Nuxt Fundamentals

Vitest Unit Testing

Peter Kassenaar –
info@kassenaar.com

Vitest Unit Testing

Using `vitest` – the test runner for Vite-applications.

Looks *a lot* like Jest and is often considered a drop-in replacement

What is ‘Testing’?

“You expect your components or apps to behave in a certain way.

You test if this behavior meets your expectations”

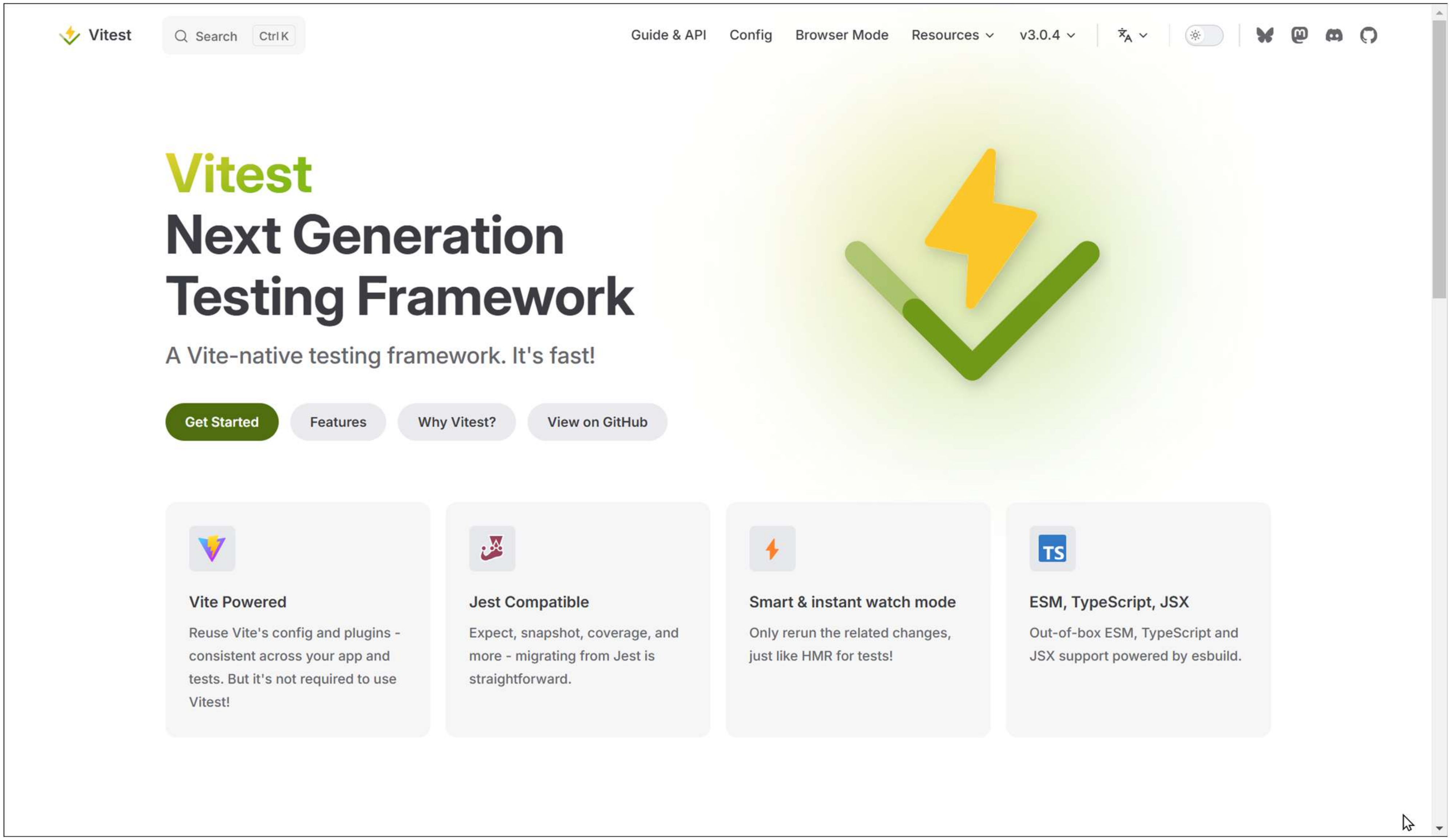
2 Basic Types of Testing

- Unit Testing
- Test individual components (‘units’) of the application
- Packages `happy-dom` or `jsdom` to simulate a runtime Nuxt environment

- End-to-End Testing
- Test the complete application
- Spin up server, navigate to route, perform actions, etc.
- Also: ‘scenario testing’
- Package `playwright` for `e2e-`testing

Test Runner

- run the `*.spec.ts` or `*.test.ts` files you created using one of the options above. Popular choices: `vitest`, `cucumber`, `jest`, `playwright`



Vitest
Next Generation Testing Framework

A Vite-native testing framework. It's fast!

Vite Powered
Reuse Vite's config and plugins - consistent across your app and tests. But it's not required to use Vitest!

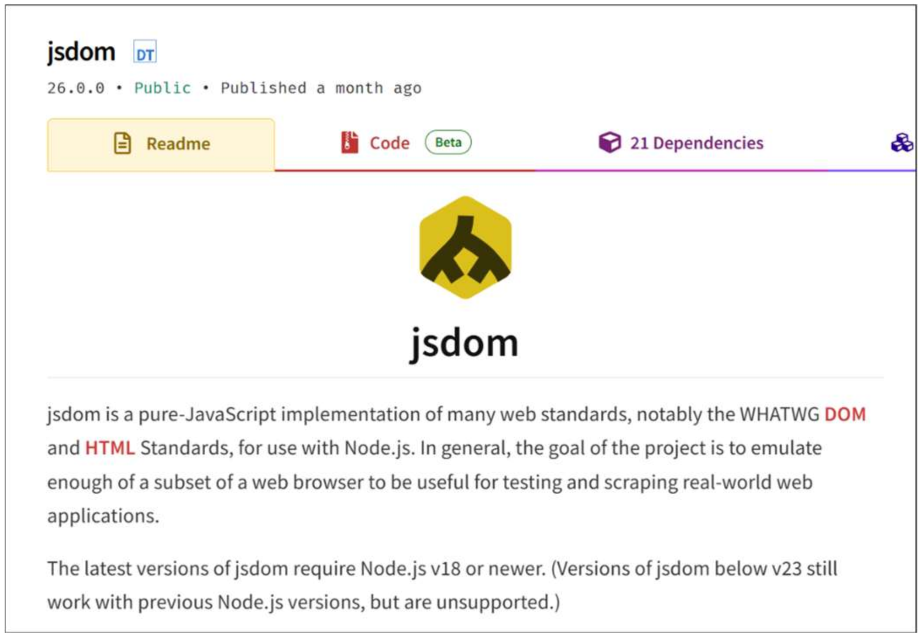
Jest Compatible
Expect, snapshot, coverage, and more - migrating from Jest is straightforward.

Smart & instant watch mode
Only rerun the related changes, just like HMR for tests!

ESM, TypeScript, JSX
Out-of-box ESM, TypeScript and JSX support powered by esbuild.

https://vitest.dev/

More testing tools…



jsdom DT

26.0.0 • Public • Published a month ago

jsdom

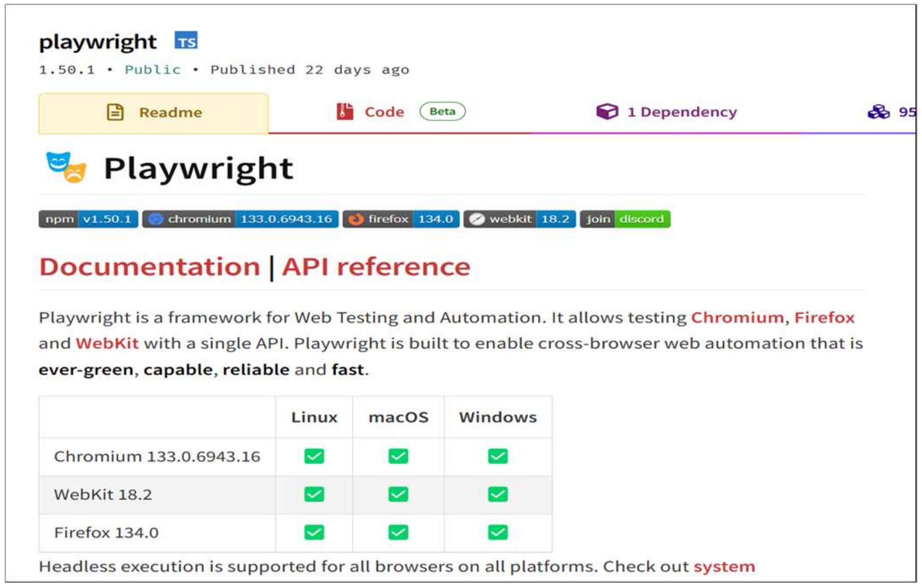
jsdom is a pure-JavaScript implementation of many web standards, notably the WHATWG DOM and HTML Standards, for use with Node.js. In general, the goal of the project is to emulate enough of a subset of a web browser to be useful for testing and scraping real-world web applications.

The latest versions of jsdom require Node.js v18 or newer. (Versions of jsdom below v23 still work with previous Node.js versions, but are unsupported.)

happy-dom TS

17.1.2 • Public • Published 11 hours ago

A JavaScript implementation of a web browser without its graphical user interface.



playwright TS

1.50.1 • Public • Published 22 days ago

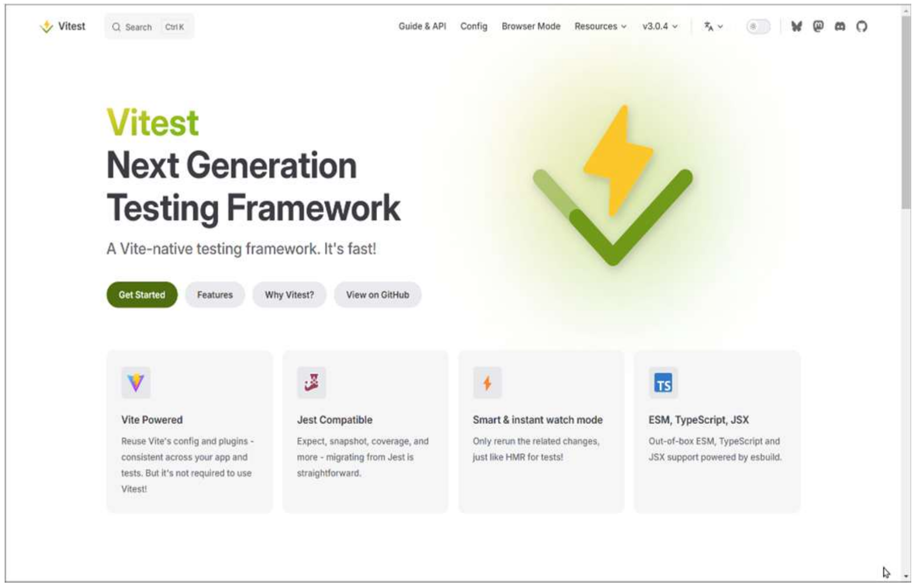
Playwright

Documentation | API reference

Playwright is a framework for Web Testing and Automation. It allows testing Chromium, Firefox and WebKit with a single API. Playwright is built to enable cross-browser web automation that is **ever-green**, **capable**, **reliable** and **fast**.

	Linux	macOS	Windows
Chromium 133.0.6943.16	✓	✓	✓
WebKit 18.2	✓	✓	✓
Firefox 134.0	✓	✓	✓

Headless execution is supported for all browsers on all platforms. Check out system



Vitest
Next Generation Testing Framework

A Vite-native testing framework. It's fast!

Vite Powered
Reuse Vite's config and plugins - consistent across your app and tests. But it's not required to use Vitest!

Jest Compatible
Expect, snapshot, coverage, and more - migrating from Jest is straightforward.

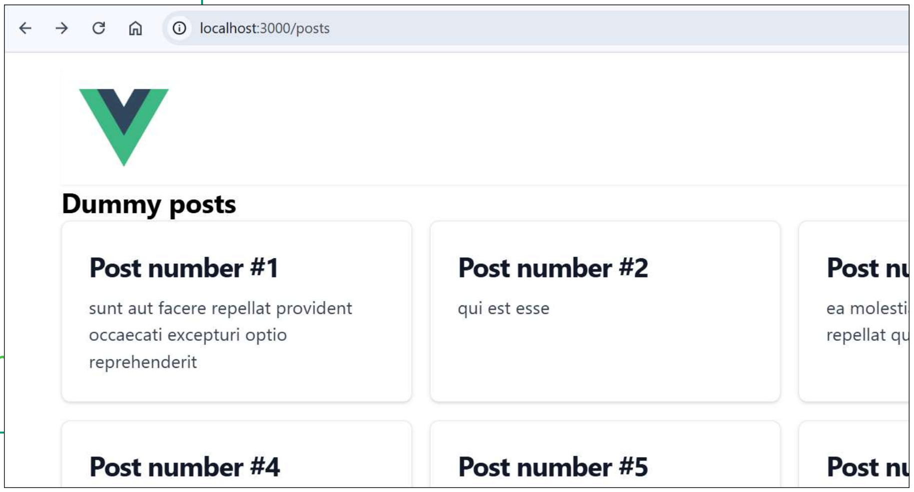
Smart & instant watch mode
Only rerun the related changes, just like HMR for tests!

ESM, TypeScript, JSX
Out-of-box ESM, TypeScript and JSX support powered by esbuild.

Introduction – what are we going to test?

1. Short introduction in Unit Testing
2. Starting point: `./pages/post/index.vue`
   1. From `./examples/150/reusable components`
   2. It has: (reactive) variables, async fetch, lifecycle, data etc.

```
<script setup lang="ts">
import {ref, onMounted} from 'vue';

// TypeScript interface for Posts.
import type {Post} from "@/types/Post";

// URL to fetch data from.
const url = 'https://jsonplaceholder.typicode.com/posts';


// Reactive variables
const posts = ref<Post[]>([]); // store the posts
const isLoading = ref(true); // handle loading state
const error = ref<string | null>(null); // handle possible er
…};
```



Dummy posts

Post number #1

sunt aut facere repellat provident occaecati excepturi optio reprehenderit

Post number #2

qui est esse

Post nu

ea molesti repellat qu

Post number #4

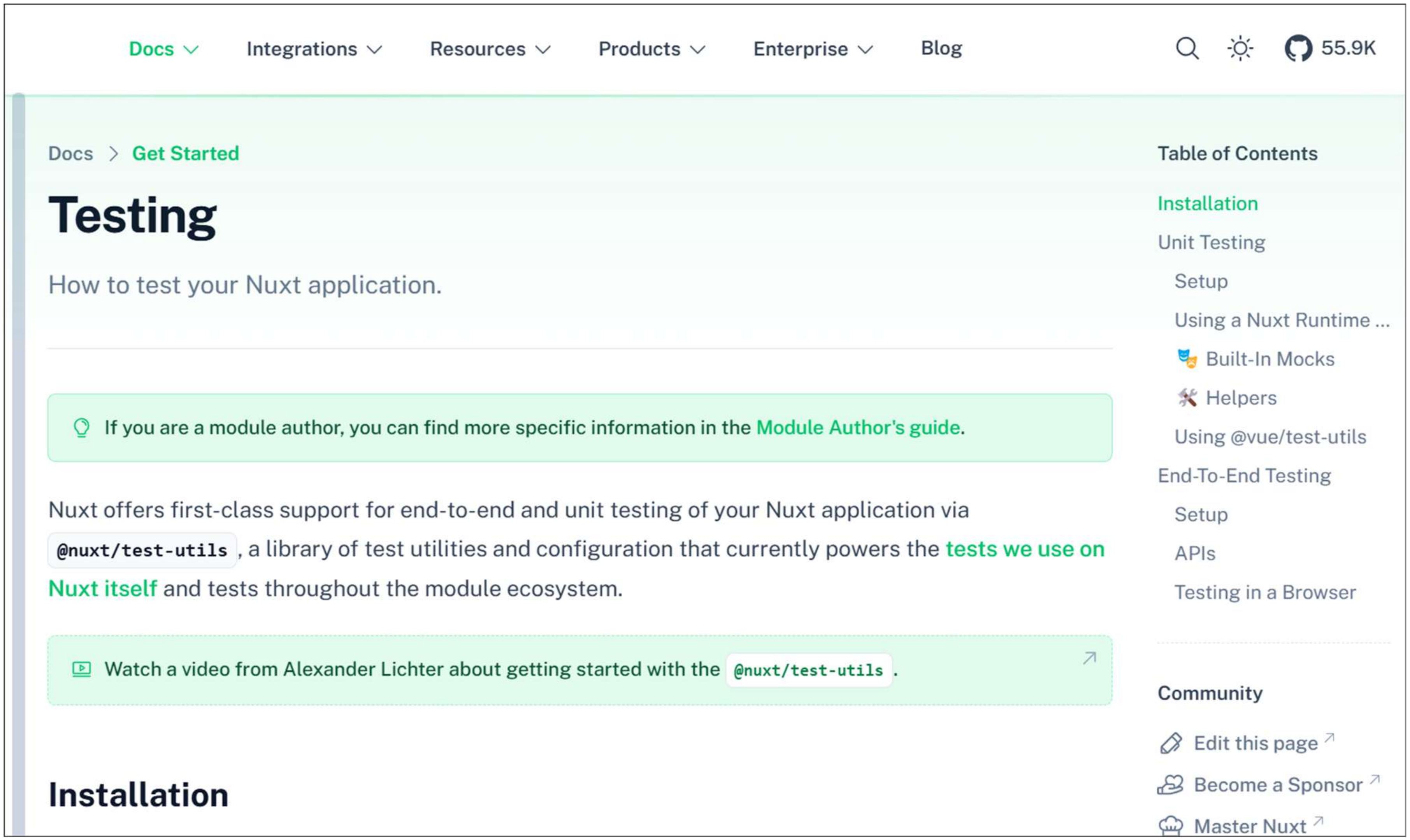
Post number #5

Post nu

Vitest Unit Testing

- Short explanation (true for all framework tests ☺):
  - "To write a unit test, we need to simulate its behavior, interactions, and lifecycle hooks"
- Configure a mock fetch-request to test the `fetchPosts()` functionality,
- mock the `onMounted` lifecycle,
- cover all UI states: `loading`, `error`, and rendering of post.
- Info on unit testing:
  - https://nuxt.com/docs/getting-started/testing
  - https://masteringnuxt.com/blog/unit-testing-in-nuxt

Official docs:



Docs > Get Started

Testing

How to test your Nuxt application.

If you are a module author, you can find more specific information in the Module Author's guide.

Nuxt offers first-class support for end-to-end and unit testing of your Nuxt application via `@nuxt/test-utils`, a library of test utilities and configuration that currently powers the tests we use on Nuxt itself and tests throughout the module ecosystem.

Watch a video from Alexander Lichter about getting started with the `@nuxt/test-utils`.

Installation

Table of Contents

Community

Edit this page
Become a Sponsor
Master Nuxt

https://nuxt.com/docs/getting-started/testing

First: additional dependencies to install

1. `vitest` - The test runner
2. `@vue/test-utils` - For testing Vue components
3. `happy-dom` or `js-dom` – optional but recommended

   lightweight DOM for testing (since `vitest` doesn’t use a browser by default)
4. `@testing-library/vue` - A friendly utility for testing Vue components (optional, for user-focused testing)
5. `@vitejs/plugin-vue` – for compiling `*.vue` files during tests

```
npm install vitest @vue/test-utils @vitejs/plugin-vue
    happy-dom --save-dev
```

```
npm install @testing-library/vue --save-dev
```
(optional)

`package.json` after installing dependencies

```json
{
  "name": "nuxt-app",
   …
  "devDependencies": {
    "@vue/test-utils": "^2.4.6",
    "@vitejs/plugin-vue": "^5.2.1"
    "happy-dom": "^16.8.1",
    "vitest": "^3.0.4"
  }
}
```

```json
"scripts": {
    …,
   "test": "vitest"
},
```

Add script to start the test runner

Configure vitest – `vitest.config.ts`

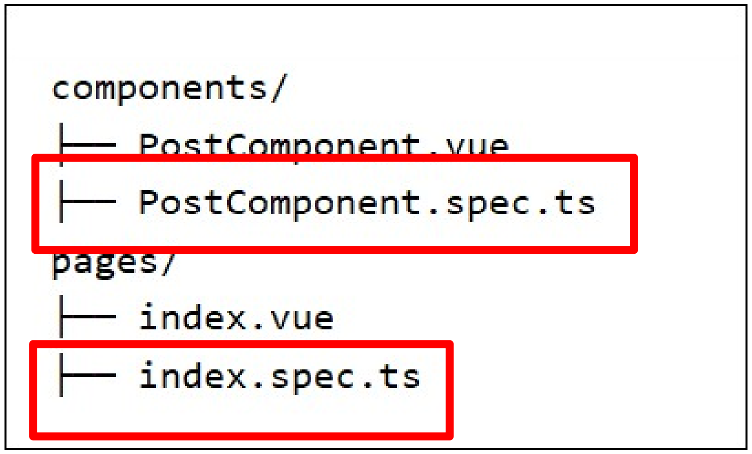
- Config file NOT strictly necessary for very simple applications
- In real life: almost always `vitest.config.ts` for configuring the test runner.
- In our case: minimal setup for using `happy-dom` and parsing `*.vue` files
- Configuration ensures the Vue templates are parsed and rendered correctly during tests
- Configuration ensures the `‘@’` is replaced with actual path in the tests
  - Remember: in tests you have to do EVERYTHING yourself

Vitest.config.ts

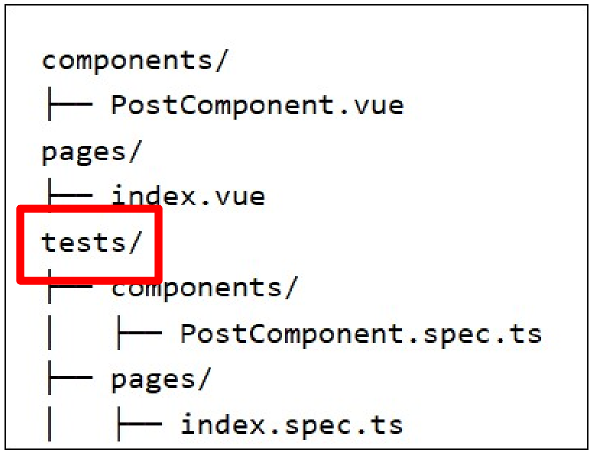
```
// vitest.config.ts
import { defineConfig } from 'vitest/config';
import vue from '@vitejs/plugin-vue';
import path from 'path';

export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      // Make sure aliases like '@' work in tests, maybe do the same for ‘~’
      '@': path.resolve(__dirname, './'),
    },
  },
  test: {
    globals: true,
    environment: 'happy-dom', // Simulate a DOM-like environment for Vue testing
  },
});
```

Conventions on testing files

- Where should we put our testing files?
  - There are options. No strict rules
  - Side-by-side: next to the Vue component
  - Filenames: `PageComponent.spec.ts` or `PageComponent.test.ts`
  - Testing folder: place unit tests in a designated folder
  - Foldernames: `tests/` or `__tests__/`
- Vue, mostly: side-by-side. Also in line with Angular/React



```
components/
├── PostComponent.vue
├── PostComponent.spec.ts
pages/
├── index.vue
├── index.spec.ts
```



```
components/
├── PostComponent.vue
pages/
├── index.vue
tests/
├── components/
│   ├── PostComponent.spec.ts
├── pages/
│   ├── index.spec.ts
```

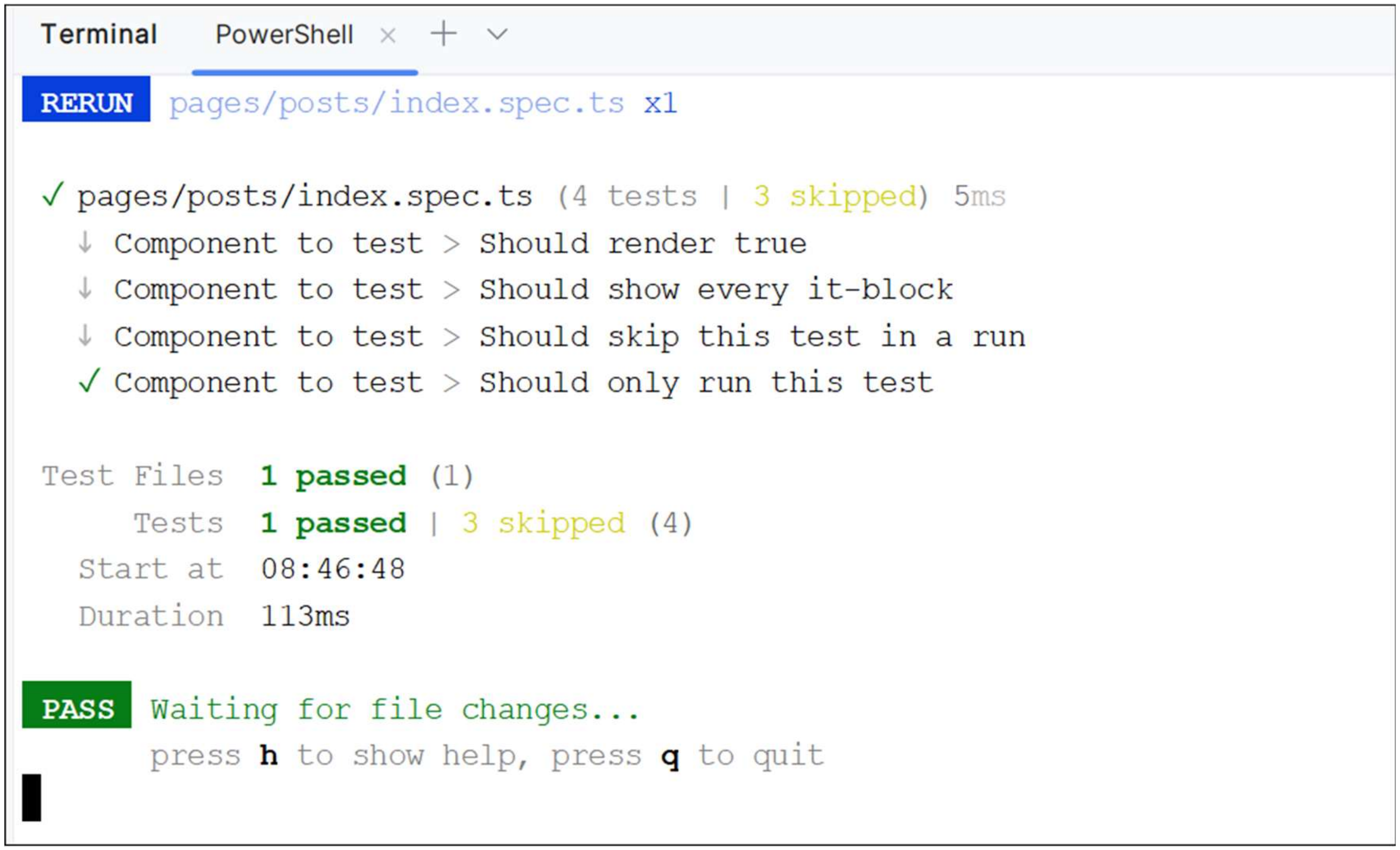
Anatomy of a testing file

```
import {afterEach, beforeEach, describe, expect, it} from 'vitest';

describe('Component to test', () => {
  beforeEach(() => {
    // Setup...
  });

  afterEach(() => {
    // Teardown...
  });

  it('Should render true', () => {
    expect(true).toBeTruthy();
  });

  it('Should show every it-block',  () => {

  });

  it.skip('Should skip this test in a run', () => {
    // expect() should be skipped
  });

  it.only('Should only run this test',  () => {
    // only this expect() should run
  });
});
```

Result – very simple

```
"scripts": {
  …
  "test": "vitest"
},
```

package.json



```
RERUN  pages/posts/index.spec.ts x1

 ✓ pages/posts/index.spec.ts (4 tests | 3 skipped) 5ms
   ↓ Component to test > Should render true
   ↓ Component to test > Should show every it-block
   ↓ Component to test > Should skip this test in a run
   ✓ Component to test > Should only run this test

 Test Files  1 passed (1)
      Tests  1 passed | 3 skipped (4)
   Start at  08:46:48
   Duration  113ms

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Verbose output

- Note: we're using the *verbose* output here.
  - By default, Vitest intentionally provides a minimal summary in the terminal during testing
- Use `--reporter=verbose` if you want more details
- OR (like in our case): update `vitest.config.ts`

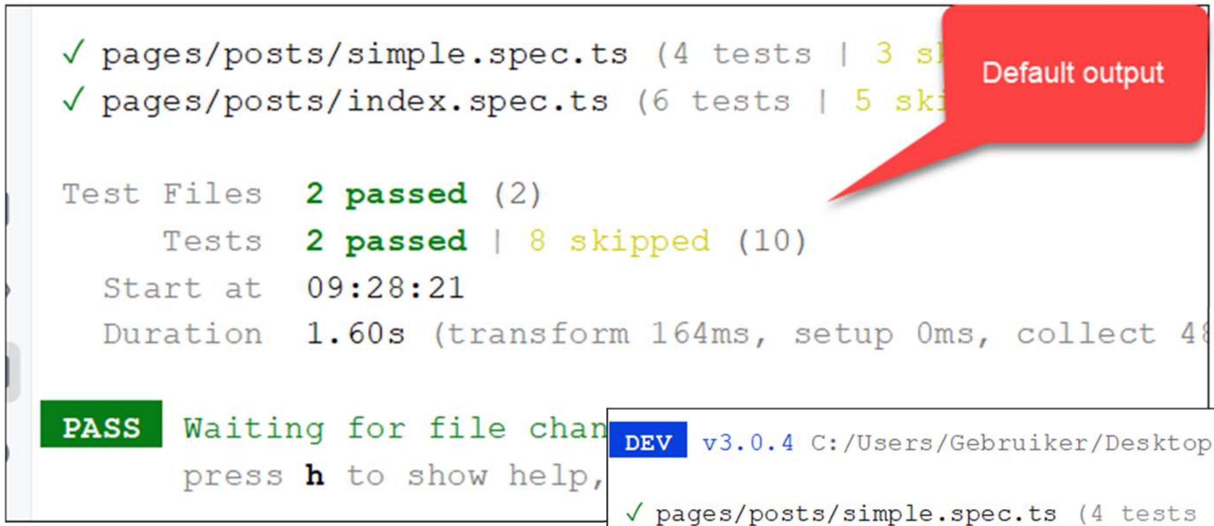
```
// vitest.config.ts
export default defineConfig({
   …
  test: {
    reporters: ['verbose’],
    …
  },
});
```

Tip: Keyboard Shortcut

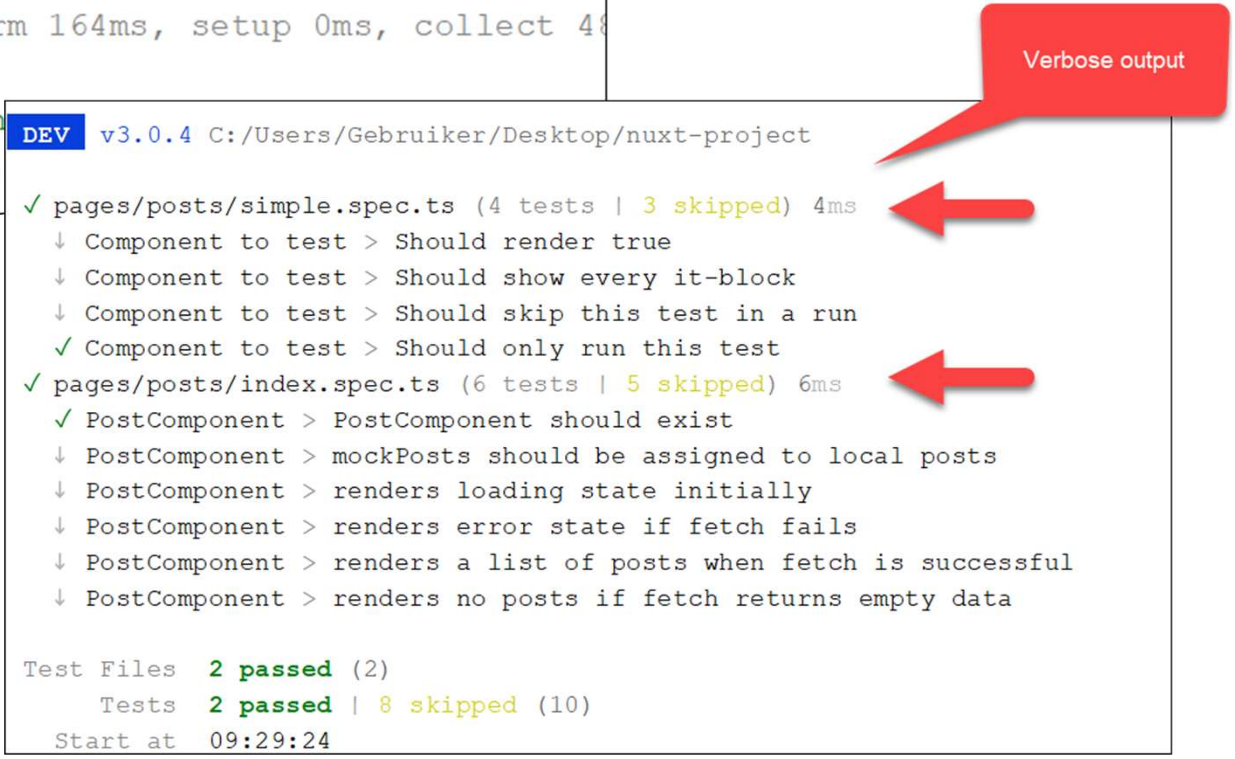
When Vitest is running in watch mode (like `PASS Waiting for file changes...`), press `h` to view `help` options.

One of the options includes switching reporters dynamically.

Default vs. Verbose output



```
✓ pages/posts/simple.spec.ts (4 tests | 3 s
✓ pages/posts/index.spec.ts (6 tests | 5 ski

 Test Files  2 passed (2)
      Tests  2 passed | 8 skipped (10)
   Start at  09:28:21
   Duration  1.60s (transform 164ms, setup 0ms, collect 4

 PASS  Waiting for file chan
       press h to show help,
```



```
 DEV  v3.0.4 C:/Users/Gebruiker/Desktop/nuxt-project

✓ pages/posts/simple.spec.ts (4 tests | 3 skipped) 4ms
  ↓ Component to test > Should render true
  ↓ Component to test > Should show every it-block
  ↓ Component to test > Should skip this test in a run
  ✓ Component to test > Should only run this test
✓ pages/posts/index.spec.ts (6 tests | 5 skipped) 6ms
  ✓ PostComponent > PostComponent should exist
  ↓ PostComponent > mockPosts should be assigned to local posts
  ↓ PostComponent > renders loading state initially
  ↓ PostComponent > renders error state if fetch fails
  ↓ PostComponent > renders a list of posts when fetch is successful
  ↓ PostComponent > renders no posts if fetch returns empty data

Test Files  2 passed (2)
     Tests  2 passed | 8 skipped (10)
  Start at  09:29:24
```

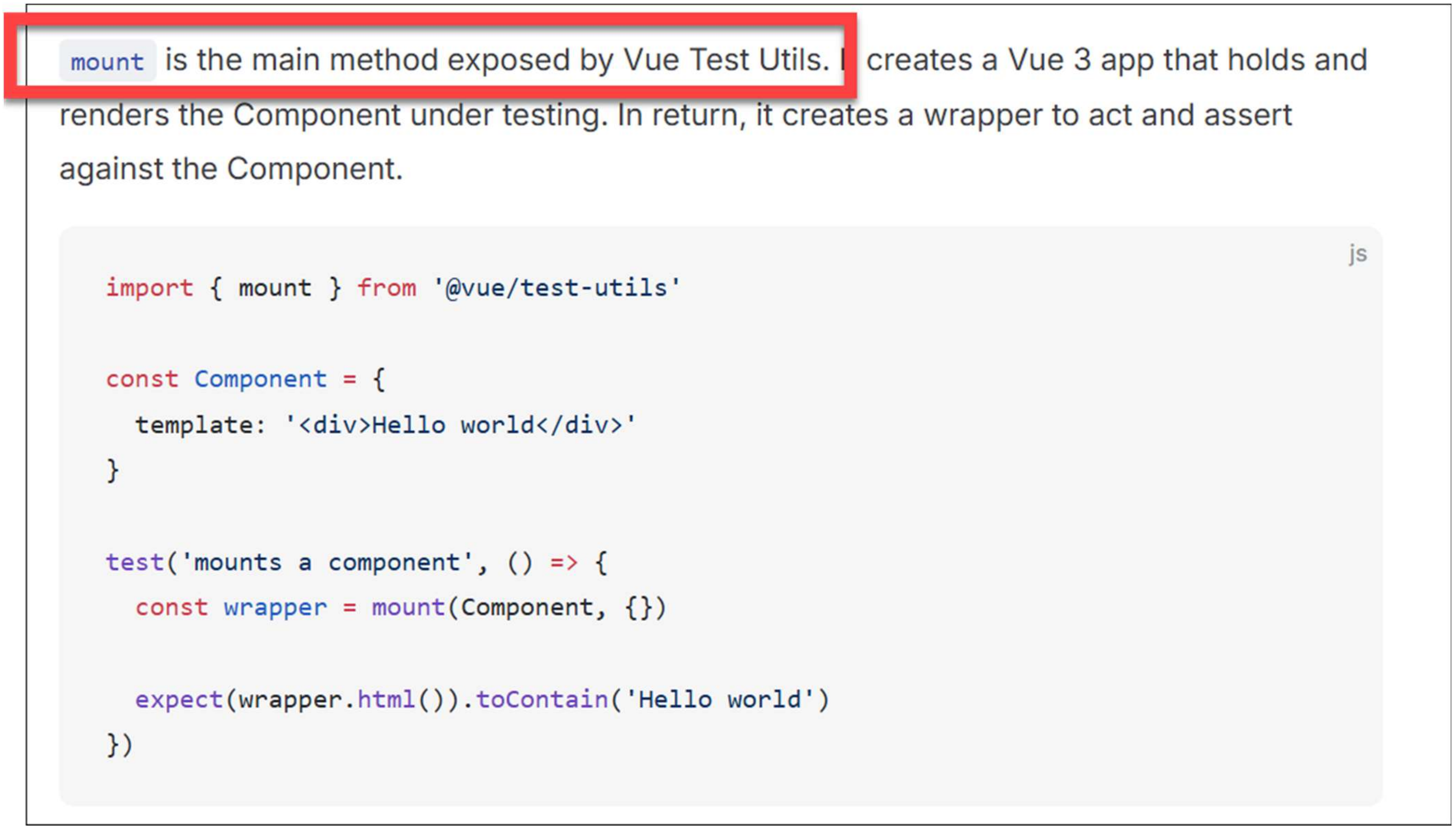
So a `*.spec.ts` file typically contains:

One or more `describe()` blocks

One or more `beforeEach()` blocks

One or (typically) more `it()` blocks, using logic, `expect()` statements and *matchers*

The importance of `mount`



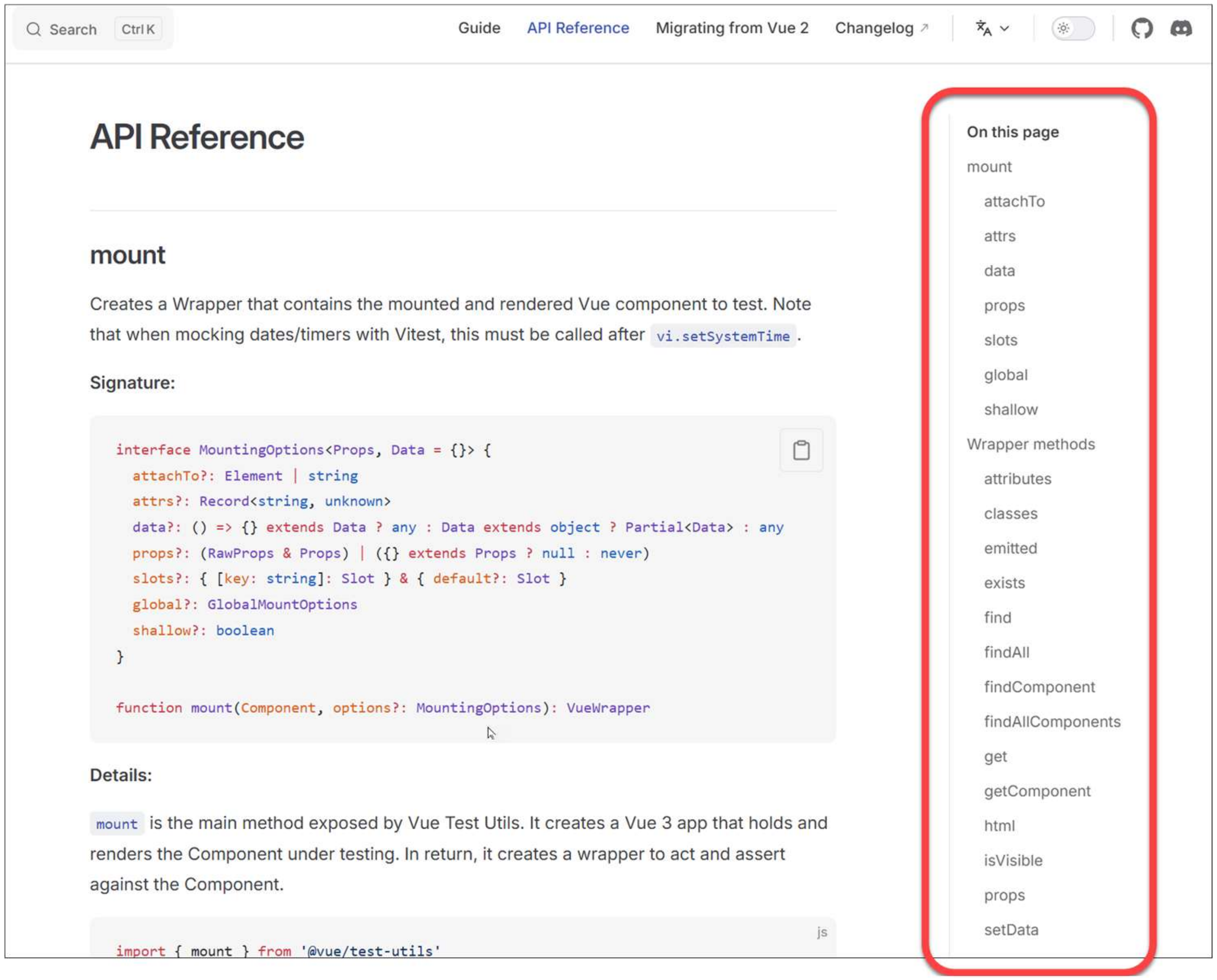
`mount` is the main method exposed by Vue Test Utils. It creates a Vue 3 app that holds and renders the Component under testing. In return, it creates a wrapper to act and assert against the Component.

js

```js
import { mount } from '@vue/test-utils'

const Component = {
  template: '<div>Hello world</div>'
}

test('mounts a component', () => {
  const wrapper = mount(Component, {})

  expect(wrapper.html()).toContain('Hello world')
})
```

https://test-utils.vuejs.org/api/

`mount` and wrapper methods



API Reference

mount

Creates a Wrapper that contains the mounted and rendered Vue component to test. Note that when mocking dates/timers with Vitest, this must be called after `vi.setSystemTime`.

Signature:

```
interface MountingOptions<Props, Data = {}> {
  attachTo?: Element | string
  attrs?: Record<string, unknown>
  data?: () => {} extends Data ? any : Data extends object ? Partial<Data> : any
  props?: (RawProps & Props) | ({} extends Props ? null : never)
  slots?: { [key: string]: Slot } & { default?: Slot }
  global?: GlobalMountOptions
  shallow?: boolean
}

function mount(Component, options?: MountingOptions): VueWrapper
```

Details:

`mount` is the main method exposed by Vue Test Utils. It creates a Vue 3 app that holds and renders the Component under testing. In return, it creates a wrapper to act and assert against the Component.

```
import { mount } from '@vue/test-utils'
```

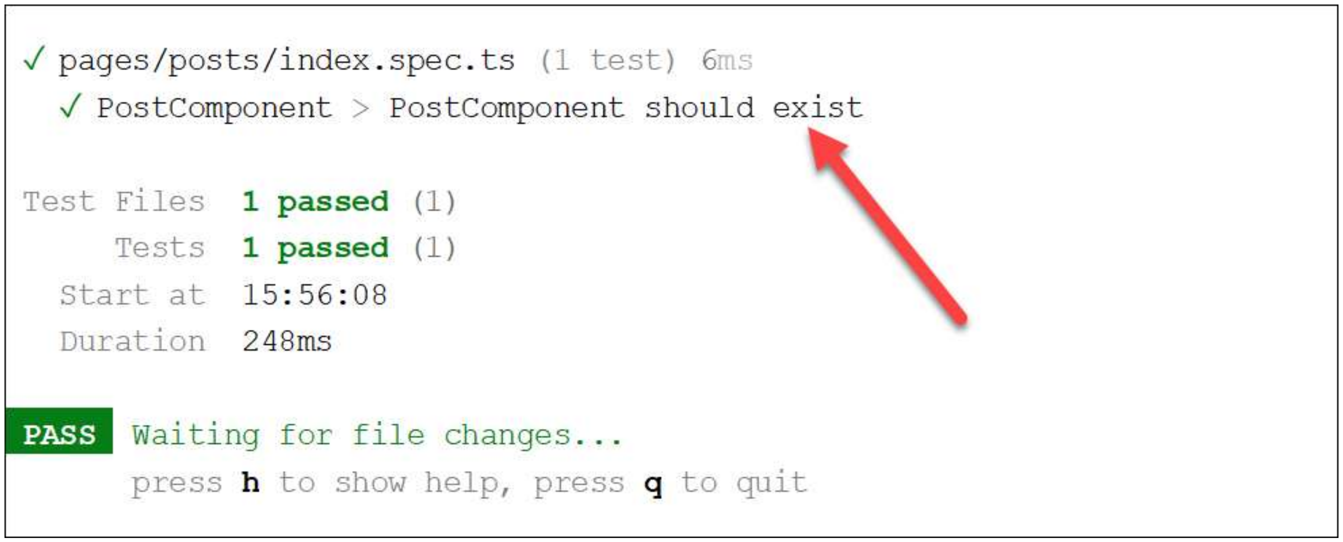
https://test-utils.vuejs.org/api/

1. Simple real Component testing file

```
import { mount } from '@vue/test-utils';
import { describe, it, beforeEach, afterEach, expect, vi } from 'vitest';
import PostComponent from './index.vue'; // Adjust the path if needed

// Mock data for testing
const mockPosts = [
  { id: 1, title: 'Post 1', body: 'Content of Post 1' },
  { id: 2, title: 'Post 2', body: 'Content of Post 2' },
];

describe('PostComponent', () => {
  beforeEach(() => {
    // Mock the global `fetch` function before each test
    global.fetch = vi.fn();
  });

  afterEach(() => {
    // Restore original implementations after each test
    vi.restoreAllMocks();
  });

  it('PostComponent should exist', () => {
    expect(PostComponent).toBeTruthy();
  });
});
```

Running the test

- Use the `test` script you defined earlier in your `package.json`
  - `npm run test`
  - Output be like:



```
✓ pages/posts/index.spec.ts (1 test) 6ms
  ✓ PostComponent > PostComponent should exist

Test Files  1 passed (1)
     Tests  1 passed (1)
  Start at  15:56:08
  Duration  248ms

PASS  Waiting for file changes...
      press h to show help, press q to quit
```

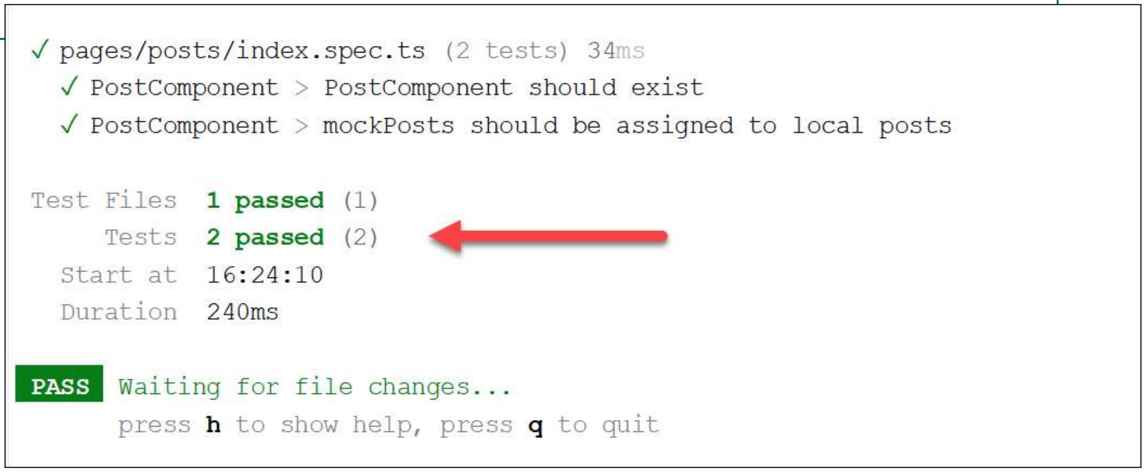
2. Testing local posts property

- Let’s say we want to assign the mocked posts to the local `posts` property and test if they can be assigned.
- Because of TypeScript, we need additional typings:

```
import type {Post} from "@/types/Post";

// Define a type for your test environment's component instance
type PostComponentInstance = {
  posts: Post[];
};
```

Mounting the instance, using a `wrapper`

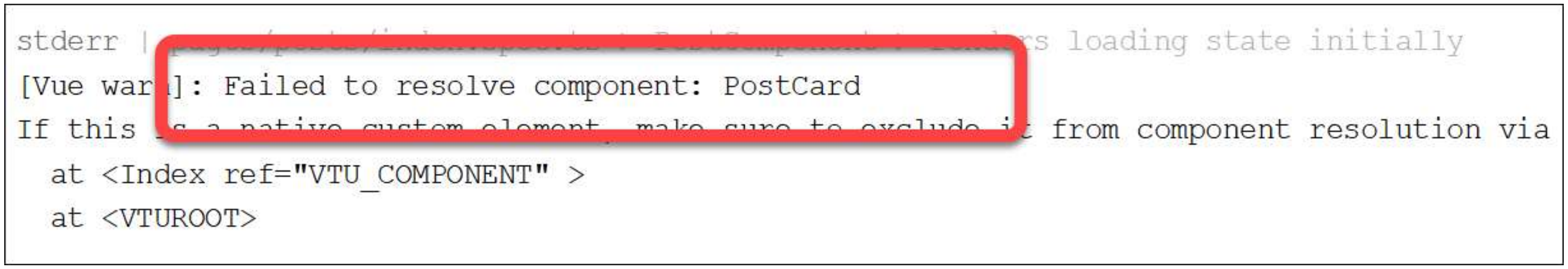
```
it('mockPosts should be assigned to local posts', async () => {
  const wrapper = mount<PostComponentInstance>(PostComponent); // Add instance type
  wrapper.vm.posts = mockPosts; // Access posts with proper typing

  await wrapper.vm.$nextTick(); // Wait for the reactive update

  expect(wrapper.vm.posts.length).toBe(2); // Check the length
  expect(wrapper.vm.posts).toEqual(mockPosts); // Assert equality
});
```



```
✓ pages/posts/index.spec.ts (2 tests) 34ms
  ✓ PostComponent > PostComponent should exist
  ✓ PostComponent > mockPosts should be assigned to local posts

 Test Files  1 passed (1)
      Tests  2 passed (2)
   Start at  16:24:10
   Duration  240ms

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Complaining that PostCard does not exist

- Since the original `index.vue` uses a `PostCard` child element to render individual posts, `vitest` is complaining:



```
stderr | pages/posts/index.spec.ts > PostComponent > renders loading state initially
[Vue warn]: Failed to resolve component: PostCard
If this is a native custom element, make sure to exclude it from component resolution via
  at <Index ref="VTU_COMPONENT" >
  at <VTUROOT>
```

Three possible solutions:

1. Mock `PostCard` by adding it as an empty stub
2. Import and register actual `PostCard` Component
3. Exclude `PostCard` rendering and ignore it altogether

Stubbing & mocking

Simulating stuff you don’t actually test.

`../210-vitest-stubbing`

Solution 1: mocking `<PostCard>`

- Since we only want to test the functionality of `index.vue`, we mock the rest of the components
- Effectively, they are thus ignored
- Update the `wrapper` definition

```
const wrapper = mount<PostComponentInstance>(PostComponent, {
    global: {
        stubs: {
            PostCard: true, // Mock <PostCard> with an empty stub
        },
    },
});
wrapper.vm.posts = mockPosts; // Access posts with proper typing
…
```

Solution 2 (less used)

- Actually import `PostCard` component and register it with the `wrapper`:
- Less used, because sometimes we need to import dozens of components, only to satisfy the test runner.
  - This is inefficient

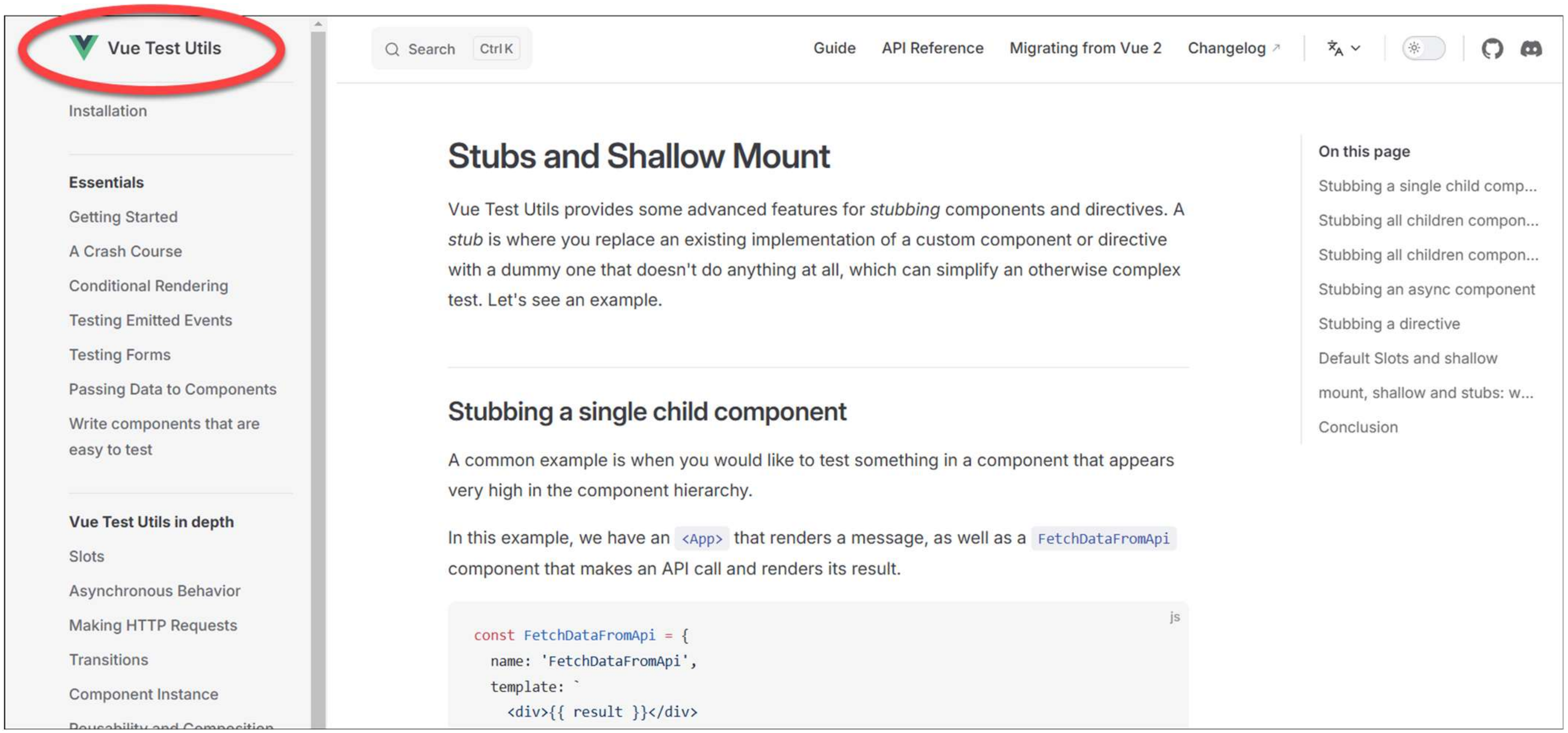
```
import PostCard from "@/components/PostCard.vue";

// …

const wrapper = mount<PostComponentInstance>(PostComponent, {
  global: {
    components: {
      PostCard, // Register <PostCard> in test environment
    },
  },
});
wrapper.vm.posts = mockPosts;
…
```

Solution 3: Exclude `<PostCard>` Rendering

- If you want to test only the data handling logic of `<PostComponent>`
  - No worries about the DOM rendering of child components (like `<PostCard>`)
- Completely stub out the template
- Use `shallow: true` for that
- Previously: `shallowMount()` - this still works, but is a bit outdated

```
const wrapper=mount<PostComponentInstance>
(PostComponent, {
   shallow: true, // completely stub out all child components
});
```

Info on Stubbing and (shallow)mount



Stubs and Shallow Mount

Vue Test Utils provides some advanced features for *stubbing* components and directives. A *stub* is where you replace an existing implementation of a custom component or directive with a dummy one that doesn't do anything at all, which can simplify an otherwise complex test. Let's see an example.

Stubbing a single child component

A common example is when you would like to test something in a component that appears very high in the component hierarchy.

In this example, we have an `<App>` that renders a message, as well as a `FetchDataFromApi` component that makes an API call and renders its result.

```js
const FetchDataFromApi = {
  name: 'FetchDataFromApi',
  template: `
    <div>{{ result }}</div>
```

https://test-utils.vuejs.org/guide/advanced/stubs-shallow-mount

Global stubbing

- If you *always* want to stub out child components, you can tell Vitest that in a `test-setup.ts` file
- Use that file in `vitest.config.ts`.

```
// test-setup.ts
import { config } from '@vue/test-utils';

// Define global stubs for all tests
config.global.stubs = {
    PostCard: true, // Stub PostCard globally
};
```

```
// vitest.config.ts
import { defineConfig } from 'vitest/config';
export default defineConfig({
  …
  test: {
    setupFiles: './test-setup.ts',
  },
});
```

./220-vitest-mocks/test-setup.ts

Mocking: `mockResolvedValueOnce()`

- Helper method, used for mocking responses once
- You'll often see in tests, code like these:

```
(global.fetch as ReturnType<typeof vi.fn>)
  .mockResolvedValueOnce({
      ok: true,
      json: () => Promise.resolve([]),
  });
```

./220-vitest-mocks/pages/posts/index.spec.ts#49

`../220-vitest-mocks`

Breakdown, page 1/2

```
(global.fetch as ReturnType<typeof vi.fn>)
```

- Ensures that TypeScript knows `global.fetch` is being mocked and adheres to the type returned by a `` `vi.fn()` `` (e.g. the Vitest mocking function).
- Effectively, this tells TypeScript the mocked `fetch` function behaves as a `vi.fn`.

```
.mockResolvedValueOnce({…});
```

- Is a helper method provided by Vitest's mocking utilities.
- Sets up the mock function (`global.fetch` in this case) to resolve with the specified value *only once* during the next invocation.
- After this, the mock fetch will revert to its default behavior unless mocked again.

Breakdown 2/2

```
{
    ok: true,
    json: () => Promise.resolve([]),
}
```

- The mock data returned by the `fetch` function for this test invocation:
- `ok: true`: Simulates a successful HTTP response.
- `json: () => Promise.resolve([]):`
- Represents the `json(…)` method of the fetch response, which resolves to an empty array `[]` when called.

It is there as part of a controlled test environment to verify how the component behaves when fetching data.

Complete test

```
it('renders loading state initially', async () => {
    // Set up fetch to return an empty response for this test
    (global.fetch as ReturnType<typeof vi.fn>).mockResolvedValueOnce({
        ok: true,
        json: () => Promise.resolve([]),
    });

    const wrapper = mount(PostComponent);

    // Verify that the loading text is visible
    expect(wrapper.text()).toContain('Loading posts...');
});
```

- The mock ensures that `PostComponent` receives an empty list of posts from the simulated `fetch`.
- This enables the test to focus on verifying whether the component initially shows a "`loading`" state or not, without interference from the actual `fetch` implementation.

`mockRejectedValueOnce()`

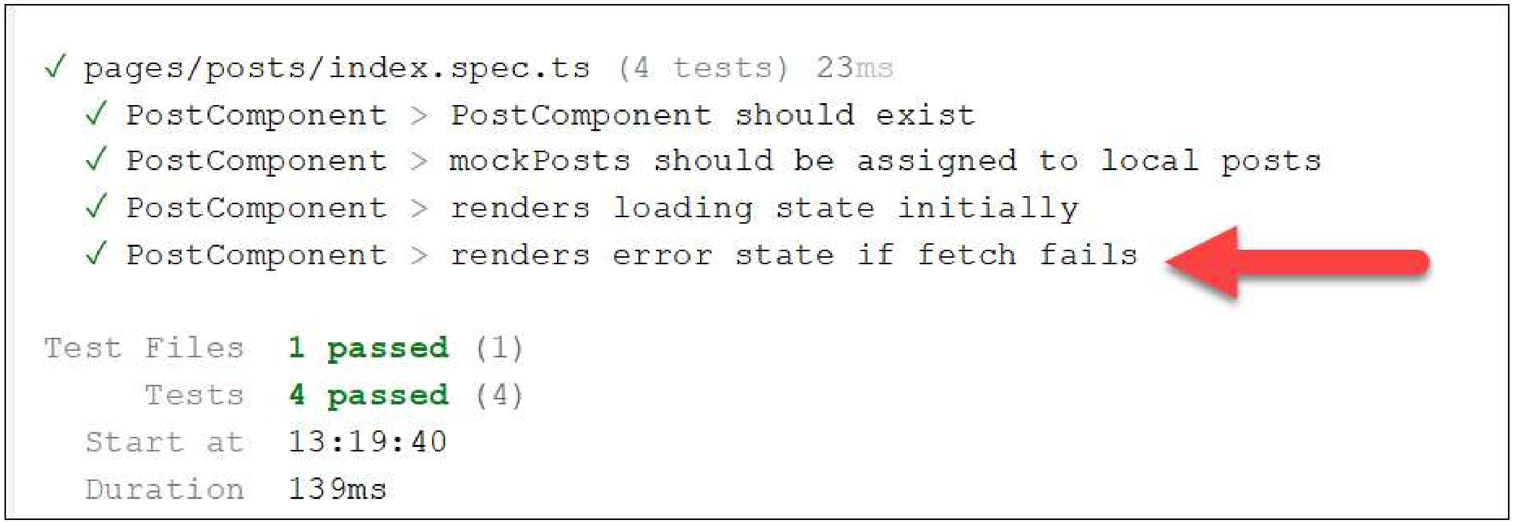
- Handy for testing and simulating errors
- The (mocked) fetch is simulating an error once, using `mockRejectedValueOnce()`

```
it('renders error state if fetch fails', async () => {
    // Mock fetch to reject (simulate an error)
    (global.fetch as ReturnType<typeof vi.fn>)
        .mockRejectedValueOnce(new Error('Failed to fetch posts'));

    const wrapper = mount(PostComponent);

    // Using flushPromises() here.
    await flushPromises();

    // Verify that an error message is displayed
    expect(wrapper.text()).toContain('Failed to fetch posts');
});
```

./220-vitest-mocks/pages/posts/index.spec.ts#64

Why it works…

- Unlike `nextTick`, which waits for DOM updates, `flushPromises()` resolves all outstanding promises in the entire queue.
- This ensures that any asynchronous operations (like fetching data or handling errors) have finished before the test makes its assertion.

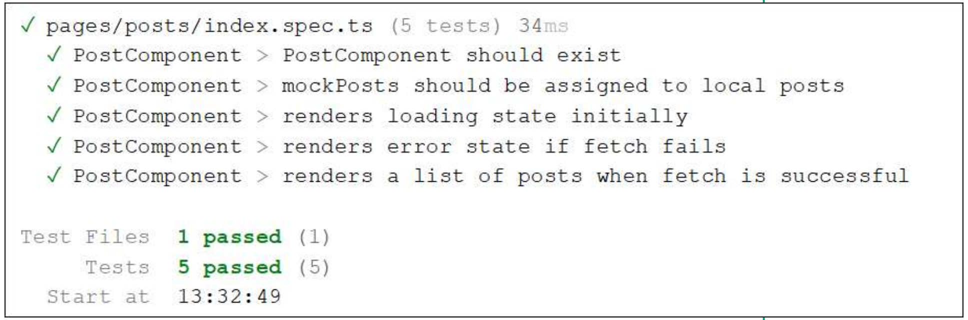


```
✓ pages/posts/index.spec.ts (4 tests) 23ms
  ✓ PostComponent > PostComponent should exist
  ✓ PostComponent > mockPosts should be assigned to local posts
  ✓ PostComponent > renders loading state initially
  ✓ PostComponent > renders error state if fetch fails

Test Files  1 passed (1)
     Tests  4 passed (4)
  Start at  13:19:40
  Duration  139ms
```

Testing if posts are rendered

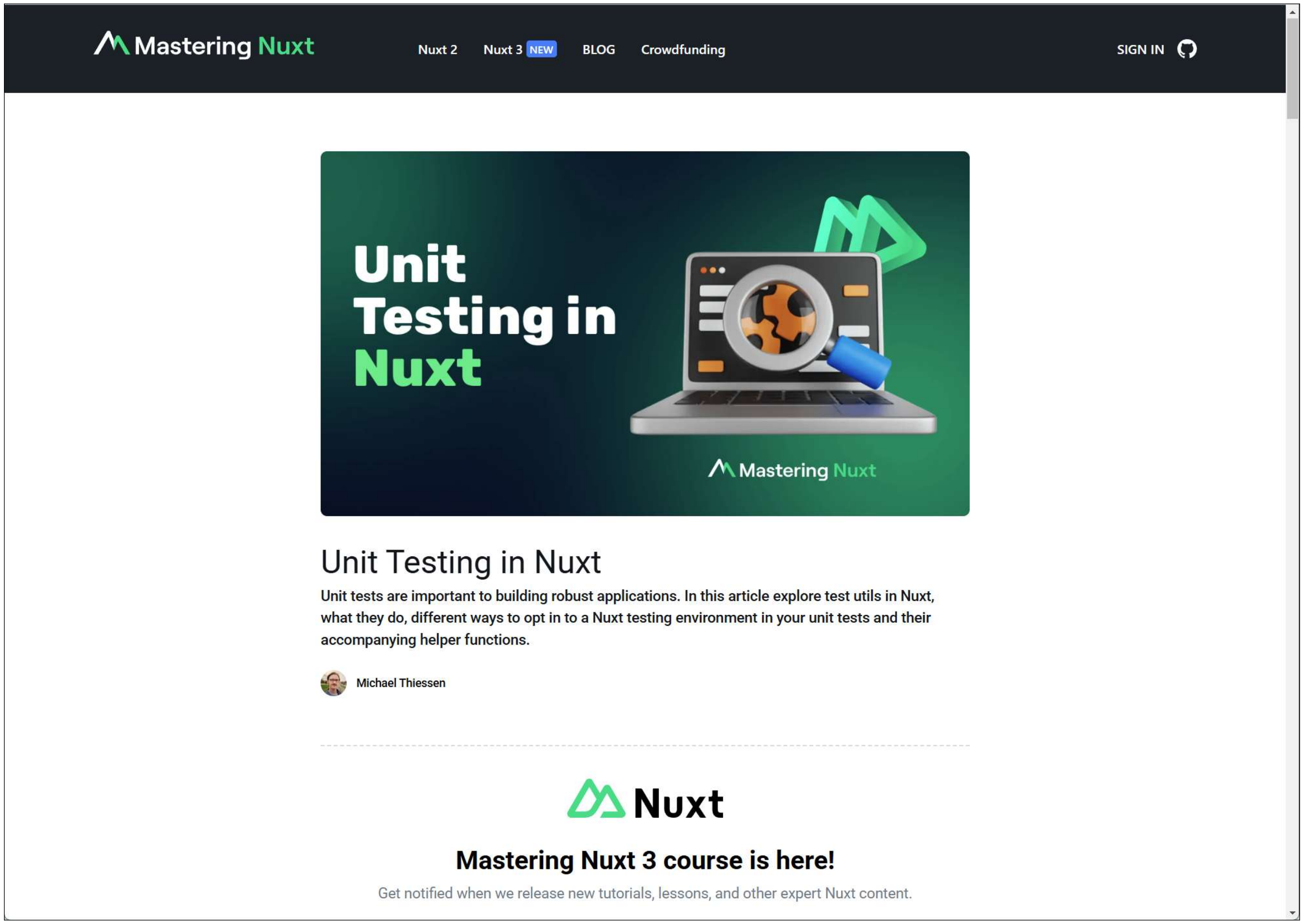
Again, using `mockResolvedValueOnce()`, but this time with the mocked post response

```
it('renders a list of posts when fetch is successful', async () => {
    // This time, resolve the promise using the mocked posts (as defined above).
    (global.fetch as ReturnType<typeof vi.fn>).mockResolvedValueOnce({
        ok: true,
        json: () => Promise.resolve(mockPosts),
    });

    // Create a wrapper
    const wrapper = mount(PostComponent);

    // wait for the promise(s) to resolve
    await flushPromises();

    // Validate that each mock post is in the rendered output
    mockPosts.forEach((post) => {
        expect(wrapper.text()).toContain(post.title);
    });
});
```



```
✓ pages/posts/index.spec.ts (5 tests) 34ms
  ✓ PostComponent > PostComponent should exist
  ✓ PostComponent > mockPosts should be assigned to local posts
  ✓ PostComponent > renders loading state initially
  ✓ PostComponent > renders error state if fetch fails
  ✓ PostComponent > renders a list of posts when fetch is successful

Test Files  1 passed (1)
     Tests  5 passed (5)
  Start at  13:32:49
```

More info



Unit Testing in Nuxt

Unit tests are important to building robust applications. In this article explore test utils in Nuxt, what they do, different ways to opt in to a Nuxt testing environment in your unit tests and their accompanying helper functions.

Michael Thiessen

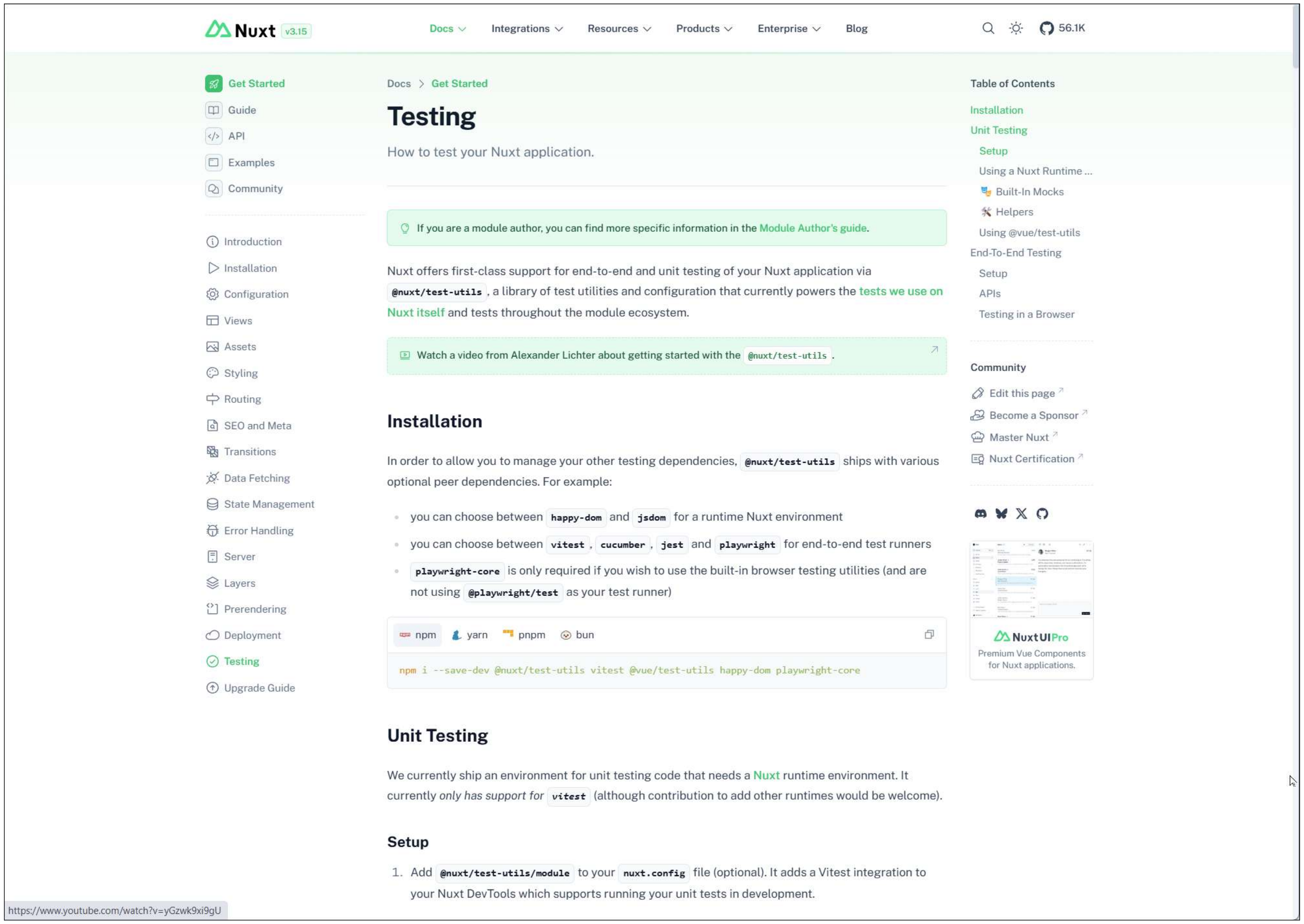
Nuxt

Mastering Nuxt 3 course is here!

Get notified when we release new tutorials, lessons, and other expert Nuxt content.

https://masteringnuxt.com/blog/unit-testing-in-nuxt

More info


Nuxt v3.15 | Docs | Integrations | Resources | Products | Enterprise | Blog | 56.1K

Get Started
Guide
API
Examples
Community

Introduction
Installation
Configuration
Views
Assets
Styling
Routing
SEO and Meta
Transitions
Data Fetching
State Management
Error Handling
Server
Layers
Prerendering
Deployment
Testing
Upgrade Guide

Docs > Get Started

Testing

How to test your Nuxt application.

If you are a module author, you can find more specific information in the Module Author's guide.

Nuxt offers first-class support for end-to-end and unit testing of your Nuxt application via `@nuxt/test-utils`, a library of test utilities and configuration that currently powers the tests we use on Nuxt itself and tests throughout the module ecosystem.

Watch a video from Alexander Lichter about getting started with the `@nuxt/test-utils`.

Installation

In order to allow you to manage your other testing dependencies, `@nuxt/test-utils` ships with various optional peer dependencies. For example:

- you can choose between `happy-dom` and `jsdom` for a runtime Nuxt environment
- you can choose between `vitest`, `cucumber`, `jest` and `playwright` for end-to-end test runners
- `playwright-core` is only required if you wish to use the built-in browser testing utilities (and are not using `@playwright/test` as your test runner)

npm | yarn | pnpm | bun

```
npm i --save-dev @nuxt/test-utils vitest @vue/test-utils happy-dom playwright-core
```

Unit Testing

We currently ship an environment for unit testing code that needs a Nuxt runtime environment. It currently *only has support for* `vitest` (although contribution to add other runtimes would be welcome).

Setup

1. Add `@nuxt/test-utils/module` to your `nuxt.config` file (optional). It adds a Vitest integration to your Nuxt DevTools which supports running your unit tests in development.

Table of Contents
Installation
Unit Testing
Setup
Using a Nuxt Runtime ...
Built-In Mocks
Helpers
Using @vue/test-utils
End-To-End Testing
Setup
APIs
Testing in a Browser

Community
Edit this page
Become a Sponsor
Master Nuxt
Nuxt Certification

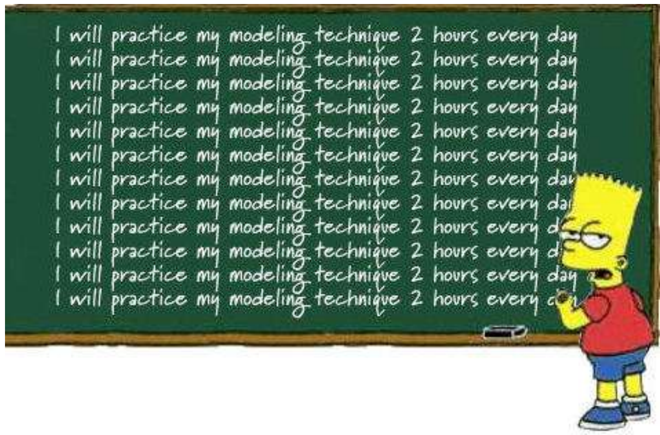
NuxtUIPro
Premium Vue Components for Nuxt applications.

https://www.youtube.com/watch?v=yGzwk9xi9gU

https://nuxt.com/docs/getting-started/testing#setup

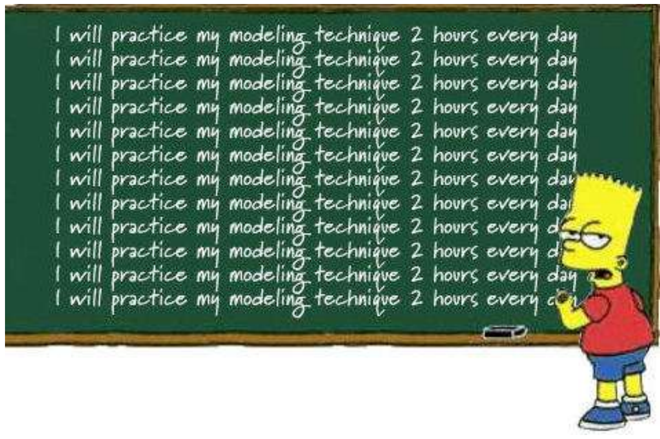
Workshop #1

- Create a very simple `<HelloWorld>` Component.
  - The component should have a `msg` property
  - So if the component is called like `<HelloWorld msg="world">`, the output should render `Hello world`.
- Write a simple unit test for it, that checks if the component works correctly
  - Tip: use `props: {msg:'world'}` as the second argument when creating the wrapper in your test file
- Tip: use AI when necessary, but always verify and test the results!



Optional workshop #2

- The project has a `movies.vue` page.
  - Write a unit test for the movies page, making sure everything works as expected.
- Requirements:
  - The initial structure is rendered
  - searchMovies function is called when the button is clicked
  - searchMovies function is called when the user hits Enter in the search field
  - Renders movies when (mock) data is available
  - Handles empty movie results correctly
- Tips:
  - Use `wrapper.find()`, `wrapper.setData()` and more!



Checkpoint

- You know what Vitest is and what it is used for
- You are able to identify testing files in a solution and sections of the testing files
- You can write unit tests for components
- You know the basics of configuring Vitest
- You know the `mount` function and to use mocked data in the results
- You can test async functions