests as soon as you can. The longer you wait to add tests to your application, the more dependencies your application will have, and the harder it will be to start.

Testing Types

When designing your Vue application's testing strategy, you should leverage the following testing types:

- **Unit**: Checks that inputs to a given function, class, or composable are producing the expected output or side effects.

and require more time to execute.

- **End-to-end**: Checks features that span multiple pages and makes real network requests against your production-built Vue application. These tests often involve standing up a database or other backend.

Each testing type plays a role in your application's testing strategy, and each will protect you against different types of issues.

Overview

We will briefly discuss what each of these are, how they can be implemented for Vue applications, and provide some general recommendations.

Unit Testing

Unit tests are written to verify that small, isolated units of code are working as expected. A unit test usually covers a single function, class, composable, or module. Unit tests focus on logical correctness and only concern themselves with a small portion of the application's overall functionality. They may mock large parts of your application's environment (e.g. initial state, complex classes, 3rd party modules, and network requests).

In general, unit tests will catch issues with a function's business logic and logical correctness.

Take for example this `increment` function:

JS helpers.js

```js
export function increment(current, max = 10) {
  if (current < max) {
    return current + 1
  }
  return current
}
```

Because it's very self-contained, it'll be easy to invoke the increment function and assert that it returns what it's supposed to, so we'll write a Unit Test.

If any of these assertions fail, it's clear that the issue is contained within the `increment` function.

```js
import { increment } from './helpers'

describe('increment', () => {
  test('increments the current number by 1', () => {
    expect(increment(0, 10)).toBe(1)
  })

  test('does not increment the current number over the max', () => {
    expect(increment(10, 10)).toBe(10)
  })

  test('has a default max of 10', () => {
    expect(increment(10)).toBe(10)
  })
})
```

As mentioned previously, unit testing is typically applied to self-contained business logic, components, classes, modules, or functions that do not involve UI rendering, network requests, or other environmental concerns.

These are typically plain JavaScript / TypeScript modules unrelated to Vue. In general, writing unit tests for business logic in Vue applications does not differ significantly from applications using other frameworks.

There are two instances where you DO unit test Vue-specific features:

1. Composables
2. Components

Composables

One category of functions specific to Vue applications is Composables, which may require special handling during tests. See Testing Composables below for more details.

Unit Testing Components

A component can be tested in two ways:

1. Whitebox: Unit Testing

   Tests that are "Whitebox tests" are aware of the implementation details and dependencies of a component. They are focused on **isolating** the component under test. These tests will usually involve mocking some, if not all of your component's children, as well as setting up plugin state and dependencies (e.g. Pinia).

2. Blackbox: Component Testing

component and the entire system. They usually render all child components and are considered more of an "integration test". See the Component Testing recommendations below.

Recommendation

- Vitest

  Since the official setup created by `create-vue` is based on Vite, we recommend using a unit testing framework that can leverage the same configuration and transform pipeline directly from Vite. Vitest is a unit testing framework designed specifically for this purpose, created and maintained by Vue / Vite team members. It integrates with Vite-based projects with minimal effort, and is blazing fast.

Other Options

- Jest is a popular unit testing framework. However, we only recommend Jest if you have an existing Jest test suite that needs to be migrated over to a Vite-based project, as Vitest offers a more seamless integration and better performance.

Component Testing

In Vue applications, components are the main building blocks of the UI. Components are therefore the natural unit of isolation when it comes to validating your application's behavior. From a granularity perspective, component testing sits somewhere above unit testing and can be considered a form of integration testing. Much of your Vue Application should be covered by a component test and we recommend that each Vue component has its own spec file.

Component tests should catch issues relating to your component's props, events, slots that it provides, styles, classes, lifecycle hooks, and more.

Component tests should not mock child components, but instead test the interactions between your component and its children by interacting with the components as a user would. For example, a component test should click on an element like a user would instead of programmatically interacting with the component.

Component tests should focus on the component's public interfaces rather than internal implementation details. For most components, the public interface is limited to: events emitted, props, and slots. When testing, remember to **test what a component does, not how it does it**.

- For **Visual** logic: assert correct render output based on inputted props and slots.
- For **Behavioral** logic: assert correct render updates or emitted events in response to user input events.

  In the below example, we demonstrate a Stepper component that has a DOM element labeled "increment" and can be clicked. We pass a prop called `max` that prevents the Stepper from being incremented past `2`, so if we click the button 3 times, the UI should still say `2`.

  We know nothing about the implementation of Stepper, only that the "input" is the `max` prop and the "output" is the state of the DOM as the user will see it.

Vue Test Utils | Cypress | Testing Library

```js
const valueSelector = '[data-testid=stepper-value]'
const buttonSelector = '[data-testid=increment]'

const wrapper = mount(Stepper, {
  props: {
    max: 1
  }
})

expect(wrapper.find(valueSelector).text()).toContain('0')

await wrapper.find(buttonSelector).trigger('click')

expect(wrapper.find(valueSelector).text()).toContain('1')
```

DON'T

- Don't assert the private state of a component instance or test the private methods of a component. Testing implementation details makes the tests brittle, as they are more likely to break and require updates when the implementation changes.

  The component's ultimate job is rendering the correct DOM output, so tests focusing on the DOM output provide the same level of correctness assurance (if not more) while being more robust and resilient to change.

  Don't rely exclusively on snapshot tests. Asserting HTML strings does not describe correctness. Write tests with intentionality.

  If a method needs to be tested thoroughly, consider extracting it into a standalone utility function and write a dedicated unit test for it. If it cannot be extracted cleanly, it may be tested as a part of a component, integration, or end-to-end test that covers it.

Recommendation

- Cypress Component Testing for components whose expected behavior depends on properly rendering styles or triggering native DOM events. It can be used with Testing Library via @testing-library/cypress.

The main differences between Vitest and browser-based runners are speed and execution context. In short, browser-based runners, like Cypress, can catch issues that node-based runners, like Vitest, cannot (e.g. style issues, real native DOM events, cookies, local storage, and network failures), but browser-based runners are *orders of magnitude slower than Vitest* because they do open a browser, compile your stylesheets, and more. Cypress is a browser-based runner that supports component testing. Please read Vitest's comparison page for the latest information comparing Vitest and Cypress.

Mounting Libraries

Component testing often involves mounting the component being tested in isolation, triggering simulated user input events, and asserting on the rendered DOM output. There are dedicated utility libraries that make these tasks simpler.

- `@vue/test-utils` is the official low-level component testing library that was written to provide users access to Vue specific APIs. It's also the lower-level library `@testing-library/vue` is built on top of.

- `@testing-library/vue` is a Vue testing library focused on testing components without relying on implementation details. Its guiding principle is that the more tests resemble the way software is used, the more confidence they can provide.

We recommend using `@vue/test-utils` for testing components in applications. `@testing-library/vue` has issues with testing asynchronous component with Suspense, so it should be used with caution.

Other Options

- Nightwatch is an E2E test runner with Vue Component Testing support. (Example Project)

- WebdriverIO for cross-browser component testing that relies on native user interaction based on standardized automation. It can also be used with Testing Library.

E2E Testing

to production. As a result, end-to-end (E2E) tests provide coverage on what is arguably the most important aspect of an application: what happens when users actually use your applications.

End-to-end tests focus on multi-page application behavior that makes network requests against your production-built Vue application. They often involve standing up a database or other backend and may even be run against a live staging environment.

End-to-end tests will often catch issues with your router, state management library, top-level components (e.g. an App or Layout), public assets, or any request handling. As stated above, they catch critical issues that may be impossible to catch with unit tests or component tests.

End-to-end tests do not import any of your Vue application's code but instead rely completely on testing your application by navigating through entire pages in a real browser.

End-to-end tests validate many of the layers in your application. They can either target your locally built application or even a live Staging environment. Testing against your Staging environment not only includes your frontend code and static server but all associated backend services and infrastructure.

> The more your tests resemble how your software is used, the more confidence they can give you. - Kent C. Dodds - Author of the Testing Library

By testing how user actions impact your application, E2E tests are often the key to higher confidence in whether an application is functioning properly or not.

Choosing an E2E Testing Solution

While end-to-end (E2E) testing on the web has gained a negative reputation for unreliable (flaky) tests and slowing down development processes, modern E2E tools have made strides forward to create more reliable, interactive, and useful tests. When choosing an E2E testing framework, the following sections provide some guidance on things to keep in mind when choosing a testing framework for your application.

Cross-browser testing

One of the primary benefits that end-to-end (E2E) testing is known for is its ability to test your application across multiple browsers. While it may seem desirable to have 100% cross-browser coverage, it is important to note that cross browser testing has diminishing returns on a team's resources due to the additional time and machine power required to run them consistently. As a result, it is important to be mindful of this trade-off when choosing the amount of cross-browser testing your application needs.

Faster feedback loops

deployment (CI/CD) pipelines. Modern E2E testing frameworks have helped to solve this by adding features like parallelization, which allows for CI/CD pipelines to often run magnitudes faster than before. In addition, when developing locally, the ability to selectively run a single test for the page you are working on while also providing hot reloading of tests can help boost a developer's workflow and productivity.

First-class debugging experience

While developers have traditionally relied on scanning logs in a terminal window to help determine what went wrong in a test, modern end-to-end (E2E) test frameworks allow developers to leverage tools they are already familiar with, e.g. browser developer tools.

Visibility in headless mode

When end-to-end (E2E) tests are run in continuous integration/deployment pipelines, they are often run in headless browsers (i.e., no visible browser is opened for the user to watch). A critical feature of modern E2E testing frameworks is the ability to see snapshots and/or videos of the application during testing, providing some insight into why errors are happening. Historically, it was tedious to maintain these integrations.

Recommendation

- Playwright is a great E2E testing solution that supports Chromium, WebKit, and Firefox. Test on Windows, Linux, and macOS, locally or on CI, headless or headed with native mobile emulation of Google Chrome for Android and Mobile Safari. It has an informative UI, excellent debuggability, built-in assertions, parallelization, traces and is designed to eliminate flaky tests. Support for Component Testing is available, but marked experimental. Playwright is open source and maintained by Microsoft.
- Cypress has an informative graphical interface, excellent debuggability, built-in assertions, stubs, flake-resistance, and snapshots. As mentioned above, it provides stable support for Component Testing. Cypress supports Chromium-based browsers, Firefox, and Electron. WebKit support is available, but marked experimental. Cypress is MIT-licensed, but some features like parallelization require a subscription to Cypress Cloud.

Other Options

- Nightwatch is an E2E testing solution based on Selenium WebDriver. This gives it the widest browser support range, including native mobile testing. Selenium-based solutions

WebDriver protocol.

Recipes

Adding Vitest to a Project

In a Vite-based Vue project, run:

```sh
> npm install -D vitest happy-dom @testing-library/vue
```

Next, update the Vite configuration to add the `test` option block:

vite.config.js

```js
import { defineConfig } from 'vite'

export default defineConfig({
  // ...
  test: {
    // enable jest-like global test APIs
    globals: true,
    // simulate DOM with happy-dom
    // (requires installing happy-dom as a peer dependency)
    environment: 'happy-dom'
  }
})
```

> ⓘ TIP
>
> If you use TypeScript, add `vitest/globals` to the `types` field in your `tsconfig.json`.
>
> tsconfig.json
>
> ```json
> {
>   "compilerOptions": {
>     "types": ["vitest/globals"]
>   }
> }
> ```

Then, create a file ending in `*.test.js` in your project. You can place all test files in a test directory in the project root or in test directories next to your source files. Vitest will

JS MyComponent.test.js

```js
import { render } from '@testing-library/vue'
import MyComponent from './MyComponent.vue'

test('it should work', () => {
  const { getByText } = render(MyComponent, {
    props: {
      /* ... */
    }
  })

  // assert output
  getByText('...')
})
```

Finally, update `package.json` to add the test script and run it:

package.json

```json
{
  // ...
  "scripts": {
    "test": "vitest"
  }
}
```

```sh
> npm test
```

Testing Composables

> This section assumes you have read the Composables section.

When it comes to testing composables, we can divide them into two categories: composables that do not rely on a host component instance, and composables that do.

A composable depends on a host component instance when it uses the following APIs:

- Lifecycle hooks
- Provide / Inject

If a composable only uses Reactivity APIs, then it can be tested by directly invoking it and asserting its returned state/methods:

JS counter.js

```
export function useCounter() {
  const count = ref(0)
  const increment = () => count.value++

  return {
    count,
    increment
  }
}
```

JS counter.test.js

```
import { useCounter } from './counter.js'

test('useCounter', () => {
  const { count, increment } = useCounter()
  expect(count.value).toBe(0)

  increment()
  expect(count.value).toBe(1)
})
```

A composable that relies on lifecycle hooks or Provide / Inject needs to be wrapped in a host component to be tested. We can create a helper like the following:

JS test-utils.js

```
import { createApp } from 'vue'

export function withSetup(composable) {
  let result
  const app = createApp({
    setup() {
      result = composable()
      // suppress missing template warning
      return () => {}
    }
  })
  app.mount(document.createElement('div'))
  // return the result and the app instance
  // for testing provide/unmount
  return [result, app]
}
```

JS foo.test.js

```
import { withSetup } from './test-utils'
import { useFoo } from './foo'

test('useFoo', () => {
  const [result, app] = withSetup(() => useFoo(123))
```

```
  // run assertions
  expect(result.foo.value).toBe(1)
  // trigger onUnmounted hook if needed
  app.unmount()
})
```

For more complex composables, it could also be easier to test it by writing tests against the wrapper component using Component Testing techniques.

Edit this page on GitHub

Previous
State Management

Next
Server-Side Rendering (SSR)