

Vitest

Version 5.1 · verified 2026-11-02

A gradual walk through Vitest 5 — config and projects, expect, async assertions, mocks, fake timers, React tests, coverage, CI and browser mode.

Contents

Mental model 2

Setup and config 2

Writing tests and matchers 4

Async tests and assertions 6

Mocks and spies 8

Fake timers and dates 10

DOM and React components 12

Coverage, UI and CI 14

Browser mode 15

Further reading 17

Mental model

Vitest runs your tests through the same Vite pipeline as your app, so tests see the same TypeScript transform, plugins and `resolve.alias` paths with no separate Babel or ts-jest setup. Each test file runs in its own isolated module graph inside a worker, and the API (`expect`, `vi`) is Jest-shaped on purpose. Most surprises come from two places: module mocks are hoisted above your imports, and asynchronous assertions only count if you `await` them.

Setup and config

Command	Does
`npm i -D vitest`	Install; needs Vite ≥ 6.4, Node ≥ 22.12
`vitest`	Watch mode locally, single run in CI
`vitest run`	One run, then exit
`vitest -p unit`	Only the `unit` project

Command	Does
`vitest related src/a.ts`	Tests that import that file

Vitest reads `vite.config.*` by default. A `vitest.config.*` file takes priority and **replaces** it, so merge the two with `mergeConfig` if you keep both. Test files match `**/*.{test,spec}.?(c|m)[jt]s?(x)` unless you set `include`. `projects` splits one run into differently configured groups; since Vitest 5, inline projects inherit the root config (plugins, aliases) without `extends: true`.

```
import { defineConfig } from 'vitest/config'

export default defineConfig({
  test: {
    projects: [
      { test: { name: 'unit',
        include: ['src/**/*.test.ts'] } },
      { test: { name: 'dom',
        environment: 'jsdom',
```

```
        include: ['src/**/*.test.tsx'] } },
    ],
  },
})
```

> **Gotcha:** `vitest` without `run` stays in watch mode in any interactive terminal. In a `lint-staged` hook or a script that must exit, call `vitest run` (or `--run`).

Writing tests and matchers

Matcher	Checks
`toBe(x)`	Same value via `Object.is`
`toEqual(x)`	Deep equality, loose on `undefined`
`toStrictEqual(x)`	Deep, plus `undefined` keys and class
`toMatchObject(x)`	Subset of properties
`toThrow('msg')`	Throws; string is a substring match

Matcher	Checks
`toHaveBeenCalledWith(...)`	A spy received these args

Import `describe`, `test` (alias `it`) and `expect` from `vitest`; globals are off by default. Use `toBe` for primitives and `toEqual` for objects. `test.each` and `test.for` run one body over a table of inputs, and each row becomes its own named test in the report.

```ts
import { describe, expect, test } from 'vitest'

const slug = (s: string) =>
  s.trim().toLowerCase().replace(/\s+/g, '-')

describe('slug', () => {
  test.each([
    ['Hello World', 'hello-world'],
    ['  Trim me ', 'trim-me'],
  ])('%s -> %s', (input, out) => {
    expect(slug(input)).toBe(out)
```

```
  })
})
```

> **Gotcha:**
> `expect({ a: 1, b: undefined }).toEqual({ a: 1 })` passes. When a missing key versus an `undefined` key matters (API payloads, serialised state), use `toStrictEqual`.

Async tests and assertions

Pattern	Use for
`await expect(p).resolves.toBe(x)`	Promise fulfils with `x`
`await expect(p).rejects.toThrow()`	Promise rejects
`await expect.poll(fn).toBe(x)`	Retry until true or timeout
`{ timeout: 10_000 }`	Per-test timeout (default 5 s)

Return or `await` everything asynchronous in the test body; Vitest waits for the returned promise and nothing else. `expect.poll` re-runs the callback until the matcher passes, which suits state that settles eventually, such as a queue drained by a background worker.

```
import { expect, test } from 'vitest'

const load = async (id: number) => {
  if (id < 0) throw new Error('bad id')
  return { id }
}

test('load', async () => {
  const ok = load(1)
  await expect(ok).resolves.toEqual({ id: 1 })
  await expect(load(-1)).rejects.toThrow('bad')
})
```

> **Gotcha:** Since Vitest 5, an un-awaited `resolves` / `rejects` fails the test. Upgrading from 4 surfaces these as new failures — they were assertions that never ran where they were written.

Mocks and spies

API	Does
`vi.fn(impl?)`	Standalone mock function
`vi.spyOn(obj, 'm')`	Wrap a real method, keep it callable
`vi.mock(path, factory?)`	Replace a module for this file
`vi.mocked(fn)`	Type a mocked import as a mock
`vi.restoreAllMocks()`	Put `spyOn` originals back

`vi.mock` is hoisted above every `import`, so the module is replaced before your code loads it. Without a factory, Vitest automocks every export; `{ spy: true }` keeps the

real implementations but records calls. Variables used inside a factory must come from `vi.hoisted`, because ordinary top-level code has not run yet.

```
import { expect, test, vi } from 'vitest'
import { hostname } from 'node:os'

vi.mock('node:os', () => ({
  hostname: vi.fn(() => 'ci-box'),
}))

test('mocked module', () => {
  expect(hostname()).toBe('ci-box')
  expect(hostname).toHaveBeenCalledTimes(1)
})
```

> **Gotcha:** Vitest 5 throws when `vi.mock` sits inside a `describe` or `test` callback — it was always hoisted to the top, so it never ran where it was written. For a per-test module mock, use `vi.doMock` and a dynamic `import()`.

Fake timers and dates

API	Does
`vi.useFakeTimers()`	Fake timers, `Date` (and `Temporal` in 5)
`vi.advanceTimersByTime(ms)`	Run timers due within `ms`
`vi.advanceTimersByTimeAsync(ms)`	Same, flushing promises between
`vi.setSystemTime(date)`	Pin “now”
`vi.useRealTimers()`	Restore

Fake timers make debounce, retry and expiry logic deterministic and instant. Advance time explicitly

rather than waiting on it. Use the `Async` variants when the code under test awaits between timers, otherwise the promise callbacks never get a turn.

```
import { expect, test, vi } from 'vitest'

test('retry fires after 1 s', () => {
  vi.useFakeTimers()
  vi.setSystemTime(new Date('2026-01-01'))
  const fn = vi.fn()
  setTimeout(fn, 1000)
  vi.advanceTimersByTime(999)
  expect(fn).not.toHaveBeenCalled()
  vi.advanceTimersByTime(1)
  expect(fn).toHaveBeenCalledOnce()
  expect(new Date().getFullYear()).toBe(2026)
  vi.useRealTimers()
})
```

> **Note:** `clearMocks` defaults to `true` in Vitest 5: call history is wiped before every test, implementations stay. `restoreMocks` and `mockReset` still default to `false`.

DOM and React components

Setting	Effect
`environment: 'jsdom'`	Simulated DOM (or `'happy-dom'`)
`setupFiles: ['./test/setup.ts']`	Runs before each test file
`globals: true`	`describe`, `afterEach`… as globals
`// @vitest-environment jsdom`	Per-file override, top of file

The default environment is `node`. Install `jsdom` or `happy-dom` and switch per project or per file. React Testing Library renders into that simulated DOM; add

`@testing-library/jest-dom/vitest` in a setup file for matchers such as `toBeInTheDocument`.

```
import { cleanup } from '@testing-library/react'
import '@testing-library/jest-dom/vitest'
import { afterEach } from 'vitest'

// test/setup.ts — explicit cleanup, needed
// when `globals` is off
afterEach(() => { cleanup() })
```

> **Gotcha:** Testing Library's automatic cleanup hooks into a global `afterEach`. With `globals: false` (the default) nothing unmounts between tests, and the next test finds the previous test's DOM — duplicate-element errors that disappear when run alone.

Coverage, UI and CI

Flag	Does
`--coverage`	Collect coverage (`v8` provider default)
`--shard=1/3`	Run one third of the files
`--reporter=junit`	Machine-readable output
`--ui`	Browser dashboard (`@vitest/ui`)
`--last-failed`	Re-run last run's failures (5.1)
`--repeats=20`	Repeat tests to hunt flakes (5.0)

Coverage needs `@vitest/coverage-v8` (or `-istanbul`). By default the report lists only files your tests imported; set `coverage.include` so untested files show up at 0 %. Thresholds fail the run when coverage drops: a positive number is a minimum percentage, a negative one a maximum count of uncovered items. Vitest 5 writes reports and artifacts under one `.vitest/` directory — add it to `.gitignore` .

```ts
import { defineConfig } from 'vitest/config'

export default defineConfig({
  test: {
    coverage: {
      include: ['src/**/*.{ts,tsx}'],
      thresholds: { lines: 80, functions: -5 },
    },
  },
})
```

> **Tip:** Vitest UI now requires a token. Open the exact URL Vitest prints (`/__vitest__/?token=…`); the bare path shows an error until that browser is authenticated.

Browser mode

Piece	Value
Status	Stable since Vitest 4.0

Piece	Value
Setup	`npx vitest init browser`
Provider	`@vitest/browser-playwright`
Assertions	`await expect.element(loc)`
Trace view	On failure by default (5.1)

Browser mode runs component tests in a real browser engine instead of jsdom, through a provider package. Locators come from `vitest/browser` and retry until they match, and since Vitest 5 they match text exactly by default. Use it when jsdom's gaps (layout, real events, CSS) make tests lie; keep plain logic in the cheaper `node` environment.

```
import { defineConfig } from 'vitest/config'
import { playwright }
  from '@vitest/browser-playwright'

export default defineConfig({
  test: {
```

```
    browser: {
      enabled: true,
      provider: playwright(),
      instances: [{ browser: 'chromium' }],
    },
  },
})
```

> **Note:** Browser mode is not end-to-end testing. It mounts components in a page Vitest serves; for whole-app flows against a running server, use Playwright.

Further reading

- Vitest — Getting Started
- Vitest — Mocking guide
- Vitest — Test projects
- Vitest — Browser Mode
- Vitest — Migration guide

- Vitest 5.0 release announcement