VITEST Docs - English

Table of contents

- Api - Index 3
- Config - Index 57
- Guide - Cli 75
- Guide - Comparisons 78
- Guide - Coverage 80
- Guide - Debugging 82
- Guide - Environment 84
- Guide - Extending matchers 86
- Guide - Features 88
- Guide - Filtering 92
- Guide - Ide 94
- Guide - In source 98
- Guide - Index 100
- Guide - Migration 103
- Guide - Mocking 105
- Guide - Snapshot 115
- Guide - Test context 119
- Guide - Testing types 121
- Guide - Ui 123
- Guide - Using plugins 125
- Guide - Why 126

API Reference

The following types are used in the type signatures below

```ts
type Awaitable<T> = T | PromiseLike<T>
type TestFunction = () => Awaitable<void>

interface TestOptions {
  timeout?: number
  retry?: number
}
```

When a test function returns a promise, the runner will wait until it is resolved to collect async expectations. If the promise is rejected, the test will fail.

> In Jest, `TestFunction` can also be of type `(done: DoneCallback) => void`. If this form is used, the test will not be concluded until `done` is called. You can achieve the same using an `async` function, see the [Migration guide Done Callback section](../guide/migration#done-callback).

test

- **Type:** `(name: string, fn: TestFunction, timeout?: number | TestOptions) => void`
- **Alias:** `it`

  `test` defines a set of related expectations. It receives the test name and a function that holds the expectations to test.

  Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before terminating. The default is 5 seconds, and can be configured globally with testTimeout

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

test('should work as expected', () => {
  expect(Math.sqrt(4)).toBe(2)
})
```

test.skip

- **Type:** `(name: string, fn: TestFunction, timeout?: number | TestOptions) => void`
- **Alias:** `it.skip`

  If you want to skip running certain tests, but you don't want to delete the code due to any reason, you can use `test.skip` to avoid running them.

```ts
import { assert, test } from 'vitest'

test.skip('skipped test', () => {
```

```
  // Test skipped, no error
  assert.equal(Math.sqrt(4), 3)
})
```

test.skipIf

- **Type:** `(condition: any) => Test`
- **Alias:** `it.skipIf`

In some cases you might run tests multiple times with different environments, and some of the tests might be environment-specific. Instead of wrapping the test code with `if`, you can use `test.skipIf` to skip the test whenever the condition is truthy.

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

const isDev = process.env.NODE_ENV === 'development'

test.skipIf(isDev)('prod only test', () => {
  // this test only runs in production
})
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.runIf

- **Type:** `(condition: any) => Test`
- **Alias:** `it.runIf`

Opposite of test.skipIf.

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

const isDev = process.env.NODE_ENV === 'development'

test.runIf(isDev)('dev only test', () => {
  // this test only runs in development
})
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.only

- **Type:** `(name: string, fn: TestFunction, timeout?: number) => void`
- **Alias:** `it.only`

Use `test.only` to only run certain tests in a given suite. This is useful when debugging.

Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before terminating. The default is 5 seconds, and can be configured globally with testTimeout.

```ts
import { assert, test } from 'vitest'

test.only('test', () => {
  // Only this test (and others marked with only) are run
  assert.equal(Math.sqrt(4), 2)
})
```

Sometimes it is very useful to run `only` tests in a certain file, ignoring all other tests from the whole test suite, which pollute the output.

In order to do that run `vitest` with specific file containing the tests in question.

```
# vitest interesting.test.ts
```

test.concurrent

- **Type:** `(name: string, fn: TestFunction, timeout?: number) => void`
- **Alias:** `it.concurrent`

`test.concurrent` marks consecutive tests to be run in parallel. It receives the test name, an async function with the tests to collect, and an optional timeout (in milliseconds).

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

// The two tests marked with concurrent will be run in parallel
describe('suite', () => {
  test('serial test', async () => { /* ... */ })
  test.concurrent('concurrent test 1', async () => { /* ... */ })
  test.concurrent('concurrent test 2', async () => { /* ... */ })
})
```

`test.skip`, `test.only`, and `test.todo` works with concurrent tests. All the following combinations are valid:

```ts
test.concurrent(/* ... */)
test.skip.concurrent(/* ... */) // or test.concurrent.skip(/* ... */)
test.only.concurrent(/* ... */) // or test.concurrent.only(/* ... */)
test.todo.concurrent(/* ... */) // or test.concurrent.todo(/* ... */)
```

When using Snapshots with async concurrent tests, due to the limitation of JavaScript, you need to use the `expect` from the Test Context to ensure the right test is being detected.

```ts
test.concurrent('test 1', async ({ expect }) => {
  expect(foo).toMatchSnapshot()
})
test.concurrent('test 2', async ({ expect }) => {
  expect(foo).toMatchSnapshot()
})
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.todo

- **Type:** `(name: string) => void`
- **Alias:** `it.todo`

Use `test.todo` to stub tests to be implemented later. An entry will be shown in the report for the tests so you know how many tests you still need to implement.

```
// An entry will be shown in the report for this test
test.todo('unimplemented test')
```

test.fails

- **Type:** `(name: string, fn: TestFunction, timeout?: number) => void`
- **Alias:** `it.fails`

Use `test.fails` to indicate that an assertion will fail explicitly.

```
import { expect, test } from 'vitest'
const myAsyncFunc = () => new Promise(resolve => resolve(1))
test.fails('fail test', async () => {
  await expect(myAsyncFunc()).rejects.toBe(1)
})
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

test.each

- **Type:** `(cases: ReadonlyArray<T>, ...args: any[]) => void`
- **Alias:** `it.each`

Use `test.each` when you need to run the same test with different variables. You can inject parameters with printf formatting in the test name in the order of the test function parameters.

- `%s` : string
- `%d` : number
- `%i` : integer
- `%f` : floating point value
- `%j` : json
- `%o` : object
- `%#` : index of the test case
- `%%` : single percent sign ('%')

```
test.each([
  [1, 1, 2],
  [1, 2, 3],
  [2, 1, 3],
])('add(%i, %i) -> %i', (a, b, expected) => {
  expect(a + b).toBe(expected)
```

```
})

// this will return
// ✓ add(1, 1) -> 2
// ✓ add(1, 2) -> 3
// ✓ add(2, 1) -> 3
```

You can also access object properties with `$` prefix, if you are using objects as arguments:

```
test.each([
  { a: 1, b: 1, expected: 2 },
  { a: 1, b: 2, expected: 3 },
  { a: 2, b: 1, expected: 3 },
])('add($a, $b) -> $expected', ({ a, b, expected }) => {
  expect(a + b).toBe(expected)
})

// this will return
// ✓ add(1, 1) -> 2
// ✓ add(1, 2) -> 3
// ✓ add(2, 1) -> 3
```

Starting from Vitest 0.25.3, you can also use template string table.

- First row should be column names, separated by `|`;
- One or more subsequent rows of data supplied as template literal expressions using `${value}` syntax.

```
test.each`
  a               | b      | expected
  ${1}            | ${1}   | ${2}
  ${'a'}          | ${'b'} | ${'ab'}
  ${[]}           | ${'b'} | ${'b'}
  ${{}}           | ${'b'} | ${'[object Object]b'}
  ${{ asd: 1 }}   | ${'b'} | ${'[object Object]b'}
`('returns $expected when $a is added $b', ({ a, b, expected }) => {
  expect(a + b).toBe(expected)
})
```

If you want to have access to `TestContext`, use `describe.each` with a single test.

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

bench

- **Type:** `(name: string, fn: BenchFunction, options?: BenchOptions) => void`

`bench` defines a benchmark. In Vitest terms benchmark is a function that defines a series of operations. Vitest runs this function multiple times to display different performance results.

Vitest uses `tinybench` library under the hood, inheriting all its options that can be used as a third argument.

```ts
import { bench } from 'vitest'

bench('normal sorting', () => {
  const x = [1, 5, 4, 2, 3]
  x.sort((a, b) => {
    return a - b
  })
}, { time: 1000 })
```

```ts
export interface Options {
  /**
   * time needed for running a benchmark task (milliseconds)
   * @default 500
   */
  time?: number

  /**
   * number of times that a task should run if even the time option is finished
   * @default 10
   */
  iterations?: number

  /**
   * function to get the current timestamp in milliseconds
   */
  now?: () => number

  /**
   * An AbortSignal for aborting the benchmark
   */
  signal?: AbortSignal

  /**
   * warmup time (milliseconds)
   * @default 100ms
   */
  warmupTime?: number

  /**
   * warmup iterations
   * @default 5
   */
  warmupIterations?: number

  /**
   * setup function to run before each benchmark task (cycle)
   */
  setup?: Hook

  /**
   * teardown function to run after each benchmark task (cycle)
   */
  teardown?: Hook
}
```

bench.skip

- **Type:** `(name: string, fn: BenchFunction, options?: BenchOptions) => void`

You can use `bench.skip` syntax to skip running certain benchmarks.

```ts
import { bench } from 'vitest'

bench.skip('normal sorting', () => {
  const x = [1, 5, 4, 2, 3]
  x.sort((a, b) => {
    return a - b
  })
})
```

bench.only

- **Type:** `(name: string, fn: BenchFunction, options?: BenchOptions) => void`

Use `bench.only` to only run certain benchmarks in a given suite. This is useful when debugging.

```ts
import { bench } from 'vitest'

bench.only('normal sorting', () => {
  const x = [1, 5, 4, 2, 3]
  x.sort((a, b) => {
    return a - b
  })
})
```

bench.todo

- **Type:** `(name: string) => void`

Use `bench.todo` to stub benchmarks to be implemented later.

```ts
import { bench } from 'vitest'

bench.todo('unimplemented test')
```

describe

When you use `test` or `bench` in the top level of file, they are collected as part of the implicit suite for it. Using `describe` you can define a new suite in the current context, as a set of related tests or benchmarks and other nested suites. A suite lets you organize your tests and benchmarks so reports are more clear.

```ts
// basic.spec.ts
// organizing tests

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

const person = {
  isActive: true,
  age: 32,
}

describe('person', () => {
  test('person is defined', () => {
    expect(person).toBeDefined()
  })

  test('is active', () => {
    expect(person.isActive).toBeTruthy()
```

```
  })

  test('age limit', () => {
    expect(person.age).toBeLessThanOrEqual(32)
  })
})
```

```
// basic.bench.ts
// organizing benchmarks

import { bench, describe } from 'vitest'

describe('sort', () => {
  bench('normal', () => {
    const x = [1, 5, 4, 2, 3]
    x.sort((a, b) => {
      return a - b
    })
  })

  bench('reverse', () => {
    const x = [1, 5, 4, 2, 3]
    x.reverse().sort((a, b) => {
      return a - b
    })
  })
})
```

You can also nest describe blocks if you have a hierarchy of tests or benchmarks:

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

const numberToCurrency = (value) => {
  if (typeof value !== 'number')
    throw new Error('Value must be a number')

  return value.toFixed(2).toString().replace(/\B(?=(\d{3})+(?!\d))/g, ',')
}

describe('numberToCurrency', () => {
  describe('given an invalid number', () => {
    test('composed of non-numbers to throw error', () => {
      expect(() => numberToCurrency('abc')).toThrowError()
    })
  })

  describe('given a valid number', () => {
    test('returns the correct currency format', () => {
      expect(numberToCurrency(10000)).toBe('10,000.00')
    })
  })
})
```

describe.skip

- **Type:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

  Use `describe.skip` in a suite to avoid running a particular describe block.

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

describe.skip('skipped suite', () => {
  test('sqrt', () => {
    // Suite skipped, no error
    assert.equal(Math.sqrt(4), 3)
  })
})
```

describe.only

- **Type:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

  Use `describe.only` to only run certain suites

```ts
// Only this suite (and others marked with only) are run
describe.only('suite', () => {
  test('sqrt', () => {
    assert.equal(Math.sqrt(4), 3)
  })
})

describe('other suite', () => {
  // ... will be skipped
})
```

  Sometimes it is very useful to run `only` tests in a certain file, ignoring all other tests from the whole test suite, which pollute the output.

  In order to do that run `vitest` with specific file containing the tests in question.

```ts
# vitest interesting.test.ts
```

describe.concurrent

- **Type:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

  `describe.concurrent` in a suite marks every tests as concurrent

```ts
// All tests within this suite will be run in parallel
describe.concurrent('suite', () => {
  test('concurrent test 1', async () => { /* ... */ })
  test('concurrent test 2', async () => { /* ... */ })
  test.concurrent('concurrent test 3', async () => { /* ... */ })
})
```

  `.skip`, `.only`, and `.todo` works with concurrent suites. All the following combinations are valid:

```ts
describe.concurrent(/* ... */)
describe.skip.concurrent(/* ... */) // or describe.concurrent.skip(/* ... */)
describe.only.concurrent(/* ... */) // or describe.concurrent.only(/* ... */)
describe.todo.concurrent(/* ... */) // or describe.concurrent.todo(/* ... */)
```

  You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

describe.shuffle

- **Type:** `(name: string, fn: TestFunction, options?: number | TestOptions) => void`

  Vitest provides a way to run all tests in random order via CLI flag `--sequence.shuffle` or config option `sequence.shuffle`, but if you want to have only part of your test suite to run tests in random order, you can mark it with this flag.

```ts
describe.shuffle('suite', () => {
  test('random test 1', async () => { /* ... */ })
  test('random test 2', async () => { /* ... */ })
  test('random test 3', async () => { /* ... */ })
})
// order depends on sequence.seed option in config (Date.now() by default)
```

`.skip`, `.only`, and `.todo` works with random suites.

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

describe.todo

- **Type:** `(name: string) => void`

  Use `describe.todo` to stub suites to be implemented later. An entry will be shown in the report for the tests so you know how many tests you still need to implement.

```ts
// An entry will be shown in the report for this suite
describe.todo('unimplemented suite')
```

describe.each

- **Type:** `(cases: ReadonlyArray<T>, ...args: any[]): (name: string, fn: (...args: T[]) => void, options?: number | TestOptions) => void`

  Use `describe.each` if you have more than one test that depends on the same data.

```ts
describe.each([
  { a: 1, b: 1, expected: 2 },
  { a: 1, b: 2, expected: 3 },
  { a: 2, b: 1, expected: 3 },
])('describe object add($a, $b)', ({ a, b, expected }) => {
  test(`returns ${expected}`, () => {
    expect(a + b).toBe(expected)
  })

  test(`returned value not be greater than ${expected}`, () => {
    expect(a + b).not.toBeGreaterThan(expected)
  })

  test(`returned value not be less than ${expected}`, () => {
    expect(a + b).not.toBeLessThan(expected)
  })
})
```

Starting from Vitest 0.25.3, you can also use template string table.

- First row should be column names, separated by `|`;
- One or more subsequent rows of data supplied as template literal expressions using `${value}` syntax.

```ts
describe.each`
  a               | b      | expected
  ${1}            | ${1}   | ${2}
  ${'a'}          | ${'b'} | ${'ab'}
  ${[]}           | ${'b'} | ${'b'}
  ${{}}           | ${'b'} | ${'[object Object]b'}
  ${{ asd: 1 }}   | ${'b'} | ${'[object Object]b'}
`('describe template string add($a, $b)', ({ a, b, expected }) => {
  test(`returns ${expected}`, () => {
    expect(a + b).toBe(expected)
  })
})
```

You cannot use this syntax, when using Vitest as [type checker](/guide/testing-types).

expect

- **Type:** `ExpectStatic & (actual: any) => Assertions`

`expect` is used to create assertions. In this context `assertions` are functions that can be called to assert a statement. Vitest provides `chai` assertions by default and also `Jest` compatible assertions build on top of `chai`.

For example, this code asserts that an `input` value is equal to `2`. If it's not, assertion will throw an error, and the test will fail.

```ts
import { expect } from 'vitest'

const input = Math.sqrt(4)

expect(input).to.equal(2) // chai API
expect(input).toBe(2) // jest API
```

Technically this example doesn't use `test` function, so in the console you will see Nodejs error instead of Vitest output. To learn more about `test`, please read next chapter.

Also, `expect` can be used statically to access matchers functions, described later, and more.

`expect` has no effect on testing types, if expression doesn't have a type error. If you want to use Vitest as [type checker](/guide/testing-types), use [`expectTypeOf`](#expecttypeof) or [`assertType`](#asserttype).

not

Using `not` will negate the assertion. For example, this code asserts that an `input` value is not equal to `2`. If it's equal, assertion will throw an error, and the test will fail.

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

const input = Math.sqrt(16)

expect(input).not.to.equal(2) // chai API
expect(input).not.toBe(2) // jest API
```

toBe

- **Type:** `(value: any) => Awaitable<void>`

  `toBe` can be used to assert if primitives are equal or that objects share the same reference. It is equivalent of calling `expect(Object.is(3, 3)).toBe(true)`. If the objects are not the same, but you want check if their structures are identical, you can use `toEqual`.

  For example, the code below checks if the trader has 13 apples.

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

const stock = {
  type: 'apples',
  count: 13,
}

test('stock has 13 apples', () => {
  expect(stock.type).toBe('apples')
  expect(stock.count).toBe(13)
})

test('stocks are the same', () => {
  const refStock = stock // same reference

  expect(stock).toBe(refStock)
})
```

  Try not to use `toBe` with floating-point numbers. Since JavaScript rounds them, `0.1 + 0.2` is not strictly `0.3`. To reliably assert floating-point numbers, use `toBeCloseTo` assertion.

toBeCloseTo

- **Type:** `(value: number, numDigits?: number) => Awaitable<void>`

  Use `toBeCloseTo` to compare floating-point numbers. The optional `numDigits` argument limits the number of digits to check *after* the decimal point. For example:

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

test.fails('decimals are not equal in javascript', () => {
  expect(0.2 + 0.1).toBe(0.3) // 0.2 + 0.1 is 0.30000000000000004
})
```

```
test('decimals are rounded to 5 after the point', () => {
  // 0.2 + 0.1 is 0.30000 | "000000000004" removed
  expect(0.2 + 0.1).toBeCloseTo(0.3, 5)
  // nothing from 0.3000000000000004 is removed
  expect(0.2 + 0.1).not.toBeCloseTo(0.3, 50)
})
```

toBeDefined

- **Type:** `() => Awaitable<void>`

`toBeDefined` asserts that the value is not equal to `undefined`. Useful use case would be to check if function *returned* anything.

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

const getApples = () => 3

test('function returned something', () => {
  expect(getApples()).toBeDefined()
})
```

toBeUndefined

- **Type:** `() => Awaitable<void>`

Opposite of `toBeDefined`, `toBeUndefined` asserts that the value *is* equal to `undefined`. Useful use case would be to check if function hasn't *returned* anything.

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

function getApplesFromStock(stock) {
  if (stock === 'Bill')
    return 13
}

test('mary doesn\'t have a stock', () => {
  expect(getApplesFromStock('Mary')).toBeUndefined()
})
```

toBeTruthy

- **Type:** `() => Awaitable<void>`

`toBeTruthy` asserts that the value is true, when converted to boolean. Useful if you don't care for the value, but just want to know it can be converted to `true`.

For example having this code you don't care for the return value of `stocks.getInfo` - it maybe complex object, a string or anything else. The code will still work.

```
import { Stocks } from './stocks'
const stocks = new Stocks()
stocks.sync('Bill')
if (stocks.getInfo('Bill'))
  stocks.sell('apples', 'Bill')
```

So if you want to test that `stocks.getInfo` will be truthy, you could write:

```
import { expect, test } from 'vitest'
import { Stocks } from './stocks'
const stocks = new Stocks()

test('if we know Bill stock, sell apples to him', () => {
  stocks.sync('Bill')
  expect(stocks.getInfo('Bill')).toBeTruthy()
})
```

Everything in JavaScript is truthy, except `false`, `0`, `''`, `null`, `undefined`, and `NaN`.

toBeFalsy

- **Type:** `() => Awaitable<void>`

`toBeFalsy` asserts that the value is false, when converted to boolean. Useful if you don't care for the value, but just want to know it can be converted to `false`.

For example having this code you don't care for the return value of `stocks.stockFailed` - it may return any falsy value, but the code will still work.

```
import { Stocks } from './stocks'
const stocks = new Stocks()
stocks.sync('Bill')
if (!stocks.stockFailed('Bill'))
  stocks.sell('apples', 'Bill')
```

So if you want to test that `stocks.stockFailed` will be falsy, you could write:

```
import { expect, test } from 'vitest'
import { Stocks } from './stocks'
const stocks = new Stocks()

test('if Bill stock hasn\'t failed, sell apples to him', () => {
  stocks.syncStocks('Bill')
  expect(stocks.stockFailed('Bill')).toBeFalsy()
})
```

Everything in JavaScript is truthy, except `false`, `0`, `''`, `null`, `undefined`, and `NaN`.

toBeNull

- **Type:** `() => Awaitable<void>`

`toBeNull` simply asserts if something is `null`. Alias for `.toBe(null)`.

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

function apples() {
  return null
}
```

```
test('we don\'t have apples', () => {
  expect(apples()).toBeNull()
})
```

toBeNaN

- **Type:** `() => Awaitable<void>`

`toBeNaN` simply asserts if something is `NaN`. Alias for `.toBe(NaN)`.

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

let i = 0

function getApplesCount() {
  i++
  return i > 1 ? NaN : i
}

test('getApplesCount has some unusual side effects...', () => {
  expect(getApplesCount()).not.toBeNaN()
  expect(getApplesCount()).toBeNaN()
})
```

toBeTypeOf

- **Type:** `(c: 'bigint' | 'boolean' | 'function' | 'number' | 'object' | 'string' | 'symbol' | 'undefined') => Awaitable<void>`

`toBeTypeOf` asserts if an actual value is of type of received type.

```
import { expect, test } from 'vitest'
const actual = 'stock'

test('stock is type of string', () => {
  expect(actual).toBeTypeOf('string')
})
```

toBeInstanceOf

- **Type:** `(c: any) => Awaitable<void>`

`toBeInstanceOf` asserts if an actual value is instance of received class.

```
import { expect, test } from 'vitest'
import { Stocks } from './stocks'
const stocks = new Stocks()

test('stocks are instance of Stocks', () => {
  expect(stocks).toBeInstanceOf(Stocks)
})
```

toBeGreaterThan

- **Type:** `(n: number | bigint) => Awaitable<void>`

`toBeGreaterThan` asserts if actual value is greater than received one. Equal values will fail the test.

```
import { expect, test } from 'vitest'
import { getApples } from './stock'

test('have more then 10 apples', () => {
  expect(getApples()).toBeGreaterThan(10)
})
```

toBeGreaterThanOrEqual

- **Type:** `(n: number | bigint) => Awaitable<void>`

`toBeGreaterThanOrEqual` asserts if actual value is greater than received one or equal to it.

```
import { expect, test } from 'vitest'
import { getApples } from './stock'

test('have 11 apples or more', () => {
  expect(getApples()).toBeGreaterThanOrEqual(11)
})
```

toBeLessThan

- **Type:** `(n: number | bigint) => Awaitable<void>`

`toBeLessThan` asserts if actual value is less than received one. Equal values will fail the test.

```
import { expect, test } from 'vitest'
import { getApples } from './stock'

test('have less then 20 apples', () => {
  expect(getApples()).toBeLessThan(20)
})
```

toBeLessThanOrEqual

- **Type:** `(n: number | bigint) => Awaitable<void>`

`toBeLessThanOrEqual` asserts if actual value is less than received one or equal to it.

```
import { expect, test } from 'vitest'
import { getApples } from './stock'

test('have 11 apples or less', () => {
  expect(getApples()).toBeLessThanOrEqual(11)
})
```

toEqual

- **Type:** `(received: any) => Awaitable<void>`

`toEqual` asserts if actual value is equal to received one or has the same structure, if it is an object (compares them recursively). You can see the difference between `toEqual` and `toBe` in this example:

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

const stockBill = {
  type: 'apples',
  count: 13,
}

const stockMary = {
  type: 'apples',
  count: 13,
}

test('stocks have the same properties', () => {
  expect(stockBill).toEqual(stockMary)
})

test('stocks are not the same', () => {
  expect(stockBill).not.toBe(stockMary)
})
```

A _deep equality_ will not be performed for `Error` objects. To test if something was thrown, use [`toThrowError`](#tothrowerror) assertion.

toStrictEqual

- **Type:** `(received: any) => Awaitable<void>`

  `toStrictEqual` asserts if actual value is equal to received one or has the same structure, if it is an object (compares them recursively), and of the same type.

  Differences from `.toEqual`:

  - Keys with `undefined` properties are checked. e.g. `{a: undefined, b: 2}` does not match `{b: 2}` when using `.toStrictEqual`.
  - Array sparseness is checked. e.g. `[, 1]` does not match `[undefined, 1]` when using `.toStrictEqual`.
  - Object types are checked to be equal. e.g. A class instance with fields `a` and `b` will not equal a literal object with fields `a` and `b`.

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

class Stock {
  constructor(type) {
    this.type = type
  }
}

test('structurally the same, but semantically different', () => {
  expect(new Stock('apples')).toEqual({ type: 'apples' })
  expect(new Stock('apples')).not.toStrictEqual({ type: 'apples' })
})
```

toContain

- **Type:** `(received: string) => Awaitable<void>`

`toContain` asserts if actual value is in an array. `toContain` can also check whether a string is a sub-string of another string.

```
import { expect, test } from 'vitest'
import { getAllFruits } from './stock'

test('the fruit list contains orange', () => {
  expect(getAllFruits()).toContain('orange')
})
```

toContainEqual

- **Type:** `(received: any) => Awaitable<void>`

`toContainEqual` asserts if an item with a specific structure and values is contained in an array. It works like `toEqual` inside for each element.

```
import { expect, test } from 'vitest'
import { getFruitStock } from './stock'

test('apple available', () => {
  expect(getFruitStock()).toContainEqual({ fruit: 'apple', count: 5 })
})
```

toHaveLength

- **Type:** `(received: number) => Awaitable<void>`

`toHaveLength` asserts if an object has a `.length` property and it is set to a certain numeric value.

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

test('toHaveLength', () => {
  expect('abc').toHaveLength(3)
  expect([1, 2, 3]).toHaveLength(3)

  expect('').not.toHaveLength(3) // doesn't have .length of 3
  expect({ length: 3 }).toHaveLength(3)
})
```

toHaveProperty

- **Type:** `(key: any, received?: any) => Awaitable<void>`

`toHaveProperty` asserts if a property at provided reference `key` exists for an object.

You can provide an optional value argument also known as deep equality, like the `toEqual` matcher to compare the received property value.

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

const invoice = {
  'isActive': true,
  'P.O': '12345',
  'customer': {
    first_name: 'John',
```

```
    last_name: 'Doe',
    location: 'China',
  },
  'total_amount': 5000,
  'items': [
    {
      type: 'apples',
      quantity: 10,
    },
    {
      type: 'oranges',
      quantity: 5,
    },
  ],
}

test('John Doe Invoice', () => {
  expect(invoice).toHaveProperty('isActive') // assert that the key exists
  expect(invoice).toHaveProperty('total_amount', 5000) // assert that the key
exists and the value is equal

  expect(invoice).not.toHaveProperty('account') // assert that this key does not
exist

  // Deep referencing using dot notation
  expect(invoice).toHaveProperty('customer.first_name')
  expect(invoice).toHaveProperty('customer.last_name', 'Doe')
  expect(invoice).not.toHaveProperty('customer.location', 'India')

  // Deep referencing using an array containing the key
  expect(invoice).toHaveProperty('items[0].type', 'apples')
  expect(invoice).toHaveProperty('items.0.type', 'apples') // dot notation also
works

  // Wrap your key in an array to avoid the key from being parsed as a deep
reference
  expect(invoice).toHaveProperty(['P.O'], '12345')
})
```

toMatch

- **Type:** `(received: string | regexp) => Awaitable<void>`

  `toMatch` asserts if a string matches a regular expression or a string.

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

test('top fruits', () => {
  expect('top fruits include apple, orange and grape').toMatch(/apple/)
  expect('applefruits').toMatch('fruit') // toMatch also accepts a string
})
```

toMatchObject

- **Type:** `(received: object | array) => Awaitable<void>`

  `toMatchObject` asserts if an object matches a subset of the properties of an object.

You can also pass an array of objects. This is useful if you want to check that two arrays match in their number of elements, as opposed to `arrayContaining`, which allows for extra elements in the received array.

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

const johnInvoice = {
  isActive: true,
  customer: {
    first_name: 'John',
    last_name: 'Doe',
    location: 'China',
  },
  total_amount: 5000,
  items: [
    {
      type: 'apples',
      quantity: 10,
    },
    {
      type: 'oranges',
      quantity: 5,
    },
  ],
}

const johnDetails = {
  customer: {
    first_name: 'John',
    last_name: 'Doe',
    location: 'China',
  },
}

test('invoice has john personal details', () => {
  expect(johnInvoice).toMatchObject(johnDetails)
})

test('the number of elements must match exactly', () => {
  // Assert that an array of object matches
  expect([{ foo: 'bar' }, { baz: 1 }]).toMatchObject([
    { foo: 'bar' },
    { baz: 1 },
  ])
})
```

toThrowError

- **Type:** `(received: any) => Awaitable<void>`

  `toThrowError` asserts if a function throws an error when it is called.

  For example, if we want to test that `getFruitStock('pineapples')` throws, we could write:

  You can provide an optional argument to test that a specific error is thrown:

  - regular expression: error message matches the pattern
  - string: error message includes the substring

You must wrap the code in a function, otherwise the error will not be caught, and the assertion will fail.

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

function getFruitStock(type) {
  if (type === 'pineapples')
    throw new DiabetesError('Pineapples is not good for people with diabetes')

  // Do some other stuff
}

test('throws on pineapples', () => {
  // Test that the error message says "diabetes" somewhere: these are equivalent
  expect(() => getFruitStock('pineapples')).toThrowError(/diabetes/)
  expect(() => getFruitStock('pineapples')).toThrowError('diabetes')

  // Test the exact error message
  expect(() => getFruitStock('pineapples')).toThrowError(
    /^Pineapples is not good for people with diabetes$/,
  )
})
```

toMatchSnapshot

- **Type:** `<T>(shape?: Partial<T> | string, message?: string) => void`

This ensures that a value matches the most recent snapshot.

You can provide an optional `hint` string argument that is appended to the test name. Although Vitest always appends a number at the end of a snapshot name, short descriptive hints might be more useful than numbers to differentiate multiple snapshots in a single it or test block. Vitest sorts snapshots by name in the corresponding `.snap` file.

When snapshot mismatch and causing the test failing, if the mismatch is expected, you can press `u` key to update the snapshot for once. Or you can pass `-u` or `--update` CLI options to make Vitest always update the tests.

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

test('matches snapshot', () => {
  const data = { foo: new Set(['bar', 'snapshot']) }
  expect(data).toMatchSnapshot()
})
```

You can also provide a shape of an object, if you are testing just a shape of an object, and don't need it to be 100% compatible:

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

test('matches snapshot', () => {
  const data = { foo: new Set(['bar', 'snapshot']) }
  expect(data).toMatchSnapshot({ foo: expect.any(Set) })
})
```

toMatchInlineSnapshot

- **Type:** `<T>(shape?: Partial<T> | string, snapshot?: string, message?: string) => void`

  This ensures that a value matches the most recent snapshot.

  Vitest adds and updates the inlineSnapshot string argument to the matcher in the test file (instead of an external `.snap` file).

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

test('matches inline snapshot', () => {
  const data = { foo: new Set(['bar', 'snapshot']) }
  // Vitest will update following content when updating the snapshot
  expect(data).toMatchInlineSnapshot(`
    {
      "foo": Set {
        "bar",
        "snapshot",
      },
    }
  `)
})
```

  You can also provide a shape of an object, if you are testing just a shape of an object, and don't need it to be 100% compatible:

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

test('matches snapshot', () => {
  const data = { foo: new Set(['bar', 'snapshot']) }
  expect(data).toMatchInlineSnapshot(
    { foo: expect.any(Set) },
    `
    {
      "foo": Any<Set>,
    }
  `
  )
})
```

toThrowErrorMatchingSnapshot

- **Type:** `(message?: string) => void`

  The same as `toMatchSnapshot`, but expects the same value as `toThrowError`.

  If the function throws an `Error`, the snapshot will be the error message. Otherwise, snapshot will be the value thrown by the function.

toThrowErrorMatchingInlineSnapshot

- **Type:** `(snapshot?: string, message?: string) => void`

  The same as `toMatchInlineSnapshot`, but expects the same value as `toThrowError`.

If the function throws an `Error`, the snapshot will be the error message. Otherwise, snapshot will be the value thrown by the function.

toHaveBeenCalled

- **Type:** `() => Awaitable<void>`

This assertion is useful for testing that a function has been called. Requires a spy function to be passed to `expect`.

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

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
}

test('spy function', () => {
  const buySpy = vi.spyOn(market, 'buy')

  expect(buySpy).not.toHaveBeenCalled()

  market.buy('apples', 10)

  expect(buySpy).toHaveBeenCalled()
})
```

toHaveBeenCalledTimes

- **Type**: `(amount: number) => Awaitable<void>`

This assertion checks if a function was called a certain amount of times. Requires a spy function to be passed to `expect`.

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

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
}

test('spy function called two times', () => {
  const buySpy = vi.spyOn(market, 'buy')

  market.buy('apples', 10)
  market.buy('apples', 20)

  expect(buySpy).toHaveBeenCalledTimes(2)
})
```

toHaveBeenCalledWith

- **Type**: `(...args: any[]) => Awaitable<void>`

This assertion checks if a function was called at least once with certain parameters. Requires a spy function to be passed to `expect`.

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

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
}

test('spy function', () => {
  const buySpy = vi.spyOn(market, 'buy')

  market.buy('apples', 10)
  market.buy('apples', 20)

  expect(buySpy).toHaveBeenCalledWith('apples', 10)
  expect(buySpy).toHaveBeenCalledWith('apples', 20)
})
```

toHaveBeenLastCalledWith

- **Type**: `(...args: any[]) => Awaitable<void>`

This assertion checks if a function was called with certain parameters at it's last invocation. Requires a spy function to be passed to `expect`.

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

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
}

test('spy function', () => {
  const buySpy = vi.spyOn(market, 'buy')

  market.buy('apples', 10)
  market.buy('apples', 20)

  expect(buySpy).not.toHaveBeenLastCalledWith('apples', 10)
  expect(buySpy).toHaveBeenLastCalledWith('apples', 20)
})
```

toHaveBeenNthCalledWith

- **Type**: `(time: number, ...args: any[]) => Awaitable<void>`

This assertion checks if a function was called with certain parameters at the certain time. The count starts at 1. So, to check the second entry, you would write `.toHaveBeenNthCalledWith(2, ...)`.

Requires a spy function to be passed to `expect`.

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

const market = {
  buy(subject: string, amount: number) {
    // ...
  },
}

test('first call of spy function called with right params', () => {
  const buySpy = vi.spyOn(market, 'buy')

  market.buy('apples', 10)
  market.buy('apples', 20)

  expect(buySpy).toHaveBeenNthCalledWith(1, 'apples', 10)
})
```

toHaveReturned

- **Type**: `() => Awaitable<void>`

This assertion checks if a function has successfully returned a value at least once (i.e., did not throw an error). Requires a spy function to be passed to `expect`.

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

const getApplesPrice = (amount: number) => {
  const PRICE = 10
  return amount * PRICE
}

test('spy function returned a value', () => {
  const getPriceSpy = vi.fn(getApplesPrice)

  const price = getPriceSpy(10)

  expect(price).toBe(100)
  expect(getPriceSpy).toHaveReturned()
})
```

toHaveReturnedTimes

- **Type**: `(amount: number) => Awaitable<void>`

This assertion checks if a function has successfully returned a value exact amount of times (i.e., did not throw an error). Requires a spy function to be passed to `expect`.

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

test('spy function returns a value two times', () => {
  const sell = vi.fn((product: string) => ({ product }))

  sell('apples')
  sell('bananas')

  expect(sell).toHaveReturnedTimes(2)
})
```

toHaveReturnedWith

- **Type**: `(returnValue: any) => Awaitable<void>`

You can call this assertion to check if a function has successfully returned a value with certain parameters at least once. Requires a spy function to be passed to `expect`.

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

test('spy function returns a product', () => {
  const sell = vi.fn((product: string) => ({ product }))

  sell('apples')

  expect(sell).toHaveReturnedWith({ product: 'apples' })
})
```

toHaveLastReturnedWith

- **Type**: `(returnValue: any) => Awaitable<void>`

You can call this assertion to check if a function has successfully returned a value with certain parameters on it's last invoking. Requires a spy function to be passed to `expect`.

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

test('spy function returns bananas on a last call', () => {
  const sell = vi.fn((product: string) => ({ product }))

  sell('apples')
  sell('bananas')

  expect(sell).toHaveLastReturnedWith({ product: 'bananas' })
})
```

toHaveNthReturnedWith

- **Type**: `(time: number, returnValue: any) => Awaitable<void>`

You can call this assertion to check if a function has successfully returned a value with certain parameters on a certain call. Requires a spy function to be passed to `expect`.

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

test('spy function returns bananas on second call', () => {
  const sell = vi.fn((product: string) => ({ product }))

  sell('apples')
  sell('bananas')

  expect(sell).toHaveNthReturnedWith(2, { product: 'bananas' })
})
```

toSatisfy

- **Type:** `(predicate: (value: any) => boolean) => Awaitable<void>`

This assertion checks if a value satisfies a certain predicate.

```ts
describe('toSatisfy()', () => {
  const isOdd = (value: number) => value % 2 !== 0

  it('pass with 0', () => {
    expect(1).toSatisfy(isOdd)
  })

  it('pass with negotiation', () => {
    expect(2).not.toSatisfy(isOdd)
  })
})
```

resolves

- **Type:** `Promisify<Assertions>`

  `resolves` is intended to remove boilerplate when asserting asynchronous code. Use it to unwrap value from the pending promise and assert its value with usual assertions. If the promise rejects, the assertion will fail.

  It returns the same `Assertions` object, but all matchers now return `Promise`, so you would need to `await` it. Also works with `chai` assertions.

  For example, if you have a function, that makes an API call and returns some data, you may use this code to assert its return value:

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

async function buyApples() {
  return fetch('/buy/apples').then(r => r.json())
}

test('buyApples returns new stock id', async () => {
  // toEqual returns a promise now, so you HAVE to await it
  await expect(buyApples()).resolves.toEqual({ id: 1 }) // jest API
  await expect(buyApples()).resolves.to.equal({ id: 1 }) // chai API
})
```

  If the assertion is not awaited, then you will have a false-positive test that will pass every time. To make sure that assertions are actually called, you may use [`expect.assertions(number)`](#expect-assertions).

rejects

- **Type:** `Promisify<Assertions>`

  `rejects` is intended to remove boilerplate when asserting asynchronous code. Use it to unwrap reason why promise was rejected, and assert its value with usual assertions. If promise successfully resolves, the assertion will fail.

It returns the same `Assertions` object, but all matchers are now return `Promise`, so you would need to `await` it. Also works with `chai` assertions.

For example, if you have a function that fails when you call it, you may use this code to assert the reason:

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

async function buyApples(id) {
  if (!id)
    throw new Error('no id')
}

test('buyApples throws an error when no id provided', async () => {
  // toThrow returns a promise now, so you HAVE to await it
  await expect(buyApples()).rejects.toThrow('no id')
})
```

If the assertion is not awaited, then you will have a false-positive test that will pass every time. To make sure that assertions are actually happened, you may use [`expect.assertions(number)`](#expect-assertions).

expect.assertions

- **Type:** `(count: number) => void`

After the test has passed or failed verifies that certain number of assertions was called during a test. Useful case would be to check if an asynchronous code was called.

For example, if we have a function that asynchronously calls two matchers, we can assert that they were actually called.

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

async function doAsync(...cbs) {
  await Promise.all(
    cbs.map((cb, index) => cb({ index })),
  )
}

test('all assertions are called', async () => {
  expect.assertions(2)
  function callback1(data) {
    expect(data).toBeTruthy()
  }
  function callback2(data) {
    expect(data).toBeTruthy()
  }

  await doAsync(callback1, callback2)
})
```

expect.hasAssertions

- **Type:** `() => void`

After the test has passed or failed verifies that at least one assertion was called during a test. Useful case would be to check if an asynchronous code was called.

For example, if you have a code that calls a callback, we can make an assertion inside a callback, but the test will always pass, if we don't check if an assertion was called.

```
import { expect, test } from 'vitest'
import { db } from './db'

const cbs = []

function onSelect(cb) {
  cbs.push(cb)
}

// after selecting from db, we call all callbacks
function select(id) {
  return db.select({ id }).then((data) => {
    return Promise.all(
      cbs.map(cb => cb(data)),
    )
  })
}

test('callback was called', async () => {
  expect.hasAssertions()
  onSelect((data) => {
    // should be called on select
    expect(data).toBeTruthy()
  })
  // if not awaited, test will fail
  // if you don't have expect.hasAssertions(), test will pass
  await select(3)
})
```

expect.anything

- **Type:** `() => any`

This asymmetric matcher, when used with equality check, will always return `true`. Useful, if you just want to be sure that the property exist.

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

test('object has "apples" key', () => {
  expect({ apples: 22 }).toEqual({ apples: expect.anything() })
})
```

expect.any

- **Type:** `(constructor: unknown) => any`

This asymmetric matcher, when used with equality check, will return `true` only if value is an instance of specified constructor. Useful, if you have a value that is generated each time, and you only want to know that it exist with a proper type.

```ts
import { expect, test } from 'vitest'
import { generateId } from './generators'

test('"id" is a number', () => {
  expect({ id: generateId() }).toEqual({ id: expect.any(Number) })
})
```

expect.arrayContaining

- **Type:** `<T>(expected: T[]) => any`

  When used with equality check, this asymmetric matcher will return `true` if value is an array and contains specified items.

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

test('basket includes fuji', () => {
  const basket = {
    varieties: [
      'Empire',
      'Fuji',
      'Gala',
    ],
    count: 3
  }
  expect(basket).toEqual({
    count: 3,
    varieties: expect.arrayContaining(['Fuji'])
  })
})
```

> You can use `expect.not` with this matcher to negate the expected value.

expect.objectContaining

- **Type:** `(expected: any) => any`

  When used with equality check, this asymmetric matcher will return `true` if value has a similar shape.

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

test('basket has empire apples', () => {
  const basket = {
    varieties: [
      {
        name: 'Empire',
        count: 1,
      }
    ],
  }
  expect(basket).toEqual({
    varieties: [
      expect.objectContaining({ name: 'Empire' }),
    ]
  })
})
```

You can use `expect.not` with this matcher to negate the expected value.

expect.stringContaining

- **Type:** `(expected: any) => any`

  When used with equality check, this asymmetric matcher will return `true` if value is a string and contains specified substring.

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

test('variety has "Emp" in its name', () => {
  const variety = {
    name: 'Empire',
    count: 1,
  }
  expect(basket).toEqual({
    name: expect.stringContaining('Emp'),
    count: 1,
  })
})
```

You can use `expect.not` with this matcher to negate the expected value.

expect.stringMatching

- **Type:** `(expected: any) => any`

  When used with equality check, this asymmetric matcher will return `true` if value is a string and contains specified substring or the string matches regular expression.

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

test('variety ends with "re"', () => {
  const variety = {
    name: 'Empire',
    count: 1,
  }
  expect(basket).toEqual({
    name: expect.stringMatching(/re$/),
    count: 1,
  })
})
```

You can use `expect.not` with this matcher to negate the expected value.

expect.addSnapshotSerializer

- **Type:** `(plugin: PrettyFormatPlugin) => void`

  This method adds custom serializers that are called when creating a snapshot. This is advanced feature - if you want to know more, please read a guide on custom serializers.

If you are adding custom serializers, you should call this method inside `setupFiles`. This will affect every snapshot.

> If you previously used Vue CLI with Jest, you might want to install [jest-serializer-vue](https://www.npmjs.com/package/jest-serializer-vue). Otherwise, your snapshots will be wrapped in a string, which cases `"` to be escaped.

expect.extend

- **Type:** `(matchers: MatchersObject) => void`

You can extend default matchers with your own. This function is used to extend the matchers object with custom matchers.

When you define matchers that way, you also create asymmetric matchers that can be used like `expect.stringContaining`.

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

test('custom matchers', () => {
  expect.extend({
    toBeFoo: (received, expected) => {
      if (received !== 'foo') {
        return {
          message: () => `expected ${received} to be foo`,
          pass: false,
        }
      }
    },
  })

  expect('foo').toBeFoo()
  expect({ foo: 'foo' }).toEqual({ foo: expect.toBeFoo() })
})
```

> If you want your matchers to appear in every test, you should call this method inside `setupFiles`.

This function is compatible with Jest's `expect.extend`, so any library that uses it to create custom matchers will work with Vitest.

If you are using TypeScript, you can extend default Matchers interface with the code bellow:

```ts
interface CustomMatchers<R = unknown> {
  toBeFoo(): R
}

declare global {
  namespace Vi {
    interface Assertion extends CustomMatchers {}
```

```
    interface AsymmetricMatchersContaining extends CustomMatchers {}
  }
}
```

Note: augmenting jest.Matchers interface will also work.

If you want to know more, checkout [guide on extending matchers](/guide/extending-matchers).

expectTypeOf

- **Type:** `<T>(a: unknown) => ExpectTypeOf`

not

- **Type:** `ExpectTypeOf`

You can negate all assertions, using `.not` property.

toEqualTypeOf

- **Type:** `<T>(expected: T) => void`

This matcher will check, if types are fully equal to each other. This matcher will not fail, if two objects have different values, but the same type, but will fail, if object is missing a property.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf({ a: 1 }).toEqualTypeOf<{ a: number }>()
expectTypeOf({ a: 1 }).toEqualTypeOf({ a: 1 })
expectTypeOf({ a: 1 }).toEqualTypeOf({ a: 2 })
expectTypeOf({ a: 1, b: 1 }).not.toEqualTypeOf<{ a: number }>()
```

toMatchTypeOf

- **Type:** `<T>(expected: T) => void`

This matcher checks if expect type extends provided type. It is different from `toEqual` and is more similar to expect's `toMatch`. With this matcher you can check, if an object "matches" a type.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf({ a: 1, b: 1 }).toMatchTypeOf({ a: 1 })
expectTypeOf<number>().toMatchTypeOf<string | number>()
expectTypeOf<string | number>().not.toMatchTypeOf<number>()
```

extract

- **Type:** `ExpectTypeOf<ExtractedUnion>`

You can use `.extract` to narrow down types for further testing.

```ts
import { expectTypeOf } from 'vitest'

type ResponsiveProp<T> = T | T[] | { xs?: T; sm?: T; md?: T }
const getResponsiveProp = <T>(_props: T): ResponsiveProp<T> => ({})
interface CSSProperties { margin?: string; padding?: string }

const cssProperties: CSSProperties = { margin: '1px', padding: '2px' }

expectTypeOf(getResponsiveProp(cssProperties))
  .extract<{ xs?: any }>() // extracts the last type from a union
  .toEqualTypeOf<{ xs?: CSSProperties; sm?: CSSProperties; md?: CSSProperties }>()

expectTypeOf(getResponsiveProp(cssProperties))
  .extract<unknown[]>() // extracts an array from a union
  .toEqualTypeOf<CSSProperties[]>()
```

If no type is found in the union, `.extract` will return `never`.

exclude

- **Type:** `ExpectTypeOf<NonExcludedUnion>`

You can use `.exclude` to remove types from a union for further testing.

```ts
import { expectTypeOf } from 'vitest'

type ResponsiveProp<T> = T | T[] | { xs?: T; sm?: T; md?: T }
const getResponsiveProp = <T>(_props: T): ResponsiveProp<T> => ({})
interface CSSProperties { margin?: string; padding?: string }

const cssProperties: CSSProperties = { margin: '1px', padding: '2px' }

expectTypeOf(getResponsiveProp(cssProperties))
  .exclude<unknown[]>()
  .exclude<{ xs?: unknown }>() // or just .exclude<unknown[] | { xs?: unknown }>()
  .toEqualTypeOf<CSSProperties>()
```

If no type is found in the union, `.exclude` will return `never`.

returns

- **Type:** `ExpectTypeOf<ReturnValue>`

You can use `.returns` to extract return value of a function type.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(() => {}).returns.toBeVoid()
expectTypeOf((a: number) => [a, a]).returns.toEqualTypeOf([1, 2])
```

If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

parameters

- **Type:** `ExpectTypeOf<Parameters>`

You can extract function arguments with `.parameters` to perform assertions on its value. Parameters are returned as an array.

```ts
import { expectTypeOf } from 'vitest'

type NoParam = () => void
type HasParam = (s: string) => void

expectTypeOf<NoParam>().parameters.toEqualTypeOf<[]>()
expectTypeOf<HasParam>().parameters.toEqualTypeOf<[string]>()
```

> If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

> You can also use [`.toBeCallableWith`](#tobecallablewith) matcher as a more expressive assertion.

parameter

- **Type:** `(nth: number) => ExpectTypeOf`

You can extract a certain function argument with `.parameter(number)` call to perform other assertions on it.

```ts
import { expectTypeOf } from 'vitest'

const foo = (a: number, b: string) => [a, b]

expectTypeOf(foo).parameter(0).toBeNumber()
expectTypeOf(foo).parameter(1).toBeString()
```

> If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

constructorParameters

- **Type:** `ExpectTypeOf<ConstructorParameters>`

You can extract constructor parameters as an array of values and perform assertions on them with this method.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(Date).constructorParameters.toEqualTypeOf<[] | [string | number | Date]>()
```

> If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

> You can also use [`.toBeConstructibleWith`](#tobeconstructiblewith) matcher as a more expressive assertion.

instance

- **Type:** `ExpectTypeOf<ConstructableInstance>`

This property gives access to matchers that can be performed on an instance of the provided class.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(Date).instance.toHaveProperty('toISOString')
```

> If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

items

- **Type:** `ExpectTypeOf<T>`

You can get array item type with `.items` to perform further assertions.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf([1, 2, 3]).items.toEqualTypeOf<number>()
expectTypeOf([1, 2, 3]).items.not.toEqualTypeOf<string>()
```

resolves

- **Type:** `ExpectTypeOf<ResolvedPromise>`

This matcher extracts resolved value of a `Promise`, so you can perform other assertions on it.

```ts
import { expectTypeOf } from 'vitest'

const asyncFunc = async () => 123

expectTypeOf(asyncFunc).returns.resolves.toBeNumber()
expectTypeOf(Promise.resolve('string')).resolves.toBeString()
```

> If used on a non-promise type, it will return `never`, so you won't be able to chain it with other matchers.

guards

- **Type:** `ExpectTypeOf<Guard>`

This matcher extracts guard value (e.g., `v is number`), so you can perform assertions on it.

```ts
import { expectTypeOf } from 'vitest'

const isString = (v: any): v is string => typeof v === 'string'
expectTypeOf(isString).guards.toBeString()
```

Returns `never`, if the value is not a guard function, so you won't be able to chain it with other matchers.

asserts

- **Type:** `ExpectTypeOf<Assert>`

This matcher extracts assert value (e.g., `assert v is number`), so you can perform assertions on it.

```ts
import { expectTypeOf } from 'vitest'

const assertNumber = (v: any): asserts v is number => {
  if (typeof v !== 'number')
    throw new TypeError('Nope !')
}

expectTypeOf(assertNumber).asserts.toBeNumber()
```

Returns `never`, if the value is not an assert function, so you won't be able to chain it with other matchers.

toBeAny

- **Type:** `() => void`

With this matcher you can check, if provided type is `any` type. If the type is too specific, the test will fail.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf<any>().toBeAny()
expectTypeOf({} as any).toBeAny()
expectTypeOf('string').not.toBeAny()
```

toBeUnknown

- **Type:** `() => void`

This matcher checks, if provided type is `unknown` type.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf().toBeUnknown()
expectTypeOf({} as unknown).toBeUnknown()
expectTypeOf('string').not.toBeUnknown()
```

toBeNever

- **Type:** `() => void`

This matcher checks, if provided type is a `never` type.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf<never>().toBeNever()
expectTypeOf((): never => {}).returns.toBeNever()
```

toBeFunction

- **Type:** `() => void`

This matcher checks, if provided type is a `functon`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(42).not.toBeFunction()
expectTypeOf((): never => {}).toBeFunction()
```

toBeObject

- **Type:** `() => void`

This matcher checks, if provided type is an `object`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(42).not.toBeObject()
expectTypeOf({}).toBeObject()
```

toBeArray

- **Type:** `() => void`

This matcher checks, if provided type is `Array<T>`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(42).not.toBeArray()
expectTypeOf([]).toBeArray()
expectTypeOf([1, 2]).toBeArray()
expectTypeOf([{}, 42]).toBeArray()
```

toBeString

- **Type:** `() => void`

This matcher checks, if provided type is a `string`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(42).not.toBeString()
expectTypeOf('').toBeString()
expectTypeOf('a').toBeString()
```

toBeBoolean

- **Type:** `() => void`

This matcher checks, if provided type is `boolean`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(42).not.toBeBoolean()
expectTypeOf(true).toBeBoolean()
expectTypeOf<boolean>().toBeBoolean()
```

toBeVoid

- **Type:** `() => void`

This matcher checks, if provided type is `void`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(() => {}).returns.toBeVoid()
expectTypeOf<void>().toBeVoid()
```

toBeSymbol

- **Type:** `() => void`

This matcher checks, if provided type is a `symbol`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(Symbol(1)).toBeSymbol()
expectTypeOf<symbol>().toBeSymbol()
```

toBeNull

- **Type:** `() => void`

This matcher checks, if provided type is `null`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(null).toBeNull()
expectTypeOf<null>().toBeNull()
expectTypeOf(undefined).not.toBeNull()
```

toBeUndefined

- **Type:** `() => void`

This matcher checks, if provided type is `undefined`.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(undefined).toBeUndefined()
expectTypeOf<undefined>().toBeUndefined()
expectTypeOf(null).not.toBeUndefined()
```

toBeNullable

- **Type:** `() => void`

This matcher checks, if you can use `null` or `undefined` with provided type.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf<1 | undefined>().toBeNullable()
expectTypeOf<1 | null>().toBeNullable()
expectTypeOf<1 | undefined | null>().toBeNullable()
```

toBeCallableWith

- **Type:** `() => void`

This matcher ensures you can call provided function with a set of parameters.

```ts
import { expectTypeOf } from 'vitest'

type NoParam = () => void
type HasParam = (s: string) => void

expectTypeOf<NoParam>().toBeCallableWith()
expectTypeOf<HasParam>().toBeCallableWith('some string')
```

> If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

toBeConstructibleWith

- **Type:** `() => void`

This matcher ensures you can create a new instance with a set of constructor parameters.

```ts
import { expectTypeOf } from 'vitest'

expectTypeOf(Date).toBeConstructibleWith(new Date())
expectTypeOf(Date).toBeConstructibleWith('01-01-2000')
```

> If used on a non-function type, it will return `never`, so you won't be able to chain it with other matchers.

toHaveProperty

- **Type:** `<K extends keyof T>(property: K) => ExpectTypeOf<T[K>`

This matcher checks if a property exists on provided object. If it exists, it also returns the same set of matchers for the type of this property, so you can chain assertions one after another.

```ts
import { expectTypeOf } from 'vitest'

const obj = { a: 1, b: '' }

expectTypeOf(obj).toHaveProperty('a')
expectTypeOf(obj).not.toHaveProperty('c')

expectTypeOf(obj).toHaveProperty('a').toBeNumber()
expectTypeOf(obj).toHaveProperty('b').toBeString()
expectTypeOf(obj).toHaveProperty('a').not.toBeString()
```

assertType

- **Type:** `<T>(value: T): void`

You can use this function as an alternative for `expectTypeOf` to easily assert that argument type is equal to provided generic.

```ts
import { assertType } from 'vitest'

function concat(a: string, b: string): string
function concat(a: number, b: number): number
function concat(a: string | number, b: string | number): string | number

assertType<string>(concat('a', 'b'))
assertType<number>(concat(1, 2))
// @ts-expect-error wrong types
assertType(concat('a', 2))
```

Setup and Teardown

These functions allow you to hook into the life cycle of tests to avoid repeating setup and teardown code. They apply to the current context: the file if they are used at the top-level or the current suite if they are inside a `describe` block. These hooks are not called, when you are running Vitest as a type checker.

beforeEach

- **Type:** `beforeEach(fn: () => Awaitable<void>, timeout?: number)`

  Register a callback to be called before each of the tests in the current context runs. If the function returns a promise, Vitest waits until the promise resolve before running the test.

  Optionally, you can pass a timeout (in milliseconds) defining how long to wait before terminating. The default is 5 seconds.

```ts
import { beforeEach } from 'vitest'

beforeEach(async () => {
  // Clear mocks and add some testing data after before each test run
  await stopMocking()
  await addUser({ name: 'John' })
})
```

Here, the `beforeEach` ensures that user is added for each test.

Since Vitest v0.10.0, `beforeEach` also accepts an optional cleanup function (equivalent to `afterEach`).

```ts
import { beforeEach } from 'vitest'

beforeEach(async () => {
  // called once before each test run
  await prepareSomething()

  // clean up function, called once after each test run
  return async () => {
    await resetSomething()
  }
})
```

afterEach

- **Type:** `afterEach(fn: () => Awaitable<void>, timeout?: number)`

Register a callback to be called after each one of the tests in the current context completes. If the function returns a promise, Vitest waits until the promise resolve before continuing.

Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before terminating. The default is 5 seconds.

```ts
import { afterEach } from 'vitest'

afterEach(async () => {
  await clearTestingData() // clear testing data after each test run
})
```

Here, the `afterEach` ensures that testing data is cleared after each test runs.

beforeAll

- **Type:** `beforeAll(fn: () => Awaitable<void>, timeout?: number)`

Register a callback to be called once before starting to run all tests in the current context. If the function returns a promise, Vitest waits until the promise resolve before running tests.

Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before terminating. The default is 5 seconds.

```ts
import { beforeAll } from 'vitest'

beforeAll(async () => {
  await startMocking() // called once before all tests run
})
```

Here the `beforeAll` ensures that the mock data is set up before tests run.

Since Vitest v0.10.0, `beforeAll` also accepts an optional cleanup function (equivalent to `afterAll`).

```ts
import { beforeAll } from 'vitest'

beforeAll(async () => {
  // called once before all tests run
  await startMocking()

  // clean up function, called once after all tests run
  return async () => {
    await stopMocking()
  }
})
```

afterAll

- **Type:** `afterAll(fn: () => Awaitable<void>, timeout?: number)`

  Register a callback to be called once after all tests have run in the current context. If the function returns a promise, Vitest waits until the promise resolve before continuing.

  Optionally, you can provide a timeout (in milliseconds) for specifying how long to wait before terminating. The default is 5 seconds.

```ts
import { afterAll } from 'vitest'

afterAll(async () => {
  await stopMocking() // this method is called after all tests run
})
```

Here the `afterAll` ensures that `stopMocking` method is called after all tests run.

Vi

Vitest provides utility functions to help you out through it's **vi** helper. You can `import { vi } from 'vitest'` or access it **globally** (when globals configuration is **enabled**).

vi.advanceTimersByTime

- **Type:** `(ms: number) => Vitest`

  Works just like `runAllTimers`, but will end after passed milliseconds. For example this will log `1, 2, 3` and will not throw:

```
let i = 0
setInterval(() => console.log(++i), 50)

vi.advanceTimersByTime(150)
```

vi.advanceTimersToNextTimer

- **Type:** `() => Vitest`

  Will call next available timer. Useful to make assertions between each timer call. You can chain call it to manage timers by yourself.

```
let i = 0
setInterval(() => console.log(++i), 50)

vi.advanceTimersToNextTimer() // log 1
  .advanceTimersToNextTimer() // log 2
  .advanceTimersToNextTimer() // log 3
```

vi.clearAllMocks

Will call `.mockClear()` on all spies. This will clear mock history, but not reset its implementation to the default one.

vi.clearAllTimers

Removes all timers that are scheduled to run. These timers will never run in the future.

vi.dynamicImportSettled

Wait for all imports to load. Useful, if you have a synchronous call that starts importing a module, that you cannot wait otherwise.

vi.fn

- **Type:** `(fn?: Function) => CallableMockInstance`

  Creates a spy on a function, though can be initiated without one. Every time a function is invoked, it stores its call arguments, returns and instances. Also, you can manipulate its behavior with methods. If no function is given, mock will return `undefined`, when invoked.

```
const getApples = vi.fn(() => 0)

getApples()

expect(getApples).toHaveBeenCalled()
expect(getApples).toHaveReturnedWith(0)

getApples.mockReturnValueOnce(5)

const res = getApples()
expect(res).toBe(5)
expect(getApples).toHaveNthReturnedWith(2, 5)
```

vi.getMockedSystemTime

- **Type**: `() => Date | null`

  Returns mocked current date that was set using `setSystemTime`. If date is not mocked, will return `null`.

vi.getRealSystemTime

- **Type**: `() => number`

  When using `vi.useFakeTimers`, `Date.now` calls are mocked. If you need to get real time in milliseconds, you can call this function.

vi.mock

- **Type**: `(path: string, factory?: () => unknown) => void`

  Makes all `imports` to passed module to be mocked. Inside a path you *can* use configured Vite aliases. The call to `vi.mock` is hoisted, so it doesn't matter where you call it. It will always be executed before all imports.

  - If `factory` is defined, will return its result. Factory function can be asynchronous. You may call `vi.importActual` inside to get the original module. Since the call to `vi.mock` is hoisted, you don't have access to variables declared in the global file scope!
  - If mocking a module with a default export, you'll need to provide a `default` key within the returned factory function object. This is an ES modules specific caveat, therefore `jest` documentation may differ as `jest` uses commonJS modules. *Example:*

```ts
vi.mock('path', () => {
  return {
    default: { myDefaultKey: vi.fn() },
    namedExport: vi.fn(),
    // etc...
  }
})
```

  - If `__mocks__` folder with file of the same name exist, all imports will return its exports. For example, `vi.mock('axios')` with `<root>/__mocks__/axios.ts` folder will return everything exported from `axios.ts`.
  - If there is no `__mocks__` folder or a file with the same name inside, will call original module and mock it. (For the rules applied, see algorithm.)

vi.mocked

- **Type**: `<T>(obj: T, deep?: boolean) => MaybeMockedDeep<T>`
- **Type**: `<T>(obj: T, options?: { partial?: boolean; deep?: boolean }) => MaybePartiallyMockedDeep<T>`

  Type helper for TypeScript. In reality just returns the object that was passed.

When `partial` is `true` it will expect a `Partial<T>` as a return value.

```ts
import example from './example'
vi.mock('./example')

test('1+1 equals 2', async () => {
  vi.mocked(example.calc).mockRestore()

  const res = example.calc(1, '+', 1)

  expect(res).toBe(2)
})
```

vi.importActual

- **Type**: `<T>(path: string) => Promise<T>`

  Imports module, bypassing all checks if it should be mocked. Can be useful if you want to mock module partially.

```ts
vi.mock('./example', async () => {
  const axios = await vi.importActual('./example')

  return { ...axios, get: vi.fn() }
})
```

vi.importMock

- **Type**: `<T>(path: string) => Promise<MaybeMockedDeep<T>>`

  Imports a module with all of its properties (including nested properties) mocked. Follows the same rules that `vi.mock` follows. For the rules applied, see algorithm.

vi.resetAllMocks

Will call `.mockReset()` on all spies. This will clear mock history and reset its implementation to an empty function (will return `undefined`).

vi.resetConfig

- **Type**: `RuntimeConfig`

  If `vi.setConfig` was called before, this will reset config to the original state.

vi.resetModules

- **Type**: `() => Vitest`

  Resets modules registry by clearing cache of all modules. Might be useful to isolate modules where local state conflicts between tests.

```ts
import { vi } from 'vitest'

beforeAll(() => {
```

```
  vi.resetModules()
})

test('change state', async () => {
  const mod = await import('./some/path')
  mod.changeLocalState('new value')
  expect(mod.getlocalState()).toBe('new value')
})

test('module has old state', async () => {
  const mod = await import('./some/path')
  expect(mod.getlocalState()).toBe('old value')
})
```

vi.restoreAllMocks

Will call `.mockRestore()` on all spies. This will clear mock history and reset its implementation to the original one.

vi.restoreCurrentDate

- **Type**: `() => void`

  Restores `Date` back to its native implementation.

vi.runAllTicks

- **Type:** `() => Vitest`

  Calls every microtask. These are usually queued by `proccess.nextTick`. This will also run all microtasks scheduled by themselves.

vi.runAllTimers

- **Type:** `() => Vitest`

  This method will invoke every initiated timer until the timers queue is empty. It means that every timer called during `runAllTimers` will be fired. If you have an infinite interval, it will throw after 10 000 tries. For example this will log `1, 2, 3`:

```
let i = 0
setTimeout(() => console.log(++i))
const interval = setInterval(() => {
  console.log(++i)
  if (i === 3)
    clearInterval(interval)

}, 50)

vi.runAllTimers()
```

vi.runOnlyPendingTimers

- **Type:** `() => Vitest`

This method will call every timer that was initiated after `vi.useFakeTimers()` call. It will not fire any timer that was initiated during its call. For example this will only log `1`:

```
let i = 0
setInterval(() => console.log(++i), 50)

vi.runOnlyPendingTimers()
```

vi.setSystemTime

- **Type**: `(date: string | number | Date) => void`

Sets current date to the one that was passed. All `Date` calls will return this date.

Useful if you need to test anything that depends on the current date - for example luxon calls inside your code.

```
const date = new Date(1998, 11, 19)

vi.useFakeTimers()
vi.setSystemTime(date)

expect(Date.now()).toBe(date.valueOf())

vi.useRealTimers()
```

vi.setConfig

- **Type**: `RuntimeConfig`

Updates config for the current test file. You can only affect values that are used, when executing tests.

vi.spyOn

- **Type:** `<T, K extends keyof T>(object: T, method: K, accessType?: 'get' | 'set') => MockInstance`

Creates a spy on a method or getter/setter of an object.

```
let apples = 0
const obj = {
  getApples: () => 13,
}

const spy = vi.spyOn(obj, 'getApples').mockImplementation(() => apples)
apples = 1

expect(obj.getApples()).toBe(1)

expect(spy).toHaveBeenCalled()
expect(spy).toHaveReturnedWith(1)
```

vi.stubGlobal

- **Type**: `(key: keyof globalThis & Window, value: any) => Vitest`

Puts a value on global variable. If you are using `jsdom` or `happy-dom`, also puts the value on `window` object.

Read more in "Mocking Globals" section.

vi.unmock

- **Type**: `(path: string) => void`

Removes module from mocked registry. All subsequent calls to import will return original module even if it was mocked.

vi.useFakeTimers

- **Type:** `() => Vitest`

To enable mocking timers, you need to call this method. It will wrap all further calls to timers (such as `setTimeout`, `setInterval`, `clearTimeout`, `clearInterval`, `nextTick`, `setImmediate`, `clearImmediate`, and `Date`), until `vi.useRealTimers()` is called.

The implementation is based internally on `@sinonjs/fake-timers`.

vi.useRealTimers

- **Type:** `() => Vitest`

When timers are run out, you may call this method to return mocked timers to its original implementations. All timers that were run before will not be restored.

MockInstance Methods

getMockName

- **Type:** `() => string`

Use it to return the name given to mock with method `.mockName(name)`.

mockClear

- **Type:** `() => MockInstance`

Clears all information about every call. After calling it, `spy.mock.calls`, `spy.mock.results` will return empty arrays. It is useful if you need to clean up spy between different assertions.

If you want this method to be called before each test automatically, you can enable `clearMocks` setting in config.

mockName

- **Type:** `(name: string) => MockInstance`

Sets internal mock name. Useful to see what mock has failed the assertion.

mockImplementation

- **Type:** `(fn: Function) => MockInstance`

Accepts a function that will be used as an implementation of the mock.

For example:

```
const mockFn = vi.fn().mockImplementation(apples => apples + 1)
// or: vi.fn(apples => apples + 1);

const NelliesBucket = mockFn(0)
const BobsBucket = mockFn(1)

NelliesBucket === 1 // true
BobsBucket === 2 // true

mockFn.mock.calls[0][0] === 0 // true
mockFn.mock.calls[1][0] === 1 // true
```

mockImplementationOnce

- **Type:** `(fn: Function) => MockInstance`

Accepts a function that will be used as an implementation of the mock for one call to the mocked function. Can be chained so that multiple function calls produce different results.

```
const myMockFn = vi
  .fn()
  .mockImplementationOnce(() => true)
  .mockImplementationOnce(() => false)

myMockFn() // true
myMockFn() // false
```

When the mocked function runs out of implementations, it will invoke the default implementation that was set with `vi.fn(() => defaultValue)` or `.mockImplementation(() => defaultValue)` if they were called:

```
const myMockFn = vi
  .fn(() => 'default')
  .mockImplementationOnce(() => 'first call')
  .mockImplementationOnce(() => 'second call')

// 'first call', 'second call', 'default', 'default'
console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn())
```

mockRejectedValue

- **Type:** `(value: any) => MockInstance`

Accepts an error that will be rejected, when async function will be called.

```
test('async test', async () => {
  const asyncMock = vi.fn().mockRejectedValue(new Error('Async error'))

  await asyncMock() // throws "Async error"
})
```

mockRejectedValueOnce

- **Type:** `(value: any) => MockInstance`

  Accepts a value that will be rejected for one call to the mock function. If chained, every consecutive call will reject passed value.

```
test('async test', async () => {
  const asyncMock = vi
    .fn()
    .mockResolvedValueOnce('first call')
    .mockRejectedValueOnce(new Error('Async error'))

  await asyncMock() // first call
  await asyncMock() // throws "Async error"
})
```

mockReset

- **Type:** `() => MockInstance`

  Does what `mockClear` does and makes inner implementation as an empty function (returning `undefined`, when invoked). This is useful when you want to completely reset a mock back to its initial state.

  If you want this method to be called before each test automatically, you can enable `mockReset` setting in config.

mockRestore

- **Type:** `() => MockInstance`

  Does what `mockReset` does and restores inner implementation to the original function.

  Note that restoring mock from `vi.fn()` will set implementation to an empty function that returns `undefined`. Restoring a `vi.fn(impl)` will restore implementation to `impl`.

  If you want this method to be called before each test automatically, you can enable `restoreMocks` setting in config.

mockResolvedValue

- **Type:** `(value: any) => MockInstance`

  Accepts a value that will be resolved, when async function will be called.

```ts
test('async test', async () => {
  const asyncMock = vi.fn().mockResolvedValue(43)

  await asyncMock() // 43
})
```

mockResolvedValueOnce

- **Type:** `(value: any) => MockInstance`

Accepts a value that will be resolved for one call to the mock function. If chained, every consecutive call will resolve passed value.

```ts
test('async test', async () => {
  const asyncMock = vi
    .fn()
    .mockResolvedValue('default')
    .mockResolvedValueOnce('first call')
    .mockResolvedValueOnce('second call')

  await asyncMock() // first call
  await asyncMock() // second call
  await asyncMock() // default
  await asyncMock() // default
})
```

mockReturnThis

- **Type:** `() => MockInstance`

Sets inner implementation to return `this` context.

mockReturnValue

- **Type:** `(value: any) => MockInstance`

Accepts a value that will be returned whenever the mock function is called.

```ts
const mock = vi.fn()
mock.mockReturnValue(42)
mock() // 42
mock.mockReturnValue(43)
mock() // 43
```

mockReturnValueOnce

- **Type:** `(value: any) => MockInstance`

Accepts a value that will be returned for one call to the mock function. If chained, every consecutive call will return passed value. When there are no more `mockReturnValueOnce` values to use, calls a function specified by `mockImplementation` or other `mockReturn*` methods.

```ts
const myMockFn = vi
  .fn()
  .mockReturnValue('default')
```

```
  .mockReturnValueOnce('first call')
  .mockReturnValueOnce('second call')

// 'first call', 'second call', 'default', 'default'
console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn())
```

MockInstance Properties

mock.calls

This is an array containing all arguments for each call. One item of the array is arguments of that call.

If a function was invoked twice with the following arguments `fn(arg1, arg2)`, `fn(arg3, arg4)` in that order, then `mock.calls` will be:

```
[
  ['arg1', 'arg2'],
  ['arg3', 'arg4'],
]
```

mock.lastCall

This contains the arguments of the last call. If spy wasn't called, will return `undefined`.

mock.results

This is an array containing all values, that were `returned` from function. One item of the array is an object with properties `type` and `value`. Available types are:

- `'return'` - function returned without throwing.
- `'throw'` - function threw a value.

The `value` property contains returned value or thrown error.

If function returned `result`, then threw an error, then `mock.results` will be:

```
[
  {
    type: 'return',
    value: 'result',
  },
  {
    type: 'throw',
    value: Error,
  },
]
```

mock.instances

This is an array containing all instances that were instantiated when mock was called with a `new` keyword. Note, this is an actual context (`this`) of the function, not a return value.

For example, if mock was instantiated with `new MyClass()`, then `mock.instances` will be an array of one value:

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

const MyClass = vi.fn()

const a = new MyClass()

expect(MyClass.mock.instances[0]).toBe(a)
```

If you return a value from constructor, it will not be in `instances` array, but instead on `results`:

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

const Spy = vi.fn(() => ({ method: vi.fn() }))

const a = new Spy()

expect(Spy.mock.instances[0]).not.toBe(a)
expect(Spy.mock.results[0]).toBe(a)
```

Go to TOC

Configuring Vitest

Configuration

`vitest` will read your root `vite.config.ts` when it is present to match with the plugins and setup as your Vite app. If you want to have a different configuration for testing or your main app doesn't rely on Vite specifically, you could either:

- Create `vitest.config.ts`, which will have the higher priority and will override the configuration from `vite.config.ts`
- Pass `--config` option to CLI, e.g. `vitest --config ./path/to/vitest.config.ts`
- Use `process.env.VITEST` or `mode` property on `defineConfig` (will be set to `test` / `benchmark` if not overridden) to conditionally apply different configuration in `vite.config.ts`

To configure `vitest` itself, add `test` property in your Vite config. You'll also need to add a reference to Vitest types using a triple slash command at the top of your config file, if you are importing `defineConfig` from `vite` itself.

using `defineConfig` from `vite` you should follow this:

```ts
/// <reference types="vitest" />
import { defineConfig } from 'vite'

export default defineConfig({
  test: {
    // ...
  },
})
```

using `defineConfig` from `vitest/config` you should follow this:

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

export default defineConfig({
  test: {
    // ...
  },
})
```

You can retrieve Vitest's default options to expand them if needed:

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

export default defineConfig({
  test: {
    exclude: [...configDefaults.exclude, 'packages/template/*'],
  },
})
```

When using a separate `vitest.config.js`, you can also extend Vite's options from another config file if needed:

```ts
import { mergeConfig } from 'vite'
import { defineConfig } from 'vitest/config'
import viteConfig from './vite.config'

export default mergeConfig(viteConfig, defineConfig({
  test: {
    exclude: ['packages/template/*'],
  },
}))
```

Options

In addition to the following options, you can also use any configuration option from [Vite](https://vitejs.dev/config/). For example, `define` to define global variables, or `resolve.alias` to define aliases.

include

- **Type:** `string[]`
- **Default:** `['**/*.{test,spec}.{js,mjs,cjs,ts,mts,cts,jsx,tsx}']`

Files to include in the test run, using glob pattern.

exclude

- **Type:** `string[]`
- **Default:** `['**/node_modules/**', '**/dist/**', '**/cypress/**', '**/.{idea,git,cache,output,temp}/**', '**/{karma,rollup,webpack,vite,vitest,jest,ava,babel,nyc,cypress}.config.*']`

Files to exclude from the test run, using glob pattern.

deps

- **Type:** `{ external?, inline? }`

Handling for dependencies inlining or externalizing

deps.external

- **Type:** `(string | RegExp)[]`
- **Default:** `['**/node_modules/**', '**/dist/**']`

Externalize means that Vite will bypass the package to native Node. Externalized dependencies will not be applied Vite's transformers and resolvers, so they do not support HMR on reload. Typically, packages under `node_modules` are externalized.

deps.inline

- **Type:** `(string | RegExp)[] | true`
- **Default:** `[]`

Vite will process inlined modules. This could be helpful to handle packages that ship `.js` in ESM format (that Node can't handle).

If `true`, every dependency will be inlined. All dependencies, specified in `ssr.noExternal` will be inlined by default.

deps.fallbackCJS

- **Type** `boolean`
- **Default:** `false`

When a dependency is a valid ESM package, try to guess the cjs version based on the path. This might be helpful, if a dependency has the wrong ESM file.

This might potentially cause some misalignment if a package has different logic in ESM and CJS mode.

deps.registerNodeLoader

- **Type:** `boolean`
- **Default:** `false`

Use experimental Node loader to resolve imports inside `node_modules`, using Vite resolve algorithm.

If disabled, your `alias` and `<plugin>.resolveId` won't affect imports inside `node_modules` or `deps.external`.

deps.interopDefault

- **Type:** `boolean`
- **Default:** `true`

Interpret CJS module's default as named exports.

benchmark

- **Type:** `{ include?, exclude?, ... }`

Options used when running `vitest bench`.

benchmark.include

- **Type:** `string[]`
- **Default:** `['**/*.{bench,benchmark}.{js,mjs,cjs,ts,mts,cts,jsx,tsx}']`

Include globs for benchmark test files

benchmark.exclude

- **Type:** `string[]`
- **Default:** `['node_modules', 'dist', '.idea', '.git', '.cache']`

Exclude globs for benchmark test files

benchmark.includeSource

- **Type:** `string[]`
- **Default:** `[]`

Include globs for in-source benchmark test files. This option is similar to `includeSource`.

When defined, Vitest will run all matched files with `import.meta.vitest` inside.

benchmark.reporters

- **Type:** `Arrayable<BenchmarkBuiltinReporters | Reporter>`
- **Default:** `'default'`

Custom reporter for output. Can contain one or more built-in report names, reporter instances, and/or paths to custom reporters.

benchmark.outputFile

- **Type:** `string | Record<string, string>`

Write benchmark results to a file when the `--reporter=json` option is also specified. By providing an object instead of a string you can define individual outputs when using multiple reporters.

To provide object via CLI command, use the following syntax: `--outputFile.json=./path --outputFile.junit=./other-path`.

alias

- **Type:** `Record<string, string> | Array<{ find: string | RegExp, replacement: string, customResolver?: ResolverFunction | ResolverObject }>`

Define custom aliases when running inside tests. They will be merged with aliases from `resolve.alias`.

globals

- **Type:** `boolean`
- **Default:** `false`

By default, `vitest` does not provide global APIs for explicitness. If you prefer to use the APIs globally like Jest, you can pass the `--globals` option to CLI or add `globals: true` in the config.

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

export default defineConfig({
  test: {
    globals: true,
  },
})
```

To get TypeScript working with the global APIs, add `vitest/globals` to the `types` field in your `tsconfig.json`

```
// tsconfig.json
{
  "compilerOptions": {
    "types": ["vitest/globals"]
  }
}
```

If you are already using `unplugin-auto-import` in your project, you can also use it directly for auto importing those APIs.

```
// vite.config.ts
import { defineConfig } from 'vitest/config'
import AutoImport from 'unplugin-auto-import/vite'

export default defineConfig({
  plugins: [
    AutoImport({
      imports: ['vitest'],
      dts: true, // generate TypeScript declaration
    }),
  ],
})
```

environment

- **Type:** `'node' | 'jsdom' | 'happy-dom' | 'edge-runtime' | string`
- **Default:** `'node'`

The environment that will be used for testing. The default environment in Vitest is a Node.js environment. If you are building a web application, you can use browser-like environment through either `jsdom` or `happy-dom` instead. If you are building edge functions, you can use `edge-runtime` environment

By adding a `@vitest-environment` docblock or comment at the top of the file, you can specify another environment to be used for all tests in that file:

Docblock style:

```
/**
 * @vitest-environment jsdom
 */

test('use jsdom in this test file', () => {
  const element = document.createElement('div')
  expect(element).not.toBeNull()
})
```

Comment style:

```
// @vitest-environment happy-dom

test('use happy-dom in this test file', () => {
```

```
  const element = document.createElement('div')
  expect(element).not.toBeNull()
})
```

For compatibility with Jest, there is also a `@jest-environment`:

```
/**
 * @jest-environment jsdom
 */

test('use jsdom in this test file', () => {
  const element = document.createElement('div')
  expect(element).not.toBeNull()
})
```

If you are running Vitest with `--no-threads` flag, your tests will be run in this order: `node`, `jsdom`, `happy-dom`, `edge-runtime`, `custom environments`. Meaning, that every test with the same environment is grouped together, but is still running sequentially.

Starting from 0.23.0, you can also define custom environment. When non-builtin environment is used, Vitest will try to load package `vitest-environment-${name}`. That package should export an object with the shape of `Environment`:

```
import type { Environment } from 'vitest'

export default <Environment>{
  name: 'custom',
  setup() {
    // custom setup
    return {
      teardown() {
        // called after all tests with this env have been run
      }
    }
  }
}
```

Vitest also exposes `builtinEnvironments` through `vitest/environments` entry, in case you just want to extend it. You can read more about extending environments in our guide.

environmentOptions

- **Type:** `Record<'jsdom' | string, unknown>`
- **Default:** `{}`

These options are passed down to `setup` method of current `environment`. By default, you can configure only JSDOM options, if you are using it as your test environment.

update

- **Type:** `boolean`
- **Default:** `false`

Update snapshot files. This will update all changed snapshots and delete obsolete ones.

watch

- **Type:** `boolean`
- **Default:** `true`

Enable watch mode

root

- **Type:** `string`

Project root

reporters

- **Type:** `Reporter | Reporter[]`
- **Default:** `'default'`

Custom reporters for output. Reporters can be a Reporter instance or a string to select built in reporters:

- `'default'` - collapse suites when they pass
- `'verbose'` - keep the full task tree visible
- `'dot'` - show each task as a single dot
- `'junit'` - JUnit XML reporter
- `'json'` - give a simple JSON summary
- path of a custom reporter (e.g. `'./path/to/reporter.ts'`, `'@scope/reporter'`)

outputTruncateLength

- **Type:** `number`
- **Default:** `80`

Truncate output diff lines up to `80` number of characters. You may wish to tune this, depending on your terminal window width.

outputDiffLines

- **Type:** `number`
- **Default:** `15`

Limit number of output diff lines up to `15`.

outputFile

- **Type:** `string | Record<string, string>`

Write test results to a file when the `--reporter=json` or `--reporter=junit` option is also specified. By providing an object instead of a string you can define individual outputs when using multiple reporters.

To provide object via CLI command, use the following syntax: `--outputFile.json=./path --outputFile.junit=./other-path`.

threads

- **Type:** `boolean`
- **Default:** `true`

Enable multi-threading using tinypool (a lightweight fork of Piscina)

> This option is different from Jest's `--runInBand`. Vitest uses workers not only for running tests in parallel, but also to provide isolation. By disabling this option, your tests will run sequentially, but in the same global context, so you must provide isolation yourself.
>
> This might cause all sorts of issues, if you are relying on global state (frontend frameworks usually do) or your code relies on environment to be defined separately for each test. But can be a speed boost for your tests (up to 3 times faster), that don't necessarily rely on global state or can easily by-pass that.

maxThreads

- **Type:** `number`
- **Default:** *available CPUs*

Maximum number of threads. You can also use `VITEST_MAX_THREADS` environment variable.

minThreads

- **Type:** `number`
- **Default:** *available CPUs*

Minimum number of threads. You can also use `VITEST_MIN_THREADS` environment variable.

testTimeout

- **Type:** `number`
- **Default:** `5000`

Default timeout of a test in milliseconds

hookTimeout

- **Type:** `number`
- **Default:** `10000`

Default timeout of a hook in milliseconds

teardownTimeout

- **Type:** `number`
- **Default:** `1000`

Default timeout to wait for close when Vitest shuts down, in milliseconds

silent

- **Type:** `boolean`
- **Default:** `false`

Silent console output from tests

setupFiles

- **Type:** `string | string[]`

Path to setup files. They will be run before each test file.

You can use `process.env.VITEST_POOL_ID` (integer-like string) inside to distinguish between threads (will always be `'1'`, if run with `threads: false`).

> Note, that if you are running [`--no-threads`](#threads), this setup file will be run in the same global scope multiple times. Meaning, that you are accessing the same global object before each test, so make sure you are not doing the same thing more than you need.need.

For example, you may rely on a global variable:

```
import { config } from '@some-testing-lib'

if (!globalThis.defined) {
  config.plugins = [myCoolPlugin]
  computeHeavyThing()
  globalThis.defined = true
}

// hooks are reset before each suite
afterEach(() => {
  cleanup()
})

globalThis.resetBeforeEachTest = true
```

globalSetup

- **Type:** `string | string[]`

Path to global setup files, relative to project root

A global setup file can either export named functions `setup` and `teardown` or a `default` function that returns a teardown function (example).

info Multiple globalSetup files are possible. setup and teardown are executed sequentially with teardown in reverse order.

Beware that the global setup is run in a different global scope, so your tests don't have access to variables defined here.

watchExclude

- **Type:** `string[]`
- **Default:** `['**/node_modules/**', '**/dist/**']`

Glob pattern of file paths to be ignored from triggering watch rerun.

forceRerunTriggers

- **Type**: `string[]`
- **Default:** `['**/package.json/**', '**/vitest.config.*/**', '**/vite.config.*/**']`

Glob pattern of file paths that will trigger the whole suite rerun. When paired with the `--changed` argument will run the whole test suite if the trigger is found in the git diff.

Useful if you are testing calling CLI commands, because Vite cannot construct a module graph:

```
test('execute a script', async () => {
  // Vitest cannot rerun this test, if content of `dist/index.js` changes
  await execa('node', ['dist/index.js'])
})
```

Make sure that your files are not excluded by `watchExclude`.

isolate

- **Type:** `boolean`
- **Default:** `true`

Isolate environment for each test file. Does not work if you disable `--threads`.

coverage

- **Type:** `CoverageC8Options | CoverageIstanbulOptions`
- **Default:** `undefined`

You can use `c8` or `istanbul` for coverage collection.

provider

- **Type:** `'c8' | 'istanbul'`
- **Default:** `'c8'`

Use `provider` to select the tool for coverage collection.

CoverageC8Options

Used when `provider: 'c8'` is set. Coverage options are passed to `c8`.

CoverageIstanbulOptions

Used when `provider: 'istanbul'` is set.

include

- **Type:** `string[]`
- **Default:** `['**']`

List of files included in coverage as glob patterns

exclude

- **Type:** `string[]`
- **Default:** `['coverage/**', 'dist/**', 'packages/*/test{,s}/**', '**/*.d.ts', 'cypress/**', 'test{,s}/**', 'test{,-*}.{js,cjs,mjs,ts,tsx,jsx}', '**/*{.,-}test.{js,cjs,mjs,ts,tsx,jsx}', '**/*{.,-}spec.{js,cjs,mjs,ts,tsx,jsx}', '**/__tests__/**', '**/{karma,rollup,webpack,vite,vitest,jest,ava,babel,nyc,cypress}.config.{js,cjs,mjs,ts}', '**/.{eslint,mocha,prettier}rc.{js,cjs,yml}']`

List of files excluded from coverage as glob patterns.

skipFull

- **Type:** `boolean`
- **Default:** `false`

Do not show files with 100% statement, branch, and function coverage.

perFile

- **Type:** `boolean`
- **Default:** `false`

Check thresholds per file.

lines

- **Type:** `number`

Threshold for lines.

functions

- **Type:** `number`

Threshold for functions.

branches

- **Type:** `number`

Threshold for branches.

statements

- **Type:** `number`

Threshold for statements.

ignoreClassMethods

- **Type:** `string[]`
- **Default:** []

Set to array of class method names to ignore for coverage.

watermarks

- **Type:**

```
{
  statements?: [number, number],
  functions?: [number, number],
  branches?: [number, number],
  lines?: [number, number]
}
```

- **Default:**

```
{
  statements: [50, 80],
  functions: [50, 80],
  branches: [50, 80],
  lines: [50, 80]
}
```

Watermarks for statements, lines, branches and functions.

all

- **Type:** `boolean`
- **Default:** false

Whether to include all files, including the untested ones into report.

testNamePattern

- **Type** `string | RegExp`

Run tests with full names matching the pattern. If you add `OnlyRunThis` to this property, tests not containing the word `OnlyRunThis` in the test name will be skipped.

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

// run
test('OnlyRunThis', () => {
  expect(true).toBe(true)
```

```
})

// skipped
test('doNotRun', () => {
  expect(true).toBe(true)
})
```

open

- **Type:** `boolean`
- **Default:** `false`

Open Vitest UI (WIP)

api

- **Type:** `boolean | number`
- **Default:** `false`

Listen to port and serve API. When set to true, the default port is 51204

clearMocks

- **Type:** `boolean`
- **Default:** `false`

Will call `.mockClear()` on all spies before each test. This will clear mock history, but not reset its implementation to the default one.

mockReset

- **Type:** `boolean`
- **Default:** `false`

Will call `.mockReset()` on all spies before each test. This will clear mock history and reset its implementation to an empty function (will return `undefined`).

restoreMocks

- **Type:** `boolean`
- **Default:** `false`

Will call `.mockRestore()` on all spies before each test. This will clear mock history and reset its implementation to the original one.

transformMode

- **Type:** `{ web?, ssr? }`

Determine the transform method of modules

transformMode.ssr

- **Type:** `RegExp[]`
- **Default:** `[/\.([cm]?[jt]sx?|json)$/]`

Use SSR transform pipeline for the specified files.
Vite plugins will receive `ssr: true` flag when processing those files.

transformMode.web

- **Type:** `RegExp[]`
- **Default:** *modules other than those specified in* *`transformMode.ssr`*

First do a normal transform pipeline (targeting browser), then do a SSR rewrite to run the code in Node.
Vite plugins will receive `ssr: false` flag when processing those files.

When you use JSX as component models other than React (e.g. Vue JSX or SolidJS), you might want to config as following to make `.tsx` / `.jsx` transformed as client-side components:

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

export default defineConfig({
  test: {
    transformMode: {
      web: [/\.[jt]sx$/],
    },
  },
})
```

snapshotFormat

- **Type:** `PrettyFormatOptions`

Format options for snapshot testing. These options are passed down to `pretty-format`.

resolveSnapshotPath

- **Type**: `(testPath: string, snapExtension: string) => string`
- **Default**: stores snapshot files in `__snapshots__` directory

Overrides default snapshot path. For example, to store snapshots next to test files:

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

export default defineConfig({
  test: {
    resolveSnapshotPath: (testPath, snapExtension) => testPath + snapExtension,
  },
})
```

allowOnly

- **Type**: `boolean`
- **Default**: `false`

Allow tests and suites that are marked as only.

dangerouslyIgnoreUnhandledErrors

- **Type**: `boolean`
- **Default**: `false`

Ignore any unhandled errors that occur.

passWithNoTests

- **Type**: `boolean`
- **Default**: `false`

Vitest will not fail, if no tests will be found.

logHeapUsage

- **Type**: `boolean`
- **Default**: `false`

Show heap usage after each test. Useful for debugging memory leaks.

css

- **Type**: `boolean | { include?, exclude?, modules? }`

Configure if CSS should be processed. When excluded, CSS files will be replaced with empty strings to bypass the subsequent processing. CSS Modules will return a proxy to not affect runtime.

css.include

- **Type**: `RegExp | RegExp[]`
- **Default**: `[]`

RegExp pattern for files that should return actual CSS and will be processed by Vite pipeline.

> To process all CSS files, use `/.+/`.

css.exclude

- **Type**: `RegExp | RegExp[]`
- **Default**: `[]`

RegExp pattern for files that will return an empty CSS file.

css.modules

- **Type**: `{ classNameStrategy? }`
- **Default**: `{}`

css.modules.classNameStrategy

- **Type**: `'stable' | 'scoped' | 'non-scoped'`
- **Default**: `'stable'`

If you decide to process CSS files, you can configure if class names inside CSS modules should be scoped. You can choose one of the options:

- `stable`: class names will be generated as `_${name}_${hashedFilename}`, which means that generated class will stay the same, if CSS content is changed, but will change, if the name of the file is modified, or file is moved to another folder. This setting is useful, if you use snapshot feature.
- `scoped`: class names will be generated as usual, respecting `css.modules.generateScopeName` method, if you have one and CSS processing is enabled. By default, filename will be generated as `_${name}_${hash}`, where hash includes filename and content of the file.
- `non-scoped`: class names will not be hashed.

> By default, Vitest exports a proxy, bypassing CSS Modules processing. If you rely on CSS properties on your classes, you have to enable CSS processing using `include` option.

maxConcurrency

- **Type**: `number`
- **Default**: `5`

A number of tests that are allowed to run at the same time marked with `test.concurrent`.

Test above this limit will be queued to run when available slot appears.

cache

- **Type**: `false | { dir? }`

Options to configure Vitest cache policy. At the moment Vitest stores cache for test results to run the longer and failed tests first.

cache.dir

- **Type**: `string`
- **Default**: `node_modules/.vitest`

Path to cache directory.

sequence

- **Type**: `{ sequencer?, shuffle?, seed?, hooks? }`

Options for how tests should be sorted.

sequence.sequencer

- **Type**: `TestSequencerConstructor`
- **Default**: `BaseSequencer`

A custom class that defines methods for sharding and sorting. You can extend `BaseSequencer` from `vitest/node`, if you only need to redefine one of the `sort` and `shard` methods, but both should exist.

Sharding is happening before sorting, and only if `--shard` option is provided.

sequence.shuffle

- **Type**: `boolean`
- **Default**: `false`

If you want tests to run randomly, you can enable it with this option, or CLI argument `--sequence.shuffle`.

Vitest usually uses cache to sort tests, so long running tests start earlier - this makes tests run faster. If your tests will run in random order you will lose this performance improvement, but it may be useful to track tests that accidentally depend on another run previously.

sequence.seed

- **Type**: `number`
- **Default**: `Date.now()`

Sets the randomization seed, if tests are running in random order.

sequence.hooks

- **Type**: `'stack' | 'list' | 'parallel'`
- **Default**: `'parallel'`

Changes the order in which hooks are executed.

- `stack` will order "after" hooks in reverse order, "before" hooks will run in the order they were defined
- `list` will order all hooks in the order they are defined
- `parallel` will run hooks in a single group in parallel (hooks in parent suites will still run before the current suite's hooks)

typecheck

Options for configuring typechecking test environment.

typecheck.checker

- **Type**: `'tsc' | 'vue-tsc' | string`
- **Default**: `tsc`

What tools to use for type checking. Vitest will spawn a process with certain parameters for easier parsing, depending on the type. Checker should implement the same output format as `tsc`.

You need to have a package installed to use typecheker:

- `tsc` requires `typescript` package
- `vue-tsc` requires `vue-tsc` package

You can also pass down a path to custom binary or command name that produces the same output as `tsc --noEmit --pretty false`.

typecheck.include

- **Type**: `string[]`
- **Default**: `['**/*.{test,spec}-d.{ts,js}']`

Glob pattern for files that should be treated as test files

typecheck.exclude

- **Type**: `string[]`
- **Default**: `['**/node_modules/**', '**/dist/**', '**/cypress/**', '**/.{idea,git,cache,output,temp}/**']`

Glob pattern for files that should not be treated as test files

typecheck.allowJs

- **Type**: `boolean`
- **Default**: `false`

Check JS files that have `@ts-check` comment. If you have it enabled in tsconfig, this will not overwrite it.

typecheck.ignoreSourceErrors

- **Type**: `boolean`
- **Default**: `false`

Do not fail, if Vitest found errors outside the test files. This will not show you non-test errors at all.

By default, if Vitest finds source error, it will fail test suite.

typecheck.tsconfig

- **Type**: `string`
- **Default**: *tries to find closest tsconfig.json*

Path to custom tsconfig, relative to the project root.

Go to TOC

Command Line Interface

Commands

`vitest`

Start Vitest in the current directory. Will enter the watch mode in development environment and run mode in CI automatically.

You can pass an additional argument as the filter of the test files to run. For example:

```
vitest foobar
```

Will run only the test file that contains `foobar` in their paths.

`vitest run`

Perform a single run without watch mode.

`vitest watch`

Run all test suites but watch for changes and rerun tests when they change. Same as calling `vitest` without an argument. Will fallback to `vitest run` in CI.

`vitest dev`

Alias to `vitest watch`.

`vitest related`

Run only tests that cover a list of source files. Works with static imports (e.g., `import('./index.ts')` or `import index from './index.ts`), but not the dynamic ones (e.g., `import(filepath)`). All files should be relative to root folder.

Useful to run with `lint-staged` or with your CI setup.

```
vitest related /src/index.ts /src/hello-world.js
```

Options

Options	
`-v, --version`	Display version number
`-r, --root <path>`	Define the project root
`-c, --config <path>`	Path to config file
`-u, --update`	Update snapshots

Options	
`-w, --watch`	Smart & instant watch mode
`-t, --testNamePattern <pattern>`	Run tests with full names matching the pattern
`--dir <path>`	Base directory to scan for the test files
`--ui`	Enable UI
`--open`	Open the UI automatically if enabled (default: `true`)
`--api [api]`	Serve API, available options: `--api.port <port>`, `--api.host [host]` and `--api.strictPort`
`--threads`	Enable Threads (default: `true`)
`--silent`	Silent console output from tests
`--isolate`	Isolate environment for each test file (default: `true`)
`--reporter <name>`	Select reporter: `default`, `verbose`, `dot`, `junit`, `json`, or a path to a custom reporter
`--outputTruncateLength <length>`	Truncate output diff lines up to `<length>` number of characters.
`--outputDiffLines <lines>`	Limit number of output diff lines up to `<lines>`.
`--outputFile <filename/-s>`	Write test results to a file when the `--reporter=json` or `--reporter=junit` option is also specified Via [cac's dot notation] you can specify individual outputs for multiple reporters
`--coverage`	Enable coverage report
`--run`	Do not watch
`--mode`	Override Vite mode (default: `test`)
`--mode <name>`	Override Vite mode (default: `test`)
`--globals`	Inject APIs globally
`--dom`	Mock browser api with happy-dom
`--browser`	Run tests in browser
`--environment <env>`	Runner environment (default: `node`)
`--passWithNoTests`	Pass when no tests found
`--allowOnly`	Allow tests and suites that are marked as `only` (default: false in CI, true otherwise)
`--dangerouslyIgnoreUnhandledErrors`	Ignore any unhandled errors that occur

Options	
`--changed [since]`	Run tests that are affected by the changed files (default: false). See docs
`--shard <shard>`	Execute tests in a specified shard
`--sequence`	Define in what order to run tests. Use [cac's dot notation] to specify options (for example, use `--sequence.shuffle` to run tests in random order)
`--no-color`	Removes colors from the console output
`--inspect`	Enables Node.js inspector
`--inspect-brk`	Enables Node.js inspector with break
`-h, --help`	Display available CLI options

changed

- **Type**: `boolean | string`
- **Default**: false

  Run tests only against changed files. If no value is provided, it will run tests against uncommitted changes (including staged and unstaged).

  To run tests against changes made in the last commit, you can use `--changed HEAD~1`. You can also pass commit hash or branch name.

  If paired with the `forceRerunTriggers` config option it will run the whole test suite if a match is found.

shard

- **Type**: `string`
- **Default**: disabled

  Test suite shard to execute in a format of `<index>/<count>`, where

  - `count` is a positive integer, count of divided parts
  - `index` is a positive integer, index of divided part

  This command will divide all tests into `count` equal parts, and will run only those that happen to be in an `index` part. For example, to split your tests suite into three parts, use this:

```
vitest run --shard=1/3
vitest run --shard=2/3
vitest run --shard=3/3
```

You cannot use this option with `--watch` enabled (enabled in dev by default).

Go to TOC

Comparisons with Other Test Runners

Jest

Jest took over the Testing Framework space by providing out-of-the-box support for most JavaScript projects, a comfortable API (`it` and `expect`), and the full pack of testing features that most setups would require (snapshots, mocks, coverage). We are thankful to the Jest team and community for creating a delightful testing API and pushing forward a lot of the testing patterns that are now a standard in the web ecosystem. It is possible to use Jest in Vite setups. @sodatea is building vite-jest, which aims to provide first-class Vite integration for Jest. The last blockers in Jest have been solved so this is a valid option for your unit tests. However, in a world where we have Vite providing support for the most common web tooling (typescript, JSX, most popular UI Frameworks), Jest represents a duplication of complexity. If your app is powered by Vite, having two different pipelines to configure and maintain is not justifiable. With Vitest you get to define the configuration for your dev, build and test environments as a single pipeline, sharing the same plugins and the same vite.config.js. Even if your library is not using Vite (for example, if it is built with esbuild or rollup), Vitest is an interesting option as it gives you a faster run for your unit tests and a jump in DX thanks to the default watch mode using Vite instant Hot Module Reload (HMR). Vitest offers compatibility with most of the Jest API and ecosystem libraries, so in most projects, it should be a drop-in replacement for Jest.

Cypress

Cypress is a browser-based test runner and a complementary tool to Vitest. If you'd like to use Cypress, we suggest using Vitest for all headless logic in your application and Cypress for all browser-based logic.

Cypress is known as an end-to-end testing tool, however their new component test runner has great support for testing Vite components and is an ideal choice to test anything that renders in a browser.

Browser-based runners, like Cypress and Web Test Runner, will catch issues that Vitest cannot because they use the real browser and real browser APIs.

Cypress's test driver is focused on determining if elements are visible, accessible, and interactive. Cypress is purpose-built for UI development and testing and its DX is centered around test driving your visual components. You see your component rendered alongside the test reporter. Once the test is complete, the component remains interactive and you can debug any failures that occur using your browser devtools.

In contrast, Vitest is focused on delivering the best DX possible for lightning fast, *headless* testing. Node-based runners like Vitest support various partially-implemented browser environments, like `jsdom`, which implement enough for you to quickly unit test any code that references browser APIs. The tradeoff is that these browser environments have limitations in what they can implement. For example, jsdom is missing a number of features like `window.navigation` or a layout engine (`offsetTop`, etc).

Lastly, in contrast to the Web Test Runner, the Cypress test runner is more like an IDE than a test runner because you also see the real rendered component in the browser, along with its test results and logs.

Cypress has also been integrating Vite in their products: re-building their App's UI using Vitesse and using Vite to test drive their project's development.

We believe that Cypress isn't a good option for unit testing headless code, but that using Cypress (for E2E and Component Testing) and Vitest (for unit tests) would cover your app's testing needs.

Web Test Runner

@web/test-runner runs tests inside a headless browser, providing the same execution environment as your web application without the need for mocking out browser APIs or the DOM. This also makes it possible to debug inside a real browser using the devtools, although there is no UI shown for stepping through the test, as there is in Cypress tests. There is a watch mode, but it is not as intelligent as that of vitest, and may not always re-run the tests you want. To use @web/test-runner with a vite project, use @remcovaes/web-test-runner-vite-plugin. @web/test-runner does not include assertion or mocking libraries, so it is up to you to add them.

uvu

uvu is a test runner for Node.js and the browser. It runs tests in a single thread, so tests are not isolated and can leak across files. Vitest, however, uses worker threads to isolate tests and run them in parallel. For transforming your code, uvu relies on require and loader hooks. Vitest uses Vite, so files are transformed with the full power of Vite's plugin system. In a world where we have Vite providing support for the most common web tooling (typescript, JSX, most popular UI Frameworks), uvu represents a duplication of complexity. If your app is powered by Vite, having two different pipelines to configure and maintain is not justifiable. With Vitest you get to define the configuration for your dev, build and test environments as a single pipeline, sharing the same plugins and the same vite.config.js. uvu does not provide an intelligent watch mode to rerun the changed tests, but Vitest gives you amazing DX thanks to the default watch mode using Vite instant Hot Module Reload (HMR). uvu is a fast option for running simple tests, but Vitest can be faster and more reliable for more complex tests and projects.

Go to TOC

Coverage

Vitest supports Native code coverage via `c8` and instrumented code coverage via `istanbul`.

Coverage Providers

> Since Vitest v0.22.0

Both `c8` and `istanbul` support are optional. By default, `c8` will be used.

You can select the coverage tool by setting `test.coverage.provider` to either `c8` or `istanbul`:

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

export default defineConfig({
  test: {
    coverage: {
      provider: 'istanbul' // or 'c8'
    },
  },
})
```

When you start the Vitest process, it will prompt you to install the corresponding support package automatically.

Or if you prefer to install them manually:

```sh
# For c8
npm i -D @vitest/coverage-c8

# For istanbul
npm i -D @vitest/coverage-istanbul
```

Coverage Setup

To test with coverage enabled, you can pass the `--coverage` flag in CLI.

```ts
{
  "scripts": {
    "test": "vitest",
    "coverage": "vitest run --coverage"
  }
}
```

To configure it, set `test.coverage` options in your config file:

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

export default defineConfig({
```

```
  test: {
    coverage: {
      reporter: ['text', 'json', 'html'],
    },
  },
})
```

Custom Coverage Provider

It's also possible to provide your custom coverage provider by passing an object to the `test.coverage.provider`:

```
// vite.config.ts
import { defineConfig } from 'vitest/config'
import CustomCoverageProvider from 'my-custom-coverage-provider'

export default defineConfig({
  test: {
    coverage: {
      provider: CustomCoverageProvider()
    },
  },
})
```

Please refer to the type definition for more details.

Go to TOC

Debugging

Terminal

To debug a test file without an IDE, you can use `ndb`. Just add a `debugger` statement anywhere in your code, and then run `ndb`:

```
# install ndb globally
npm install -g ndb

# alternatively, with yarn
yarn global add ndb

# run tests with debugger enabled
ndb npm run test
```

VSCode

To debug a test file in VSCode, create the following launch configuration.

```
{
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug Current Test File",
      "autoAttachChildProcesses": true,
      "skipFiles": ["<node_internals>/**", "**/node_modules/**"],
      "program": "${workspaceRoot}/node_modules/vitest/vitest.mjs",
      "args": ["run", "${relativeFile}"],
      "smartStep": true,
      "console": "integratedTerminal"
    }
  ]
}
```

Then in the debug tab, ensure 'Debug Current Test File' is selected. You can then open the test file you want to debug and press F5 to start debugging.

IntelliJ IDEA

Create a 'Node.js' run configuration. Use the following settings to run all tests in debug mode:

Setting	Value
Working directory	/path/to/your-project-root
JavaScript file	./node_modules/vitest/vitest.mjs
Application parameters	run --threads false

Then run this configuration in debug mode. The IDE will stop at JS/TS breakpoints set in the editor.

Go to TOC

Test Environment

Vitest provides `environment` option to run code inside a specific environment. You can modify how environment behaves with `environmentOptions` option.

By default, you can use these environments:

- `node` is default environment
- `jsdom` emulates browser environment by providing Browser API, uses `jsdom` package
- `happy-dom` emulates browser environment by providing Browser API, and considered to be faster than jsdom, but lacks some API, uses `happy-dom` package
- `edge-runtime` emulates Vercel's edge-runtime, uses `@edge-runtime/vm` package

Starting from 0.23.0, you can create your own package to extend Vitest environment. To do so, create package with the name `vitest-environment-${name}`. That package should export an object with the shape of `Environment`:

```
import type { Environment } from 'vitest'

export default <Environment>{
  name: 'custom',
  setup() {
    // custom setup
    return {
      teardown() {
        // called after all tests with this env have been run
      }
    }
  }
}
```

You also have access to default Vitest environments through `vitest/environments` entry:

```
import { builtinEnvironments, populateGlobal } from 'vitest/environments'

console.log(builtinEnvironments) // { jsdom, happy-dom, node, edge-runtime }
```

Vitest also provides `populateGlobal` utility function, which can be used to move properties from object into the global namespace:

```
interface PopulateOptions {
  // should non-class functions be bind to the global namespace
  bindFunctions?: boolean
}

interface PopulateResult {
  // a list of all keys that were copied, even if value doesn't exist on original
object
  keys: Set<string>
  // a map of original object that might have been overriden with keys
  // you can return these values inside `teardown` function
  originals: Map<string | symbol, any>
}
```

```
export function populateGlobal(global: any, original: any, options:
PopulateOptions): PopulateResult
```

Go to TOC

Extending Matchers

Since Vitest is compatible with both Chai and Jest, you can use either the `chai.use` API or `expect.extend`, whichever you prefer.

This guide will explore extending matchers with `expect.extend`. If you are interested in Chai's API, check their guide.

To extend default matchers, call `expect.extend` with an object containing your matchers.

```
expect.extend({
  toBeFoo(received, expected) {
    const { isNot } = this
    return {
      // do not alter your "pass" based on isNot. Vitest does it for you
      pass: received === 'foo',
      message: () => `${received} is${isNot ? ' not' : ''} foo`
    }
  }
})
```

The return value of a matcher should be compatible with the following interface:

```
interface MatcherResult {
  pass: boolean
  message: () => string
  // If you pass these, they will automatically appear inside a diff when
  // the matcher does not pass, so you don't need to print the diff yourself
  actual?: unknown
  expected?: unknown
}
```

If you create an asynchronous matcher, don't forget to `await` the result (`await expect('foo').toBeFoo()`) in the test itself.

The first argument inside a matcher's function is the received value (the one inside `expect(received)`). The rest are arguments passed directly to the matcher.

Matcher function have access to `this` context with the following properties:

- `isNot`

  Returns true, if matcher was called on `not` (`expect(received).not.toBeFoo()`).

- `promise`

  If matcher was called on `resolved/rejected`, this value will contain the name of modifier. Otherwise, it will be an empty string.

- `equals`

This is a utility function that allows you to compare two values. It will return `true` if values are equal, `false` otherwise. This function is used internally for almost every matcher. It supports objects with asymmetric matchers by default.

- `utils`

  This contains a set of utility functions that you can use to display messages.

`this` context also contains information about the current test. You can also get it by calling `expect.getState()`. The most useful properties are:

- `currentTestName`

  Full name of the current test (including describe block).

- `testPath`

  Path to the current test.

Go to TOC

Features

Shared config between test, dev and build

Vite's config, transformers, resolvers, and plugins. Use the same setup from your app to run the tests.

Learn more at Configuring Vitest.

Watch Mode

```
$ vitest
```

When you modify your source code or the test files, Vitest smartly searches the module graph and only re-runs the related tests, just like how HMR works in Vite!

`vitest` starts in `watch mode` **by default in development environment** and `run mode` in CI environment (when `process.env.CI` presents) smartly. You can use `vitest watch` or `vitest run` to explicitly specify the desired mode.

Common web idioms out-of-the-box

Out-of-the-box ES Module / TypeScript / JSX support / PostCSS

Threads

Workers multi-threading via Tinypool (a lightweight fork of Piscina), allowing tests to run simultaneously. Threads are enabled by default in Vitest, and can be disabled by passing `--no-threads` in the CLI.

Vitest also isolates each file's environment so env mutations in one file don't affect others. Isolation can be disabled by passing `--no-isolate` to the CLI (trading correctness for run performance).

Test Filtering

Vitest provided many ways to narrow down the tests to run in order to speed up testing so you can focus on development.

Learn more about Test Filtering.

Running tests concurrently

Use `.concurrent` in consecutive tests to run them in parallel.

```
import { describe, it } from 'vitest'

// The two tests marked with concurrent will be run in parallel
describe('suite', () => {
```

```
  it('serial test', async () => { /* ... */ })
  it.concurrent('concurrent test 1', async () => { /* ... */ })
  it.concurrent('concurrent test 2', async () => { /* ... */ })
})
```

If you use `.concurrent` on a suite, every test in it will be run in parallel.

```
import { describe, it } from 'vitest'

// All tests within this suite will be run in parallel
describe.concurrent('suite', () => {
  it('concurrent test 1', async () => { /* ... */ })
  it('concurrent test 2', async () => { /* ... */ })
  it.concurrent('concurrent test 3', async () => { /* ... */ })
})
```

You can also use `.skip`, `.only`, and `.todo` with concurrent suites and tests. Read more in the API Reference.

Snapshot

Jest-compatible snapshot support.

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

it('renders correctly', () => {
  const result = render()
  expect(result).toMatchSnapshot()
})
```

Learn more at Snapshot.

Chai and Jest `expect` compatibility

Chai is built-in for assertions plus Jest `expect`-compatible APIs.

Notice that if you are using third-party libraries that add matchers, setting `test.globals` to `true` will provide better compatibility.

Mocking

Tinyspy is built-in for mocking with `jest`-compatible APIs on `vi` object.

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

const fn = vi.fn()

fn('hello', 1)

expect(vi.isMockFunction(fn)).toBe(true)
expect(fn.mock.calls[0]).toEqual(['hello', 1])

fn.mockImplementation(arg => arg)
```

```
fn('world', 2)

expect(fn.mock.results[1].value).toBe('world')
```

Vitest supports both happy-dom or jsdom for mocking DOM and browser APIs. They don't come with Vitest, you might need to install them:

```
$ npm i -D happy-dom
# or
$ npm i -D jsdom
```

After that, change the `environment` option in your config file:

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

export default defineConfig({
  test: {
    environment: 'happy-dom', // or 'jsdom', 'node'
  },
})
```

Learn more at Mocking.

Coverage

Vitest supports Native code coverage via `c8` and instrumented code coverage via `istanbul`.

```
{
  "scripts": {
    "test": "vitest",
    "coverage": "vitest run --coverage"
  }
}
```

Learn more at Coverage.

In-source testing

Vitest also provides a way to run tests within your source code along with the implementation, similar to Rust's module tests.

This makes the tests share the same closure as the implementations and able to test against private states without exporting. Meanwhile, it also brings the feedback loop closer for development.

```
// src/index.ts

// the implementation
export function add(...args: number[]) {
  return args.reduce((a, b) => a + b, 0)
}

// in-source test suites
if (import.meta.vitest) {
```

```
  const { it, expect } = import.meta.vitest
  it('add', () => {
    expect(add()).toBe(0)
    expect(add(1)).toBe(1)
    expect(add(1, 2, 3)).toBe(6)
  })
}
```

Learn more at In-source testing.

Benchmarking `experimental`

Since Vitest 0.23.0, you can run benchmark tests with `bench` function via Tinybench to compare performance results.

```
import { bench, describe } from 'vitest'

describe('sort', () => {
  bench('normal', () => {
    const x = [1, 5, 4, 2, 3]
    x.sort((a, b) => {
      return a - b
    })
  })

  bench('reverse', () => {
    const x = [1, 5, 4, 2, 3]
    x.reverse().sort((a, b) => {
      return a - b
    })
  })
})
```

Type Testing `experimental`

Since Vitest 0.25.0 you can write tests to catch type regressions. Vitest comes with `expect-type` package to provide you with a similar and easy to understand API.

```
import { assertType, expectTypeOf } from 'vitest'
import { mount } from './mount.js'

test('my types work properly', () => {
  expectTypeOf(mount).toBeFunction()
  expectTypeOf(mount).parameter(0).toMatchTypeOf<{ name: string }>()

  // @ts-expect-error name is a string
  assertType(mount({ name: 42 }))
})
```

Go to TOC

Test Filtering

Filtering, timeouts, concurrent for suite and tests

CLI

You can use CLI to filter test files by name:

```
$ vitest basic
```

Will only execute test files that contain `basic`, e.g.

```
basic.test.ts
basic-foo.test.ts
```

Specifying a Timeout

You can optionally pass a timeout in milliseconds as third argument to tests. The default is 5 seconds.

```ts
import { test } from 'vitest'

test('name', async () => { /* ... */ }, 1000)
```

Hooks also can receive a timeout, with the same 5 seconds default.

```ts
import { beforeAll } from 'vitest'

beforeAll(async () => { /* ... */ }, 1000)
```

Skipping Suites and Tests

Use `.skip` to avoid running certain suites or tests

```ts
import { assert, describe, it } from 'vitest'

describe.skip('skipped suite', () => {
  it('test', () => {
    // Suite skipped, no error
    assert.equal(Math.sqrt(4), 3)
  })
})

describe('suite', () => {
  it.skip('skipped test', () => {
    // Test skipped, no error
    assert.equal(Math.sqrt(4), 3)
  })
})
```

Selecting Suites and Tests to Run

Use `.only` to only run certain suites or tests

```
import { assert, describe, it } from 'vitest'

// Only this suite (and others marked with only) are run
describe.only('suite', () => {
  it('test', () => {
    assert.equal(Math.sqrt(4), 3)
  })
})

describe('another suite', () => {
  it('skipped test', () => {
    // Test skipped, as tests are running in Only mode
    assert.equal(Math.sqrt(4), 3)
  })

  it.only('test', () => {
    // Only this test (and others marked with only) are run
    assert.equal(Math.sqrt(4), 2)
  })
})
```

Unimplemented Suites and Tests

Use `.todo` to stub suites and tests that should be implemented

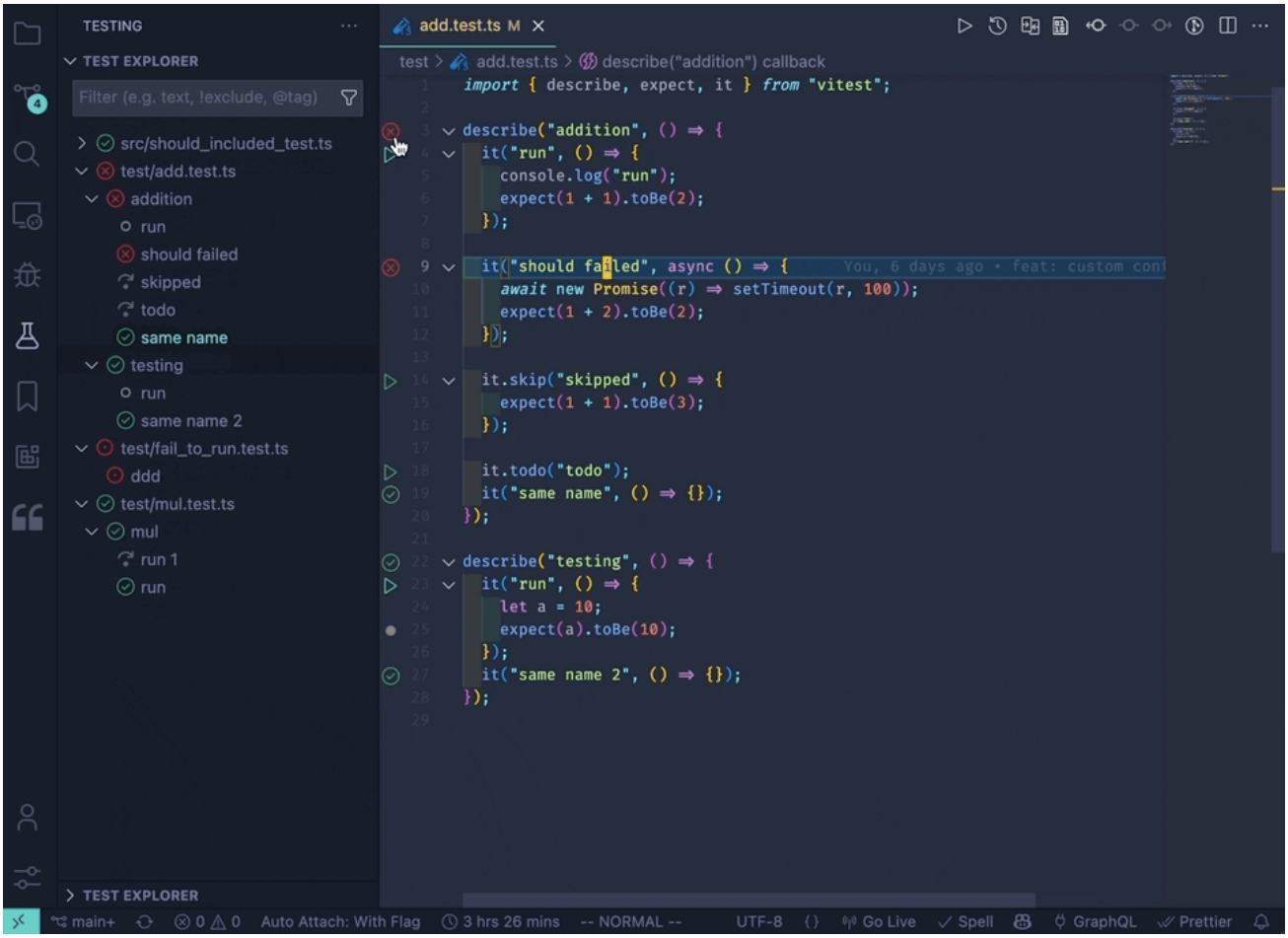
```
import { describe, it } from 'vitest'

// An entry will be shown in the report for this suite
describe.todo('unimplemented suite')

// An entry will be shown in the report for this test
describe('suite', () => {
  it.todo('unimplemented test')
})
```

Go to TOC

IDE Integrations

VS Code `Official`

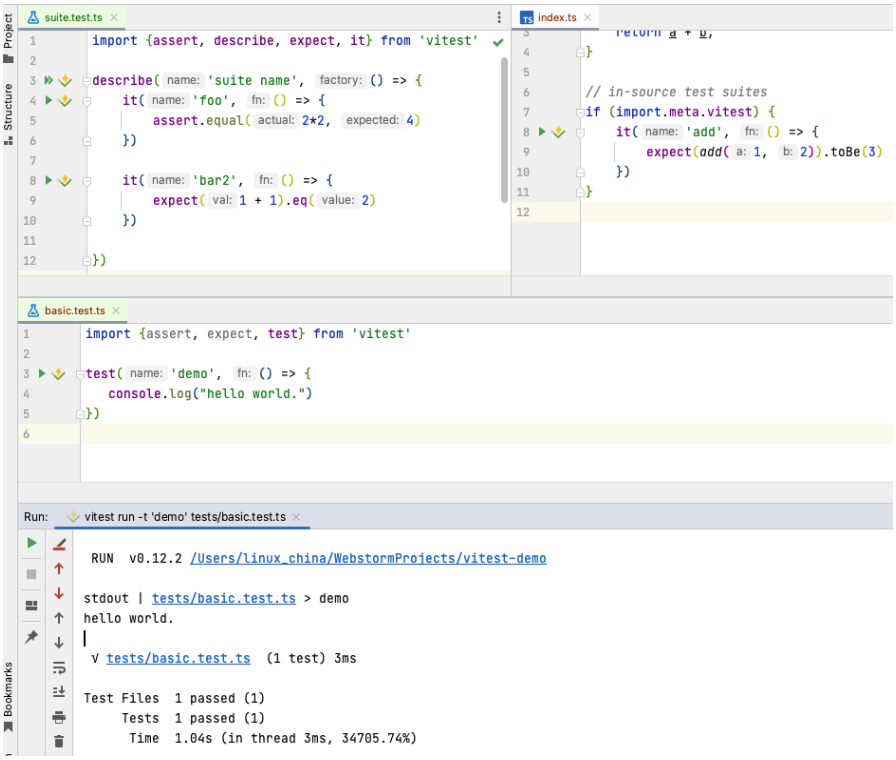
GitHub | VS Code Marketplace



IntelliJ/WebStorm `Community`

Maintained by @linux-china.

GitHub | JetBrains Plugin Marketplace



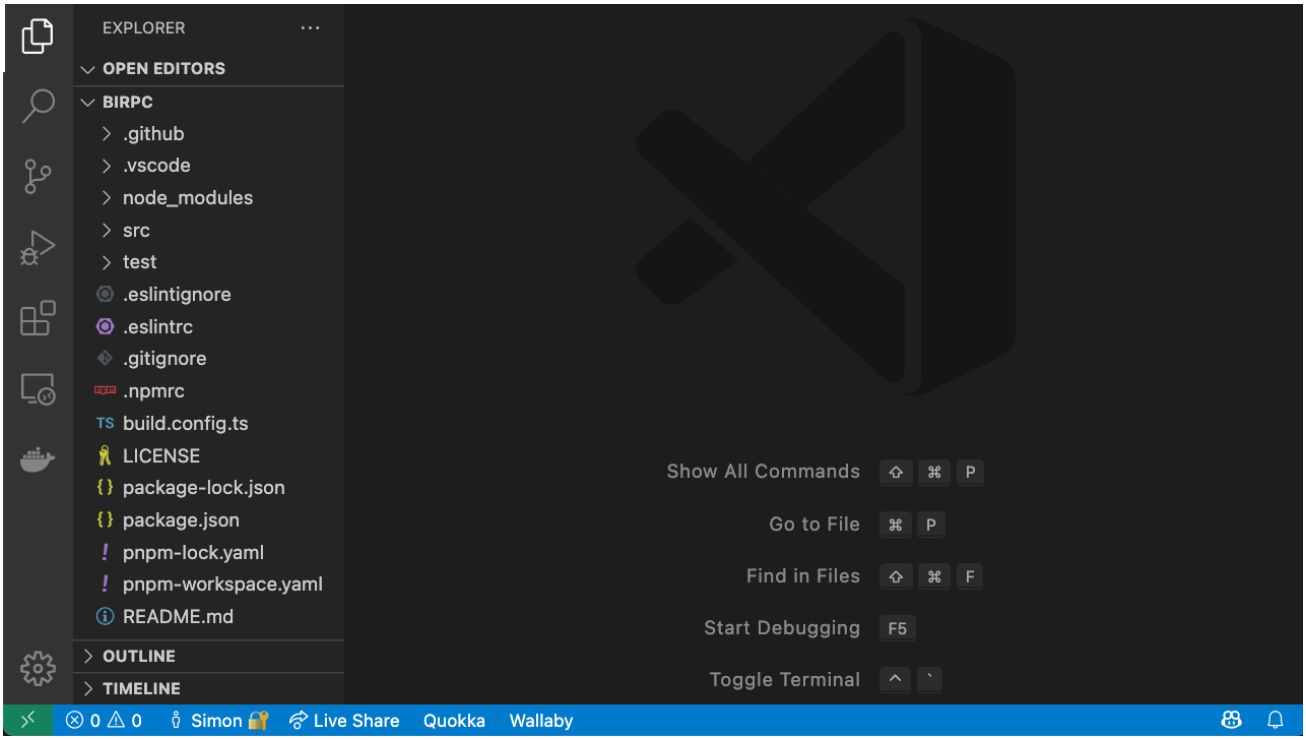
Vitest WebStorm Screenshot

Wallaby.js `Paid (free for OSS)`

Created by The Wallaby Team

Wallaby.js runs your Vitest tests immediately as you type, highlighting results in your IDE right next to your code.

VS Code | JetBrains | Visual Studio | Sublime Text


EXPLORER
OPEN EDITORS
BIRPC
.github
.vscode
node_modules
src
test
.eslintignore
.eslintrc
.gitignore
.npmrc
build.config.ts
LICENSE
package-lock.json
package.json
pnpm-lock.yaml
pnpm-workspace.yaml
README.md
OUTLINE
TIMELINE
Show All Commands
Go to File
Find in Files
Start Debugging
F5
Toggle Terminal
Simon
Live Share
Quokka
Wallaby

Go to TOC

In-source testing

Vitest also provides a way to run tests within your source code along side the implementation, similar to Rust's module tests.

This makes the tests share the same closure as the implementations and able to test against private states without exporting. Meanwhile, it also brings a closer feedback loop for development.

Setup

To get started, put a `if (import.meta.vitest)` block at the end of your source file and write some tests inside it. For example:

```ts
// src/index.ts

// the implementation
export function add(...args: number[]) {
  return args.reduce((a, b) => a + b, 0)
}

// in-source test suites
if (import.meta.vitest) {
  const { it, expect } = import.meta.vitest
  it('add', () => {
    expect(add()).toBe(0)
    expect(add(1)).toBe(1)
    expect(add(1, 2, 3)).toBe(6)
  })
}
```

Update the `includeSource` config for Vitest to grab the files under `src/`:

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

export default defineConfig({
  test: {
    includeSource: ['src/**/*.{js,ts}'],
  },
})
```

Then you can start to test!

```
$ npx vitest
```

Production build

For the production build, you will need to set the `define` options in your config file, letting the bundler do the dead code elimination. For example, in Vite

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

export default defineConfig({
+ define: {
+   'import.meta.vitest': 'undefined',
+ },
  test: {
    includeSource: ['src/**/*.{js,ts}']
  },
})
```

Other Bundlers

unbuild rollup

TypeScript

To get TypeScript support for `import.meta.vitest`, add `vitest/importMeta` to your `tsconfig.json`:

```
// tsconfig.json
{
  "compilerOptions": {
    "types": [
+     "vitest/importMeta"
    ]
  }
}
```

Reference to `test/import-meta` for the full example.

Notes

This feature could be useful for:

- Unit testing for small-scoped functions or utilities
- Prototyping
- Inline Assertion

It's recommended to **use separate test files instead** for more complex tests like components or E2E testing.

Go to TOC

Getting Started

Overview

Vitest is a blazing fast unit test framework powered by Vite.

You can learn more about the rationale behind the project in the Why Vitest section.

Trying Vitest Online

You can try Vitest online on StackBlitz. It runs Vitest directly in the browser, and it is almost identical to the local setup but doesn't require installing anything on your machine.

Adding Vitest to your Project

With npm

```
npm install -D vitest
```

or with yarn

```
yarn add -D vitest
```

or with pnpm

```
pnpm add -D vitest
```

> Vitest requires Vite >=v3.0.0 and Node >=v14

Configuring Vitest

One of the main advantages of Vitest is its unified configuration with Vite. If present, `vitest` will read your root `vite.config.ts` to match with the plugins and setup as your Vite app. For example, your Vite resolve.alias and plugins configuration will work out-of-the-box. If you want a different configuration during testing, you can:

- Create `vitest.config.ts`, which will have the higher priority
- Pass `--config` option to CLI, e.g. `vitest --config ./path/to/vitest.config.ts`
- Use `process.env.VITEST` or `mode` property on `defineConfig` (will be set to `test` if not overridden) to conditionally apply different configuration in `vite.config.ts`

To configure `vitest` itself, add `test` property in your Vite config. You'll also need to add a reference to Vitest types using a triple slash command at the top of your config file, if you are importing `defineConfig` from `vite` itself.

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

export default defineConfig({
  test: {
    // ...
  },
})
```

See the list of config options in the Config Reference

Command Line Interface

In a project where Vitest is installed, you can use the `vitest` binary in your npm scripts, or run it directly with `npx vitest`. Here are the default npm scripts in a scaffolded Vitest project:

```ts
{
  "scripts": {
    "test": "vitest",
    "coverage": "vitest run --coverage"
  }
}
```

To run tests once without watching for file changes, use `vitest run`. You can specify additional CLI options like `--port` or `--https`. For a full list of CLI options, run `npx vitest --help` in your project.

Learn more about the Command Line Interface

IDE Integrations

We also provided a official extension for Visual Studio Code to enhance your testing experience with Vitest.

Install from VS Code Marketplace

Learn more about IDE Integrations

Examples

@@include

Projects using Vitest

- unocss
- unplugin-auto-import
- unplugin-vue-components
- vitesse
- vitesse-lite
- fluent-vue
- vueuse
- milkdown
- gridjs-svelte

- spring-easing
- bytemd
- faker
- million
- Vitamin
- neodrag
- svelte-multiselect
- iconify
- tdesign-vue-next
- cz-git

Using Unreleased Commits

If you can't wait for a new release to test the latest features, you will need to clone the vitest repo to your local machine and then build and link it yourself (pnpm is required):

```
git clone https://github.com/vitest-dev/vitest.git
cd vitest
pnpm install
cd packages/vitest
pnpm run build
pnpm link --global # you can use your preferred package manager for this step
```

Then go to the project where you are using Vitest and run `pnpm link --global vitest` (or the package manager that you used to link `vitest` globally).

Community

If you have questions or need help, reach out to the community at Discord and GitHub Discussions.

Go to TOC

Migration Guide

Migrating from Jest

Vitest has been designed with a Jest compatible API, in order to make the migration from Jest as simple as possible. Despite those efforts, you may still run into the following differences:

Globals as a Default

Jest has their globals API enabled by default. Vitest does not. You can either enable globals via the `globals` configuration setting or update your code to use imports from the `vitest` module instead.

If you decide to keep globals disabled, be aware that common libraries like `testing-library` will not run auto DOM cleanup.

Module mocks

When mocking a module in Jest, the factory argument's return value is the default export. In Vitest, the factory argument has to return an object with each export explicitly defined. For example, the following `jest.mock` would have to be updated as follows:

```
- jest.mock('./some-path', () => 'hello')
+ vi.mock('./some-path', () => ({
+   default: 'hello',
+ })
```

For more details please refer to the vi.mock api

Auto-Mocking Behaviour

Unlike Jest, mocked modules in `<root>/__mocks__` are not loaded unless `vi.mock()` is called. If you need them to be mocked in every test, like in Jest, you can mock them inside `setupFiles`.

Jasmine API

Jest exports various `jasmine` globals (such as `jasmine.any()`). Any such instances will need to be migrated to their Vitest counterparts.

Envs

Just like Jest, Vitest sets `NODE_ENV` to `test`, if it wasn't set before. Vitest also has a counterpart for `JEST_WORKER_ID` called `VITEST_POOL_ID` (always less than or equal to `maxThreads`), so if you rely on it, don't forget to rename it. Vitest also exposes `VITEST_WORKER_ID` which is a unique ID of a running worker - this number is not affected by `maxThreads`, and will increase with each created worker.

Done Callback

From Vitest v0.10.0, the callback style of declaring tests is deprecated. You can rewrite them to use `async`/`await` functions, or use Promise to mimic the callback style.

```
- it('should work', (done) => {
+ it('should work', () => new Promise(done => {
    // ...
    done()
- })
+ }))
```

Hooks

`beforeAll` / `beforeEach` hooks may return teardown function in Vitest. Because of that you may need to rewrite your hooks declarations, if they return something other than `undefined` or `null`:

```
- beforeEach(() => setActivePinia(createTestingPinia()))
+ beforeEach(() => { setActivePinia(createTestingPinia()) })
```

Types

Vitest doesn't expose a lot of types on `Vi` namespace, it exists mainly for compatibility with matchers, so you might need to import types directly from `vitest` instead of relying on `Vi` namespace:

```
- let fn: jest.Mock<string, [string]>
+ import type { Mock } from 'vitest'
+ let fn: Mock<[string], string>
```

Also, Vitest has `Args` type as a first argument instead of `Returns`, as you can see in diff.

Timers

Vitest doesn't support Jest's legacy timers.

Vue Snapshots

This is not a Jest-specific feature, but if you previously were using Jest with vue-cli preset, you will need to install `jest-serializer-vue` package, and use it inside setupFiles:

```
import vueSnapshotSerializer from 'jest-serializer-vue'

// Add Snapshot Serializer
expect.addSnapshotSerializer(vueSnapshotSerializer)
```

Otherwise your snapshots will have a lot of escaped `"` characters.

Go to TOC

Mocking

When writing tests it's only a matter of time before you need to create a "fake" version of an internal — or external — service. This is commonly referred to as **mocking**. Vitest provides utility functions to help you out through its **vi** helper. You can `import { vi } from 'vitest'` or access it **globally** (when global configuration is **enabled**).

> Always remember to clear or restore mocks before or after each test run to undo mock state changes between runs! See [`mockReset`](../api.html#mockreset) docs for more info.

If you wanna dive in head first, check out the API section otherwise keep reading to take a deeper dive into the world of mocking.

Dates

Sometimes you need to be in control of the date to ensure consistency when testing. Vitest uses `@sinonjs/fake-timers` package for manipulating timers, as well as system date. You can find more about the specific API in detail here.

Example

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

const businessHours = [9, 17]

const purchase = () => {
  const currentHour = new Date().getHours()
  const [open, close] = businessHours

  if (currentHour > open && currentHour < close)
    return { message: 'Success' }

  return { message: 'Error' }
}

describe('purchasing flow', () => {
  beforeEach(() => {
    // tell vitest we use mocked time
    vi.useFakeTimers()
  })

  afterEach(() => {
    // restoring date after each test run
    vi.useRealTimers()
  })

  it('allows purchases within business hours', () => {
    // set hour within business hours
    const date = new Date(2000, 1, 1, 13)
    vi.setSystemTime(date)

    // access Date.now() will result in the date set above
```

```
    expect(purchase()).toEqual({ message: 'Success' })
  })

  it('disallows purchases outside of business hours', () => {
    // set hour outside business hours
    const date = new Date(2000, 1, 1, 19)
    vi.setSystemTime(date)

    // access Date.now() will result in the date set above
    expect(purchase()).toEqual({ message: 'Error' })
  })
})
```

Functions

Mocking functions can be split up into two different categories; *spying* & *mocking*.

Sometimes all you need is to validate whether or not a specific function has been called (and possibly which arguments were passed). In these cases a spy would be all we need which you can use directly with `vi.spyOn()` (read more here).

However spies can only help you **spy** on functions, they are not able to alter the implementation of those functions. In the case where we do need to create a fake (or mocked) version of a function we can use `vi.fn()` (read more here).

We use Tinyspy as a base for mocking functions, but we have our own wrapper to make it `jest` compatible. Both `vi.fn()` and `vi.spyOn()` share the same methods, however only the return result of `vi.fn()` is callable.

Example

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

const getLatest = (index = messages.items.length - 1) => messages.items[index]

const messages = {
  items: [
    { message: 'Simple test message', from: 'Testman' },
    // ...
  ],
  getLatest, // can also be a `getter or setter if supported`
}

describe('reading messages', () => {
  afterEach(() => {
    vi.restoreAllMocks()
  })

  it('should get the latest message with a spy', () => {
    const spy = vi.spyOn(messages, 'getLatest')
    expect(spy.getMockName()).toEqual('getLatest')

    expect(messages.getLatest()).toEqual(
      messages.items[messages.items.length - 1],
    )

    expect(spy).toHaveBeenCalledTimes(1)
```

```
    spy.mockImplementationOnce(() => 'access-restricted')
    expect(messages.getLatest()).toEqual('access-restricted')

    expect(spy).toHaveBeenCalledTimes(2)
  })

  it('should get with a mock', () => {
    const mock = vi.fn().mockImplementation(getLatest)

    expect(mock()).toEqual(messages.items[messages.items.length - 1])
    expect(mock).toHaveBeenCalledTimes(1)

    mock.mockImplementationOnce(() => 'access-restricted')
    expect(mock()).toEqual('access-restricted')

    expect(mock).toHaveBeenCalledTimes(2)

    expect(mock()).toEqual(messages.items[messages.items.length - 1])
    expect(mock).toHaveBeenCalledTimes(3)
  })
})
```

More

- Jest's Mock Functions

Globals

You can mock global variables that are not present with `jsdom` or `node` by using `vi.stubGlobal` helper. It will put the value of the global variable into a `globalThis` object.

```
import { vi } from 'vitest'

const IntersectionObserverMock = vi.fn(() => ({
  disconnect: vi.fn(),
  observe: vi.fn(),
  takeRecords: vi.fn(),
  unobserve: vi.fn(),
}))

vi.stubGlobal('IntersectionObserver', IntersectionObserverMock)

// now you can access it as `IntersectionObserver` or
`window.IntersectionObserver`
```

Modules

Mock modules observe third-party-libraries, that are invoked in some other code, allowing you to test arguments, output or even redeclare its implementation.

See the `vi.mock()` api section for a more in-depth detailed API description.

Automocking algorithm

If your code is importing a mocked module, without any associated `__mocks__` file or `factory` for this module, Vitest will mock the module itself by invoking it and mocking every export.

The following principles apply

- All arrays will be emptied
- All primitives and collections will stay the same
- All objects will be deeply cloned
- All instances of classes and their prototypes will be deeply cloned

Example

```
import { afterEach, beforeEach, describe, expect, it, vi } from 'vitest'
import { Client } from 'pg'
import { failure, success } from './handlers'

// handlers
export function success(data) {}
export function failure(data) {}

// get todos
export const getTodos = async (event, context) => {
  const client = new Client({
    // ...clientOptions
  })

  await client.connect()

  try {
    const result = await client.query('SELECT * FROM todos;')

    client.end()

    return success({
      message: `${result.rowCount} item(s) returned`,
      data: result.rows,
      status: true,
    })
  }
  catch (e) {
    console.error(e.stack)

    client.end()

    return failure({ message: e, status: false })
  }
}

vi.mock('pg', () => {
  const Client = vi.fn()
  Client.prototype.connect = vi.fn()
  Client.prototype.query = vi.fn()
  Client.prototype.end = vi.fn()

  return { Client }
})

vi.mock('./handlers', () => {
  return {
    success: vi.fn(),
    failure: vi.fn(),
  }
})
```

```
describe('get a list of todo items', () => {
  let client

  beforeEach(() => {
    client = new Client()
  })

  afterEach(() => {
    vi.clearAllMocks()
  })

  it('should return items successfully', async () => {
    client.query.mockResolvedValueOnce({ rows: [], rowCount: 0 })

    await getTodos()

    expect(client.connect).toBeCalledTimes(1)
    expect(client.query).toBeCalledWith('SELECT * FROM todos;')
    expect(client.end).toBeCalledTimes(1)

    expect(success).toBeCalledWith({
      message: '0 item(s) returned',
      data: [],
      status: true,
    })
  })

  it('should throw an error', async () => {
    const mError = new Error('Unable to retrieve rows')
    client.query.mockRejectedValueOnce(mError)

    await getTodos()

    expect(client.connect).toBeCalledTimes(1)
    expect(client.query).toBeCalledWith('SELECT * FROM todos;')
    expect(client.end).toBeCalledTimes(1)
    expect(failure).toBeCalledWith({ message: mError, status: false })
  })
})
```

Requests

Because Vitest runs in Node, mocking network requests is tricky; web APIs are not available, so we need something that will mimic network behavior for us. We recommend Mock Service Worker to accomplish this. It will let you mock both `REST` and `GraphQL` network requests, and is framework agnostic.

Mock Service Worker (MSW) works by intercepting the requests your tests make, allowing you to use it without changing any of your application code. In-browser, this uses the Service Worker API. In Node.js, and for Vitest, it uses node-request-interceptor. To learn more about MSW, read their introduction

Configuration

You can use it like below in your setup file

```
import { afterAll, afterEach, beforeAll } from 'vitest'
import { setupServer } from 'msw/node'
import { graphql, rest } from 'msw'
```

```
const posts = [
  {
    userId: 1,
    id: 1,
    title: 'first post title',
    body: 'first post body',
  },
  // ...
]

export const restHandlers = [
  rest.get('https://rest-endpoint.example/path/to/posts', (req, res, ctx) => {
    return res(ctx.status(200), ctx.json(posts))
  }),
]

const graphqlHandlers = [
  graphql.query('https://graphql-endpoint.example/api/v1/posts', (req, res, ctx)
=> {
    return res(ctx.data(posts))
  }),
]

const server = setupServer(...restHandlers, ...graphqlHandlers)

// Start server before all tests
beforeAll(() => server.listen({ onUnhandledRequest: 'error' }))

//  Close server after all tests
afterAll(() => server.close())

// Reset handlers after each test `important for test isolation`
afterEach(() => server.resetHandlers())
```

Configuring the server with `onUnhandleRequest: 'error'` ensures that an error is thrown whenever there is a request that does not have a corresponding request handler.

Example

We have a full working example which uses MSW: React Testing with MSW.

More

There is much more to MSW. You can access cookies and query parameters, define mock error responses, and much more! To see all you can do with MSW, read their documentation.

Timers

Whenever we test code that involves timeouts or intervals, instead of having our tests wait it out or timeout. We can speed up our tests by using "fake" timers by mocking calls to `setTimeout` and `setInterval`, too.

See the `vi.usefaketimers` api section for a more in depth detailed API description.

Example

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

const executeAfterTwoHours = (func) => {
  setTimeout(func, 1000 * 60 * 60 * 2) // 2 hours
}

const executeEveryMinute = (func) => {
  setInterval(func, 1000 * 60) // 1 minute
}

const mock = vi.fn(() => console.log('executed'))

describe('delayed execution', () => {
  beforeEach(() => {
    vi.useFakeTimers()
  })
  afterEach(() => {
    vi.restoreAllMocks()
  })
  it('should execute the function', () => {
    executeAfterTwoHours(mock)
    vi.runAllTimers()
    expect(mock).toHaveBeenCalledTimes(1)
  })
  it('should not execute the function', () => {
    executeAfterTwoHours(mock)
    // advancing by 2ms won't trigger the func
    vi.advanceTimersByTime(2)
    expect(mock).not.toHaveBeenCalled()
  })
  it('should execute every minute', () => {
    executeEveryMinute(mock)
    vi.advanceTimersToNextTimer()
    expect(mock).toHaveBeenCalledTimes(1)
    vi.advanceTimersToNextTimer()
    expect(mock).toHaveBeenCalledTimes(2)
  })
})
```

Cheat Sheet

info `vi` in the examples below is imported directly from `vitest`. You can also use it globally, if you set `globals` to `true` in your [config](/config/).

I want to…

- Spy on a `method`

```
const instance = new SomeClass()
vi.spyOn(instance, 'method')
```

- Mock exported variables

```
// some-path.ts
export const getter = 'variable'
```

```ts
// some-path.test.ts
import * as exports from 'some-path'
vi.spyOn(exports, 'getter', 'get').mockReturnValue('mocked')
```

- Mock exported function

Example with `vi.mock`:

```ts
// some-path.ts
export function method() {}
```

```ts
import { method } from 'some-path'
vi.mock('some-path', () => ({
  method: vi.fn()
}))
```

Example with `vi.spyOn`:

```ts
import * as exports from 'some-path'
vi.spyOn(exports, 'method').mockImplementation(() => {})
```

- Mock exported class implementation

Example with `vi.mock` and prototype:

```ts
// some-path.ts
export class SomeClass {}
```

```ts
import { SomeClass } from 'some-path'
vi.mock('some-path', () => {
  const SomeClass = vi.fn()
  SomeClass.prototype.someMethod = vi.fn()
  return { SomeClass }
})
// SomeClass.mock.instances will have SomeClass
```

Example with `vi.mock` and return value:

```ts
import { SomeClass } from 'some-path'
vi.mock('some-path', () => {
  const SomeClass = vi.fn(() => ({
    someMethod: vi.fn()
  }))
  return { SomeClass }
})
// SomeClass.mock.returns will have returned object
```

Example with `vi.spyOn`:

```ts
import * as exports from 'some-path'
vi.spyOn(exports, 'SomeClass').mockImplementation(() => {
  // whatever suites you from first two examples
})
```

- Spy on an object returned from a function

Example using cache:

```ts
// some-path.ts
export function useObject() {
  return { method: () => true }
}
```

```ts
// useObject.js
import { useObject } from 'some-path'
const obj = useObject()
obj.method()
```

```ts
// useObject.test.js
import { useObject } from 'some-path'
vi.mock('some-path', () => {
  let _cache
  const useObject = () => {
    if (!_cache) {
      _cache = {
        method: vi.fn(),
      }
    }
    // now everytime useObject() is called it will
    // return the same object reference
    return _cache
  }
  return { useObject }
})

const obj = useObject()
// obj.method was called inside some-path
expect(obj.method).toHaveBeenCalled()
```

- Mock part of a module

```ts
import { mocked, original } from 'some-path'
vi.mock('some-path', async () => {
  const mod = await vi.importActual<typeof import('some-path')>('some-path')
  return {
    ...mod,
    mocked: vi.fn()
  }
})
original() // has original behaviour
mocked() // is a spy function
```

- Mock current date

```ts
const mockDate = new Date(2022, 0, 1)
vi.setSystemTime(mockDate)
const now = new Date()
expect(now.valueOf()).toBe(mockDate.valueOf())
```

- Mock global variable

```ts
vi.stubGlobal('__VERSION__', '1.0.0')
expect(__VERSION__).toBe('1.0.0')
```

Go to TOC

Snapshot

Snapshot tests are a very useful tool whenever you want to make sure the output of your functions does not change unexpectedly.

When using snapshot, Vitest will take a snapshot of the given value, then compares it to a reference snapshot file stored alongside the test. The test will fail if the two snapshots do not match: either the change is unexpected, or the reference snapshot needs to be updated to the new version of the result.

Use Snapshots

To snapshot a value, you can use the `toMatchSnapshot()` from `expect()` API:

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

it('toUpperCase', () => {
  const result = toUpperCase('foobar')
  expect(result).toMatchSnapshot()
})
```

The first time this test is run, Vitest creates a snapshot file that looks like this:

```
// Vitest Snapshot v1

exports['toUpperCase 1'] = '"FOOBAR"'
```

The snapshot artifact should be committed alongside code changes, and reviewed as part of your code review process. On subsequent test runs, Vitest will compare the rendered output with the previous snapshot. If they match, the test will pass. If they don't match, either the test runner found a bug in your code that should be fixed, or the implementation has changed and the snapshot needs to be updated.

Inline Snapshots

Similarly, you can use the `toMatchInlineSnapshot()` to store the snapshot inline within the test file.

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

it('toUpperCase', () => {
  const result = toUpperCase('foobar')
  expect(result).toMatchInlineSnapshot()
})
```

Instead of creating a snapshot file, Vitest will modify the test file directory to update the snapshot as a string:

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

it('toUpperCase', () => {
```

```
  const result = toUpperCase('foobar')
  expect(result).toMatchInlineSnapshot('"FOOBAR"')
})
```

This allows you to see the expected output directly without jumping across different files.

Updating Snapshots

When the received value doesn't match the snapshot, the test fails and shows you the difference between them. When the snapshot change is expected, you may want to update the snapshot from the current state.

In watch mode, you can press the `u` key in the terminal to update the failed snapshot directly.

Or you can use the `--update` or `-u` flag in the CLI to make Vitest update snapshots.

```
vitest -u
```

Image Snapshots

It's also possible to snapshot images using `jest-image-snapshot`.

```
npm i -D jest-image-snapshot
```

```
test('image snapshot', () => {
  expect(readFileSync('./test/stubs/input-image.png'))
    .toMatchImageSnapshot()
})
```

You can learn more in the `examples/image-snapshot` example.

Custom Serializer

You can add your own logic to alter how your snapshots are serialized. Like Jest, Vitest has default serializers for built-in JavaScript types, HTML elements, ImmutableJS and for React elements.

Example serializer module:

```
expect.addSnapshotSerializer({
  serialize(val, config, indentation, depth, refs, printer) {
    // `printer` is a function that serializes a value using existing plugins.
    return `Pretty foo: ${printer(val.foo)}`
  },
  test(val) {
    return val && Object.prototype.hasOwnProperty.call(val, 'foo')
  },
})
```

After adding a test like this:

```
test('foo snapshot test', () => {
  const bar = {
    foo: {
      x: 1,
```

```
      y: 2,
    },
  }

  expect(bar).toMatchSnapshot()
})
```

You will get the following snapshot:

```
Pretty foo: Object {
  "x": 1,
  "y": 2,
}
```

We are using Jest's `pretty-format` for serializing snapshots. You can read more about it here: pretty-format.

Difference from Jest

Vitest provides an almost compatible Snapshot feature with Jest's with a few exceptions:

1. Comment header in the snapshot file is different

```
- // Jest Snapshot v1
+ // Vitest Snapshot v1
```

This does not really affect the functionality but might affect your commit diff when migrating from Jest.

2. `printBasicPrototype` is default to `false`

Both Jest and Vitest's snapshots are powered by `pretty-format`. In Vitest we set `printBasicPrototype` default to `false` to provide a cleaner snapshot output, while in Jest it's `true` by default.

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

test('snapshot', () => {
  const bar = [
    {
      foo: 'bar',
    },
  ]

  // in Jest
  expect(bar).toMatchInlineSnapshot(`
    Array [
      Object {
        "foo": "bar",
      },
    ]
  `)

  // in Vitest
  expect(bar).toMatchInlineSnapshot(`
    [
      {
        "foo": "bar",
```

```
      },
    ]
  `)
})
```

We believe this is a more reasonable default for readability and overall DX. If you still prefer Jest's behavior, you can change your config:

```
// vitest.config.js
export default defineConfig({
  test: {
    snapshotFormat: {
      printBasicPrototype: true
    }
  }
})
```

Go to TOC

Test Context

Inspired by Playwright Fixtures, Vitest's test context allows you to define utils, states, and fixtures that can be used in your tests.

Usage

The first argument for each test callback is a test context.

```
import { it } from 'vitest'

it('should work', (ctx) => {
  // prints name of the test
  console.log(ctx.meta.name)
})
```

Built-in Test Context

`context.meta`

A readonly object containing metadata about the test.

`context.expect`

The `expect` API bound to the current test.

Extend Test Context

The contexts are different for each test. You can access and extend them within the `beforeEach` and `afterEach` hooks.

```
import { beforeEach, it } from 'vitest'

beforeEach(async (context) => {
  // extend context
  context.foo = 'bar'
})

it('should work', ({ foo }) => {
  console.log(foo) // 'bar'
})
```

TypeScript

To provide property types for all your custom contexts, you can aggregate the `TestContext` type by adding

```ts
declare module 'vitest' {
  export interface TestContext {
    foo?: string
  }
}
```

If you want to provide property types only for specific `beforeEach`, `afterEach`, `it` and `test` hooks, you can pass the type as a generic.

```ts
interface LocalTestContext {
  foo: string
}

beforeEach<LocalTestContext>(async (context) => {
  // typeof context is 'TestContext & LocalTestContext'
  context.foo = 'bar'
})

it<LocalTestContext>('should work', ({ foo }) => {
  // typeof foo is 'string'
  console.log(foo) // 'bar'
})
```<span style='float: footnote;'><a href="../index.html#toc">Go to TOC</a></span>

Testing Types

Vitest allows you to write tests for your types, using `expectTypeOf` or `assertType` syntaxes. By default all tests inside `*.test-d.ts` files are considered type tests, but you can change it with `typecheck.include` config option.

Under the hood Vitest calls `tsc` or `vue-tsc`, depending on your config, and parses results. Vitest will also print out type errors in your source code, if it finds any. You can disable it with `typecheck.ignoreSourceErrors` config option.

Keep in mind that Vitest doesn't run or compile these files, they are only statically analyzed by the compiler, and because of that you cannot use any dynamic statements. Meaning, you cannot use dynamic test names, and `test.each`, `test.runIf`, `test.skipIf`, `test.each`, `test.concurrent` APIs. But you can use other APIs, like `test`, `describe`, `.only`, `.skip` and `.todo`.

Using CLI flags, like `--allowOnly` and `-t` are also supported for type checking.

```ts
import { assertType, expectTypeOf } from 'vitest'
import { mount } from './mount.js'

test('my types work properly', () => {
  expectTypeOf(mount).toBeFunction()
  expectTypeOf(mount).parameter(0).toMatchTypeOf<{ name: string }>()

  // @ts-expect-error name is a string
  assertType(mount({ name: 42 }))
})
```

Any type error triggered inside a test file will be treated as a test error, so you can use any type trick you want to test types of your project.

You can see a list of possible matchers in API section.

Reading Errors

If you are using `expectTypeOf` API, you might notice hard to read errors or unexpected:

```ts
expectTypeOf(1).toEqualTypeOf<string>()
//              ^^^^^^^^^^^^^^^^^^^^^^^
// index-c3943160.d.ts(90, 20): Arguments for the rest parameter 'MISMATCH' were
not provided.
```

This is due to how `expect-type` handles type errors.

Unfortunately, TypeScript doesn't provide type metadata without patching, so we cannot provide useful error messages at this point, but there are works in TypeScript project to fix this. If you want better messages, please, ask TypeScript team to have a look at mentioned PR.

If you find it hard working with `expectTypeOf` API and figuring out errors, you can always use more simple `assertType` API:

```
const answer = 42

assertType<number>(answer)
// @ts-expect-error answer is not a string
assertType<string>(answer)
```

When using `@ts-expect-error` syntax, you might want to make sure that you didn't make a typo. You can do that by including your type files in [`test.include`](../config.html#include) config option, so Vitest will also actually *run* these tests and fail with `ReferenceError`.

This will pass, because it expects an error, but the word “answer” has a typo, so it's a false positive error:

```
// @ts-expect-error answer is not a string
assertType<string>(answr) //
```

Run typechecking

Add this command to your `scripts` section in `package.json`:

```
{
  "scripts": {
    "typecheck": "vitest typecheck"
  }
}
```

Now you can run typecheck:

```
# npm
npm run typecheck

# yarn
yarn typecheck

# pnpm
pnpm run typecheck
```

Vitest uses `tsc --noEmit` or `vue-tsc --noEmit`, depending on your configuration, so you can remove these scripts from your pipeline.

Go to TOC

Vitest UI

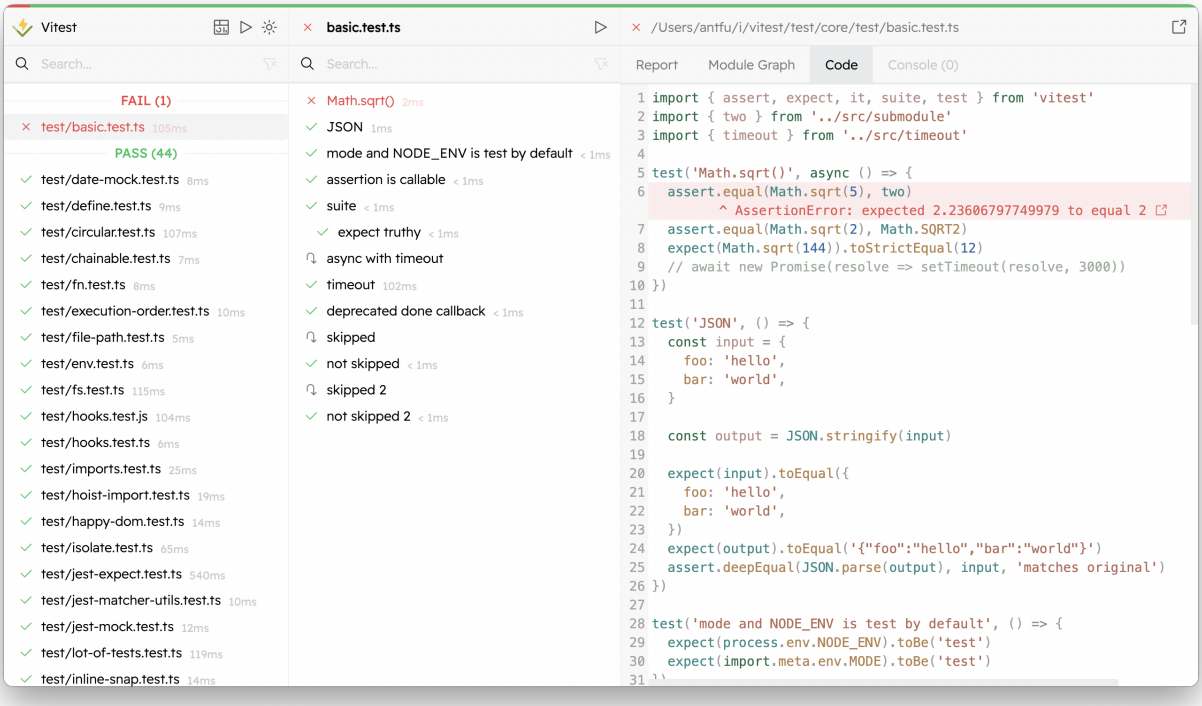
Powered by Vite, Vitest also has a dev server under the hood when running the tests. This allows Vitest to provide a beautiful UI to view and interact with your tests. The Vitest UI is optional, so you'll need to install it with:

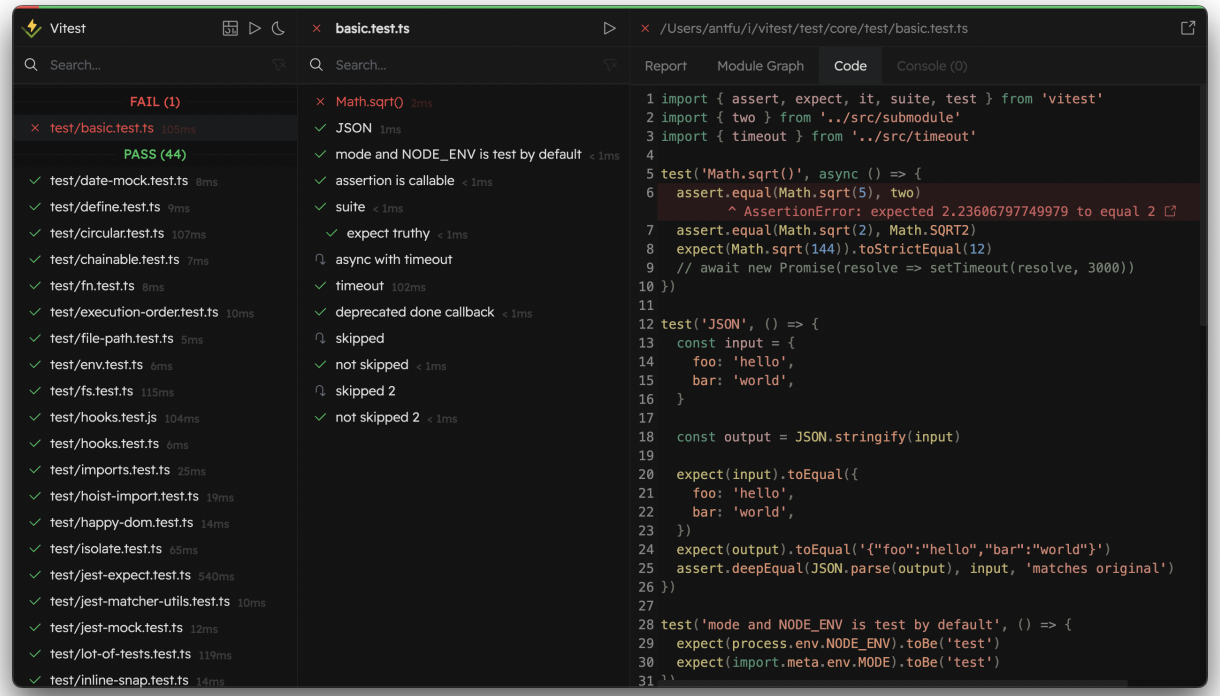
```
npm i -D @vitest/ui
```

Then you can start the tests with UI by passing the `--ui` flag:

```
vitest --ui
```

Then you can visit the Vitest UI at `http://localhost:51204/__vitest__/`


Vitest
basic.test.ts
/Users/antfu/i/vitest/test/core/test/basic.test.ts
Search...
Search...
Report
Module Graph
Code
Console (0)
FAIL (1)
test/basic.test.ts 105ms
PASS (44)
test/date-mock.test.ts 8ms
test/define.test.ts 9ms
test/circular.test.ts 107ms
test/chainable.test.ts 7ms
test/fn.test.ts 8ms
test/execution-order.test.ts 10ms
test/file-path.test.ts 5ms
test/env.test.ts 6ms
test/fs.test.ts 115ms
test/hooks.test.js 104ms
test/hooks.test.ts 6ms
test/imports.test.ts 25ms
test/hoist-import.test.ts 19ms
test/happy-dom.test.ts 14ms
test/isolate.test.ts 65ms
test/jest-expect.test.ts 540ms
test/jest-matcher-utils.test.ts 10ms
test/jest-mock.test.ts 12ms
test/lot-of-tests.test.ts 119ms
test/inline-snap.test.ts 14ms
Math.sqrt() 2ms
JSON 1ms
mode and NODE_ENV is test by default 1ms
assertion is callable 1ms
suite 1ms
expect truthy 1ms
async with timeout
timeout 102ms
deprecated done callback 1ms
skipped
not skipped 1ms
skipped 2
not skipped 2 1ms
1 import { assert, expect, it, suite, test } from 'vitest'
2 import { two } from '../src/submodule'
3 import { timeout } from '../src/timeout'
4
5 test('Math.sqrt()', async () => {
6 assert.equal(Math.sqrt(5), two)
^ AssertionError: expected 2.23606797749979 to equal 2
7 assert.equal(Math.sqrt(2), Math.SQRT2)
8 expect(Math.sqrt(144)).toStrictEqual(12)
9 // await new Promise(resolve => setTimeout(resolve, 3000))
10 })
11
12 test('JSON', () => {
13 const input = {
14 foo: 'hello',
15 bar: 'world',
16 }
17
18 const output = JSON.stringify(input)
19
20 expect(input).toEqual({
21 foo: 'hello',
22 bar: 'world',
23 })
24 expect(output).toEqual('{"foo":"hello","bar":"world"}')
25 assert.deepEqual(JSON.parse(output), input, 'matches original')
26 })
27
28 test('mode and NODE_ENV is test by default', () => {
29 expect(process.env.NODE_ENV).toBe('test')
30 expect(import.meta.env.MODE).toBe('test')
31 })


Vitest
basic.test.ts
/Users/antfu/i/vitest/test/core/test/basic.test.ts
Search...
Search...
Report
Module Graph
Code
Console (0)
FAIL (1)
test/basic.test.ts 105ms
PASS (44)
test/date-mock.test.ts 8ms
test/define.test.ts 9ms
test/circular.test.ts 107ms
test/chainable.test.ts 7ms
test/fn.test.ts 8ms
test/execution-order.test.ts 10ms
test/file-path.test.ts 5ms
test/env.test.ts 6ms
test/fs.test.ts 115ms
test/hooks.test.js 104ms
test/hooks.test.ts 6ms
test/imports.test.ts 25ms
test/hoist-import.test.ts 19ms
test/happy-dom.test.ts 14ms
test/isolate.test.ts 65ms
test/jest-expect.test.ts 540ms
test/jest-matcher-utils.test.ts 10ms
test/jest-mock.test.ts 12ms
test/lot-of-tests.test.ts 119ms
test/inline-snap.test.ts 14ms
Math.sqrt() 2ms
JSON 1ms
mode and NODE_ENV is test by default 1ms
assertion is callable 1ms
suite 1ms
expect truthy 1ms
async with timeout
timeout 102ms
deprecated done callback 1ms
skipped
not skipped 1ms
skipped 2
not skipped 2 1ms
1 import { assert, expect, it, suite, test } from 'vitest'
2 import { two } from '../src/submodule'
3 import { timeout } from '../src/timeout'
4
5 test('Math.sqrt()', async () => {
6 assert.equal(Math.sqrt(5), two)
^ AssertionError: expected 2.23606797749979 to equal 2
7 assert.equal(Math.sqrt(2), Math.SQRT2)
8 expect(Math.sqrt(144)).toStrictEqual(12)
9 // await new Promise(resolve => setTimeout(resolve, 3000))
10 })
11
12 test('JSON', () => {
13 const input = {
14 foo: 'hello',
15 bar: 'world',
16 }
17
18 const output = JSON.stringify(input)
19
20 expect(input).toEqual({
21 foo: 'hello',
22 bar: 'world',
23 })
24 expect(output).toEqual('{"foo":"hello","bar":"world"}')
25 assert.deepEqual(JSON.parse(output), input, 'matches original')
26 })
27
28 test('mode and NODE_ENV is test by default', () => {
29 expect(process.env.NODE_ENV).toBe('test')
30 expect(import.meta.env.MODE).toBe('test')
31 })

Go to TOC

Using Plugins

Vitest can be extended using plugins...

Go to TOC

Why Vitest

NOTE This guide assumes that you are familiar with Vite. A good way to start learning more is to read the [Why Vite Guide](https://vitejs.dev/guide/why.html), and [Next generation frontend tooling with ViteJS](https://www.youtube.com/watch?v=UJypSr8IhKY), a stream where [Evan You](https://twitter.com/youyuxi) did a demo explaining the main concepts.

The need for a Vite native test runner

Vite's out-of-the-box support for common web patterns, features like glob imports and SSR primitives, and its many plugins and integrations are fostering a vibrant ecosystem. Its dev and build story are key to its success. For docs, there are several SSG-based alternatives powered by Vite. Vite's Unit Testing story hasn't been clear though. Existing options like Jest were created in a different context. There is a lot of duplication between Jest and Vite, forcing users to configure two different pipelines.

Using Vite dev server to transform your files during testing, enables the creation of a simple runner that doesn't need to deal with the complexity of transforming source files and can solely focus on providing the best DX during testing. A test runner that uses the same configuration of your App (through `vite.config.js`), sharing a common transformation pipeline during dev, build, and test time. That is extensible with the same plugin API that lets you and the maintainers of your tools provide first-class integration with Vite. A tool that is built with Vite in mind from the start, taking advantage of its improvements in DX, like its instant Hot Module Reload (HMR). This is Vitest, a blazing fast unit-test framework powered by Vite.

Given Jest's massive adoption, Vitest provides a compatible API that allows you to use it as a drop-in replacement in most projects. It also includes the most common features required when setting up your unit tests (mocking, snapshots, coverage). Vitest cares a lot about performance and uses Worker threads to run as much as possible in parallel. Some ports have seen test running an order of magnitude faster. Watch mode is enabled by default, aligning itself with the way Vite pushes for a dev first experience. Even with all these improvements in DX, Vitest stays lightweight by carefully choosing its dependencies (or directly inlining needed pieces).

Vitest aims to position itself as the Test Runner of choice for Vite projects, and as a solid alternative even for projects not using Vite.

Continue reading in the Getting Started Guide

How is Vitest Different from X?

You can check out the Comparisons section for more details on how Vitest differs from other similar tools.

Go to TOC

Colophon

This book is created by using the following sources:

- Vitest - English
- GitHub source: vitest-dev/vitest/docs
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf