Test runner | Node.js v20.3.1 Documentation

Source Code: lib/test.js

The `node:test` module facilitates the creation of JavaScript tests. To access it:

```
import test from 'node:test';const test = require('node:test');
```

This module is only available under the `node:` scheme. The following will not work:

```
import test from 'test';const test = require('test');
```

Tests created via the `test` module consist of a single function that is processed in one of three ways:

1. A synchronous function that is considered failing if it throws an exception, and is considered passing otherwise.
2. A function that returns a `Promise` that is considered failing if the `Promise` rejects, and is considered passing if the `Promise` resolves.
3. A function that receives a callback function. If the callback receives any truthy value as its first argument, the test is considered failing. If a falsy value is passed as the first argument to the callback, the test is considered passing. If the test function receives a callback function and also returns a `Promise`, the test will fail.

The following example illustrates how tests are written using the `test` module.

```
test('synchronous passing test', (t) => {
  // This test passes because it does not throw an exception.
  assert.strictEqual(1, 1);
});

test('synchronous failing test', (t) => {
  // This test fails because it throws an exception.
  assert.strictEqual(1, 2);
});

test('asynchronous passing test', async (t) => {
  // This test passes because the Promise returned by the async
  // function is not rejected.
```

```
  assert.strictEqual(1, 1);
});

test('asynchronous failing test', async (t) => {
  // This test fails because the Promise returned by the async
  // function is rejected.
  assert.strictEqual(1, 2);
});

test('failing test using Promises', (t) => {
  // Promises can be used directly as well.
  return new Promise((resolve, reject) => {
    setImmediate(() => {
      reject(new Error('this will cause the test to fail'));
    });
  });
});

test('callback passing test', (t, done) => {
  // done() is the callback function. When the setImmediate() runs
, it invokes
  // done() with no arguments.
  setImmediate(done);
});

test('callback failing test', (t, done) => {
  // When the setImmediate() runs, done() is invoked with an Error
object and
  // the test fails.
  setImmediate(() => {
    done(new Error('callback failure'));
  });
});
```

If any tests fail, the process exit code is set to `1`.

Subtests#

The test context's `test()` method allows subtests to be created. This method behaves identically to the top level `test()` function. The following example demonstrates the creation of a top level test with two subtests.

```
test('top level test', async (t) => {
  await t.test('subtest 1', (t) => {
    assert.strictEqual(1, 1);
  });

  await t.test('subtest 2', (t) => {
    assert.strictEqual(2, 2);
  });
```

```
});
```

In this example, `await` is used to ensure that both subtests have completed. This is necessary because parent tests do not wait for their subtests to complete. Any subtests that are still outstanding when their parent finishes are cancelled and treated as failures. Any subtest failures cause the parent test to fail.

Skipping tests#

Individual tests can be skipped by passing the `skip` option to the test, or by calling the test context's `skip()` method as shown in the following example.

```
// The skip option is used, but no message is provided.
test('skip option', { skip: true }, (t) => {
  // This code is never executed.
});

// The skip option is used, and a message is provided.
test('skip option with message', { skip: 'this is skipped' }, (t)
=> {
  // This code is never executed.
});

test('skip() method', (t) => {
  // Make sure to return here as well if the test contains additio
nal logic.
  t.skip();
});

test('skip() method with message', (t) => {
  // Make sure to return here as well if the test contains additio
nal logic.
  t.skip('this is skipped');
});
```

`describe`/`it` syntax#

Running tests can also be done using `describe` to declare a suite and `it` to declare a test. A suite is used to organize and group related tests together. `it` is a shorthand for `test()`.

```
describe('A thing', () => {
  it('should work', () => {
    assert.strictEqual(1, 1);
  });

  it('should be ok', () => {
    assert.strictEqual(2, 2);
```

```
  });

  describe('a nested thing', () => {
    it('should work', () => {
      assert.strictEqual(3, 3);
    });
  });
});
```

`describe` and `it` are imported from the `node:test` module.

```
import { describe, it } from 'node:test';const { describe, it } =
require('node:test');
```

`only` tests#

If Node.js is started with the `--test-only` command-line option, it is possible to skip all top level tests except for a selected subset by passing the `only` option to the tests that should be run. When a test with the `only` option set is run, all subtests are also run. The test context's `runOnly()` method can be used to implement the same behavior at the subtest level.

```
// Assume Node.js is run with the --test-only command-line option.
// The 'only' option is set, so this test is run.
test('this test is run', { only: true }, async (t) => {
  // Within this test, all subtests are run by default.
  await t.test('running subtest');

  // The test context can be updated to run subtests with the 'onl
y' option.
  t.runOnly(true);
  await t.test('this subtest is now skipped');
  await t.test('this subtest is run', { only: true });

  // Switch the context back to execute all tests.
  t.runOnly(false);
  await t.test('this subtest is now run');

  // Explicitly do not run these tests.
  await t.test('skipped subtest 3', { only: false });
  await t.test('skipped subtest 4', { skip: true });
});

// The 'only' option is not set, so this test is skipped.
test('this test is not run', () => {
  // This code is not run.
  throw new Error('fail');
});
```

Filtering tests by name#

The `--test-name-pattern` command-line option can be used to only run tests whose name matches the provided pattern. Test name patterns are interpreted as JavaScript regular expressions. The `--test-name-pattern` option can be specified multiple times in order to run nested tests. For each test that is executed, any corresponding test hooks, such as `beforeEach()`, are also run.

Given the following test file, starting Node.js with the `--test-name-pattern="test [1-3]"` option would cause the test runner to execute `test 1`, `test 2`, and `test 3`. If `test 1` did not match the test name pattern, then its subtests would not execute, despite matching the pattern. The same set of tests could also be executed by passing `--test-name-pattern` multiple times (e.g. `--test-name-pattern="test 1"`, `--test-name-pattern="test 2"`, etc.).

```
test('test 1', async (t) => {
  await t.test('test 2');
  await t.test('test 3');
});

test('Test 4', async (t) => {
  await t.test('Test 5');
  await t.test('test 6');
});
```

Test name patterns can also be specified using regular expression literals. This allows regular expression flags to be used. In the previous example, starting Node.js with `--test-name-pattern="/test [4-5]/i"` would match `Test 4` and `Test 5` because the pattern is case-insensitive.

Test name patterns do not change the set of files that the test runner executes.

Extraneous asynchronous activity#

Once a test function finishes executing, the results are reported as quickly as possible while maintaining the order of the tests. However, it is possible for the test function to generate asynchronous activity that outlives the test itself. The test runner handles this type of activity, but does not delay the reporting of test results in order to accommodate it.

In the following example, a test completes with two `setImmediate()` operations still outstanding. The first `setImmediate()` attempts to create a new subtest. Because the parent test has already finished and output its results, the new subtest

is immediately marked as failed, and reported later to the <TestsStream>.

The second `setImmediate()` creates an `uncaughtException` event. `uncaughtException` and `unhandledRejection` events originating from a completed test are marked as failed by the `test` module and reported as diagnostic warnings at the top level by the <TestsStream>.

```
test('a test that creates asynchronous activity', (t) => {
  setImmediate(() => {
    t.test('subtest that is created too late', (t) => {
      throw new Error('error1');
    });
  });

  setImmediate(() => {
    throw new Error('error2');
  });

  // The test finishes after this line.
});
```

Watch mode#

Added in: v19.2.0, v18.13.0

The Node.js test runner supports running in watch mode by passing the `--watch` flag:

```
node --test --watch
```

In watch mode, the test runner will watch for changes to test files and their dependencies. When a change is detected, the test runner will rerun the tests affected by the change. The test runner will continue to run until the process is terminated.

Running tests from the command line#

The Node.js test runner can be invoked from the command line by passing the `--test` flag:

```
node --test
```

By default, Node.js will recursively search the current directory for JavaScript source files matching a specific naming convention. Matching files are executed as test files. More information on the expected test file naming convention and

behavior can be found in the test runner execution model section.

Alternatively, one or more paths can be provided as the final argument(s) to the Node.js command, as shown below.

```
node --test test1.js test2.mjs custom_test_dir/
```

In this example, the test runner will execute the files `test1.js` and `test2.mjs`. The test runner will also recursively search the `custom_test_dir/` directory for test files to execute.

Test runner execution model#

When searching for test files to execute, the test runner behaves as follows:

- Any files explicitly provided by the user are executed.
- If the user did not explicitly specify any paths, the current working directory is recursively searched for files as specified in the following steps.
- `node_modules` directories are skipped unless explicitly provided by the user.
- If a directory named `test` is encountered, the test runner will search it recursively for all all `.js`, `.cjs`, and `.mjs` files. All of these files are treated as test files, and do not need to match the specific naming convention detailed below. This is to accommodate projects that place all of their tests in a single `test` directory.
- In all other directories, `.js`, `.cjs`, and `.mjs` files matching the following patterns are treated as test files:
  - `^test$` - Files whose basename is the string `'test'`. Examples: `test.js`, `test.cjs`, `test.mjs`.
  - `^test-.+` - Files whose basename starts with the string `'test-'` followed by one or more characters. Examples: `test-example.js`, `test-another-example.mjs`.
  - `.+[\.\-\_]test$` - Files whose basename ends with `.test`, `-test`, or `_test`, preceded by one or more characters. Examples: `example.test.js`, `example-test.cjs`, `example_test.mjs`.
  - Other file types understood by Node.js such as `.node` and `.json` are not automatically executed by the test runner, but are supported if explicitly provided on the command line.

Each matching test file is executed in a separate child process. If the child process finishes with an exit code of 0, the test is considered passing. Otherwise, the test is considered to be a failure. Test files must be executable by Node.js, but are not required to use the `node:test` module internally.

Each test file is executed as if it was a regular script. That is, if the test file itself uses `node:test` to define tests, all of those tests will be executed within a single application thread, regardless of the value of the `concurrency` option of `test()`.

Collecting code coverage#

When Node.js is started with the `--experimental-test-coverage` command-line flag, code coverage is collected and statistics are reported once all tests have completed. If the `NODE_V8_COVERAGE` environment variable is used to specify a code coverage directory, the generated V8 coverage files are written to that directory. Node.js core modules and files within `node_modules/` directories are not included in the coverage report. If coverage is enabled, the coverage report is sent to any test reporters via the `'test:coverage'` event.

Coverage can be disabled on a series of lines using the following comment syntax:

```
/* node:coverage disable */
if (anAlwaysFalseCondition) {
  // Code in this branch will never be executed, but the lines are
ignored for
  // coverage purposes. All lines following the 'disable' comment
are ignored
  // until a corresponding 'enable' comment is encountered.
  console.log('this is never executed');
}
/* node:coverage enable */
```

Coverage can also be disabled for a specified number of lines. After the specified number of lines, coverage will be automatically reenabled. If the number of lines is not explicitly provided, a single line is ignored.

```
/* node:coverage ignore next */
if (anAlwaysFalseCondition) { console.log('this is never executed'
); }

/* node:coverage ignore next 3 */
if (anAlwaysFalseCondition) {
  console.log('this is never executed');
}
```

The test runner's code coverage functionality has the following limitations, which will be addressed in a future Node.js release:

- Source maps are not supported.
- Excluding specific files or directories from the coverage report is not supported.

Mocking#

The `node:test` module supports mocking during testing via a top-level `mock` object. The following example creates a spy on a function that adds two numbers together. The spy is then used to assert that the function was called as expected.

```
import assert from 'node:assert';
import { mock, test } from 'node:test';

test('spies on a function', () => {
  const sum = mock.fn((a, b) => {
    return a + b;
  });

  assert.strictEqual(sum.mock.calls.length, 0);
  assert.strictEqual(sum(3, 4), 7);
  assert.strictEqual(sum.mock.calls.length, 1);

  const call = sum.mock.calls[0];
  assert.deepStrictEqual(call.arguments, [3, 4]);
  assert.strictEqual(call.result, 7);
  assert.strictEqual(call.error, undefined);

  // Reset the globally tracked mocks.
  mock.reset();
});'use strict';
const assert = require('node:assert');
const { mock, test } = require('node:test');

test('spies on a function', () => {
  const sum = mock.fn((a, b) => {
    return a + b;
  });

  assert.strictEqual(sum.mock.calls.length, 0);
  assert.strictEqual(sum(3, 4), 7);
  assert.strictEqual(sum.mock.calls.length, 1);

  const call = sum.mock.calls[0];
  assert.deepStrictEqual(call.arguments, [3, 4]);
  assert.strictEqual(call.result, 7);
  assert.strictEqual(call.error, undefined);
```

```
  // Reset the globally tracked mocks.
  mock.reset();
});
```

The same mocking functionality is also exposed on the `TestContext` object of each test. The following example creates a spy on an object method using the API exposed on the `TestContext`. The benefit of mocking via the test context is that the test runner will automatically restore all mocked functionality once the test finishes.

```
test('spies on an object method', (t) => {
  const number = {
    value: 5,
    add(a) {
      return this.value + a;
    },
  };

  t.mock.method(number, 'add');
  assert.strictEqual(number.add.mock.calls.length, 0);
  assert.strictEqual(number.add(3), 8);
  assert.strictEqual(number.add.mock.calls.length, 1);

  const call = number.add.mock.calls[0];

  assert.deepStrictEqual(call.arguments, [3]);
  assert.strictEqual(call.result, 8);
  assert.strictEqual(call.target, undefined);
  assert.strictEqual(call.this, number);
});
```

Test reporters#

The `node:test` module supports passing `--test-reporter` flags for the test runner to use a specific reporter.

The following built-reporters are supported:

- `tap` The `tap` reporter outputs the test results in the TAP format.
- `spec` The `spec` reporter outputs the test results in a human-readable format.
- `dot` The `dot` reporter outputs the test results in a compact format, where each passing test is represented by a `.`, and each failing test is represented by a `X`.

When `stdout` is a TTY, the `spec` reporter is used by default. Otherwise, the `tap` reporter is used by default.

The exact output of these reporters is subject to change between versions of Node.js, and should not be relied on programmatically. If programmatic access to the test runner's output is required, use the events emitted by the <TestsStream>.

The reporters are available via the `node:test/reporters` module:

```
import { tap, spec, dot } from 'node:test/reporters';const { tap,
spec, dot } = require('node:test/reporters');
```

Custom reporters#

`--test-reporter` can be used to specify a path to custom reporter. A custom reporter is a module that exports a value accepted by stream.compose. Reporters should transform events emitted by a <TestsStream>

Example of a custom reporter using <stream.Transform>:

```
import { Transform } from 'node:stream';

const customReporter = new Transform({
  writableObjectMode: true,
  transform(event, encoding, callback) {
    switch (event.type) {
      case 'test:start':
        callback(null, `test ${event.data.name} started`);
        break;
      case 'test:pass':
        callback(null, `test ${event.data.name} passed`);
        break;
      case 'test:fail':
        callback(null, `test ${event.data.name} failed`);
        break;
      case 'test:plan':
        callback(null, 'test plan');
        break;
      case 'test:diagnostic':
        callback(null, event.data.message);
        break;
      case 'test:coverage': {
        const { totalLineCount } = event.data.summary.totals;
        callback(null, `total line count: ${totalLineCount}\n`);
        break;
      }
    }
  },
});

export default customReporter;const { Transform } = require('node:
stream');
```

```
const customReporter = new Transform({
  writableObjectMode: true,
  transform(event, encoding, callback) {
    switch (event.type) {
      case 'test:start':
        callback(null, `test ${event.data.name} started`);
        break;
      case 'test:pass':
        callback(null, `test ${event.data.name} passed`);
        break;
      case 'test:fail':
        callback(null, `test ${event.data.name} failed`);
        break;
      case 'test:plan':
        callback(null, 'test plan');
        break;
      case 'test:diagnostic':
        callback(null, event.data.message);
        break;
      case 'test:coverage': {
        const { totalLineCount } = event.data.summary.totals;
        callback(null, `total line count: ${totalLineCount}\n`);
        break;
      }
    }
  },
});

module.exports = customReporter;
```

Example of a custom reporter using a generator function:

```
export default async function * customReporter(source) {
  for await (const event of source) {
    switch (event.type) {
      case 'test:start':
        yield `test ${event.data.name} started\n`;
        break;
      case 'test:pass':
        yield `test ${event.data.name} passed\n`;
        break;
      case 'test:fail':
        yield `test ${event.data.name} failed\n`;
        break;
      case 'test:plan':
        yield 'test plan';
        break;
      case 'test:diagnostic':
        yield `${event.data.message}\n`;
        break;
      case 'test:coverage': {
```

```
        const { totalLineCount } = event.data.summary.totals;
        yield `total line count: ${totalLineCount}\n`;
        break;
      }
    }
  }
}module.exports = async function * customReporter(source) {
  for await (const event of source) {
    switch (event.type) {
      case 'test:start':
        yield `test ${event.data.name} started\n`;
        break;
      case 'test:pass':
        yield `test ${event.data.name} passed\n`;
        break;
      case 'test:fail':
        yield `test ${event.data.name} failed\n`;
        break;
      case 'test:plan':
        yield 'test plan\n';
        break;
      case 'test:diagnostic':
        yield `${event.data.message}\n`;
        break;
      case 'test:coverage': {
        const { totalLineCount } = event.data.summary.totals;
        yield `total line count: ${totalLineCount}\n`;
        break;
      }
    }
  }
};
```

The value provided to `--test-reporter` should be a string like one used in an `import()` in JavaScript code, or a value provided for `--import`.

Multiple reporters#

The `--test-reporter` flag can be specified multiple times to report test results in several formats. In this situation it is required to specify a destination for each reporter using `--test-reporter-destination`. Destination can be `stdout`, `stderr`, or a file path. Reporters and destinations are paired according to the order they were specified.

In the following example, the `spec` reporter will output to `stdout`, and the `dot` reporter will output to `file.txt`:

```
node --test-reporter=spec --test-reporter=dot --test-reporter-dest
```

```
ination=stdout --test-reporter-destination=file.txt
```

When a single reporter is specified, the destination will default to `stdout`, unless a destination is explicitly provided.

`run([options])#`

- `options` <Object> Configuration options for running tests. The following properties are supported:
  - `concurrency` <number> | <boolean> If a number is provided, then that many test processes would run in parallel, where each process corresponds to one test file. If `true`, it would run `os.availableParallelism() - 1` test files in parallel. If `false`, it would only run one test file at a time. **Default:** `false`.
  - `files`: <Array> An array containing the list of files to run. **Default** matching files from test runner execution model.
  - `inspectPort` <number> | <Function> Sets inspector port of test child process. This can be a number, or a function that takes no arguments and returns a number. If a nullish value is provided, each process gets its own port, incremented from the primary's `process.debugPort`. **Default:** `undefined`.
  - `setup` <Function> A function that accepts the `TestsStream` instance and can be used to setup listeners before any tests are run. **Default:** `undefined`.
  - `signal` <AbortSignal> Allows aborting an in-progress test execution.
  - `testNamePatterns` <string> | <RegExp> | <Array> A String, RegExp or a RegExp Array, that can be used to only run tests whose name matches the provided pattern. Test name patterns are interpreted as JavaScript regular expressions. For each test that is executed, any corresponding test hooks, such as `beforeEach()`, are also run. **Default:** `undefined`.
  - `timeout` <number> A number of milliseconds the test execution will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.
  - `watch` <boolean> Whether to run in watch mode or not. **Default:** `false`.
- Returns: <TestsStream>

```
import { tap } from 'node:test/reporters';
import process from 'node:process';

run({ files: [path.resolve('./tests/test.js')] })
  .compose(tap)
  .pipe(process.stdout);const { tap } = require('node:test/reporte
rs');

run({ files: [path.resolve('./tests/test.js')] })
  .compose(tap)
  .pipe(process.stdout);
```

`test([name][, options][, fn])#`

- `name` <string> The name of the test, which is displayed when reporting test results. **Default:** The `name` property of `fn`, or `'<anonymous>'` if `fn` does not have a name.
- `options` <Object> Configuration options for the test. The following properties are supported:
  - `concurrency` <number> | <boolean> If a number is provided, then that many tests would run in parallel within the application thread. If `true`, all scheduled asynchronous tests run concurrently within the thread. If `false`, only one test runs at a time. If unspecified, subtests inherit this value from their parent. **Default:** `false`.
  - `only` <boolean> If truthy, and the test context is configured to run `only` tests, then this test will be run. Otherwise, the test is skipped. **Default:** `false`.
  - `signal` <AbortSignal> Allows aborting an in-progress test.
  - `skip` <boolean> | <string> If truthy, the test is skipped. If a string is provided, that string is displayed in the test results as the reason for skipping the test. **Default:** `false`.
  - `todo` <boolean> | <string> If truthy, the test marked as `TODO`. If a string is provided, that string is displayed in the test results as the reason why the test is `TODO`. **Default:** `false`.
  - `timeout` <number> A number of milliseconds the test will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.
- `fn` <Function> | <AsyncFunction> The function under test. The first argument to

this function is a `TestContext` object. If the test uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- Returns: <Promise> Resolved with `undefined` once the test completes, or immediately if the test runs within `describe()`.

The `test()` function is the value imported from the `test` module. Each invocation of this function results in reporting the test to the <TestsStream>.

The `TestContext` object passed to the `fn` argument can be used to perform actions related to the current test. Examples include skipping the test, adding additional diagnostic information, or creating subtests.

`test()` returns a `Promise` that resolves once the test completes. if `test()` is called within a `describe()` block, it resolve immediately. The return value can usually be discarded for top level tests. However, the return value from subtests should be used to prevent the parent test from finishing first and cancelling the subtest as shown in the following example.

```
test('top level test', async (t) => {
  // The setTimeout() in the following subtest would cause it to o
utlive its
  // parent test if 'await' is removed on the next line. Once the
parent test
  // completes, it will cancel any outstanding subtests.
  await t.test('longer running subtest', async (t) => {
    return new Promise((resolve, reject) => {
      setTimeout(resolve, 1000);
    });
  });
});
```

The `timeout` option can be used to fail the test if it takes longer than `timeout` milliseconds to complete. However, it is not a reliable mechanism for canceling tests because a running test might block the application thread and thus prevent the scheduled cancellation.

`test.skip([name][, options][, fn])#`

Shorthand for skipping a test, same as `test([name], { skip: true }[, fn])`.

`test.todo([name][, options][, fn])#`

Shorthand for marking a test as `TODO`, same as `test([name], { todo: true }[,`

fn]).

test.only([name][, options][, fn])#

Shorthand for marking a test as only, same as test([name], { only: true }[, fn]).

describe([name][, options][, fn])#

- name <string> The name of the suite, which is displayed when reporting test results. **Default:** The name property of fn, or '<anonymous>' if fn does not have a name.
- options <Object> Configuration options for the suite. supports the same options as test([name][, options][, fn]).
- fn <Function> | <AsyncFunction> The function under suite declaring all subtests and subsuites. The first argument to this function is a SuiteContext object. **Default:** A no-op function.
- Returns: undefined.

The describe() function imported from the node:test module. Each invocation of this function results in the creation of a Subtest. After invocation of top level describe functions, all top level tests and suites will execute.

describe.skip([name][, options][, fn])#

Shorthand for skipping a suite, same as describe([name], { skip: true }[, fn]).

describe.todo([name][, options][, fn])#

Shorthand for marking a suite as TODO, same as describe([name], { todo: true }[, fn]).

describe.only([name][, options][, fn])#

Added in: v19.8.0, v18.15.0

Shorthand for marking a suite as only, same as describe([name], { only: true }[, fn]).

`it([name][, options][, fn])#`

Shorthand for `test()`.

The `it()` function is imported from the `node:test` module.

`it.skip([name][, options][, fn])#`

Shorthand for skipping a test, same as `it([name], { skip: true }[, fn])`.

`it.todo([name][, options][, fn])#`

Shorthand for marking a test as `TODO`, same as `it([name], { todo: true }[, fn])`.

`it.only([name][, options][, fn])#`

Added in: v19.8.0, v18.15.0

Shorthand for marking a test as `only`, same as `it([name], { only: true }[, fn])`.

`before([fn][, options])#`

Added in: v18.8.0, v16.18.0

- `fn` <Function> | <AsyncFunction> The hook function. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running before running a suite.

```
describe('tests', async () => {
  before(() => console.log('about to run some test'));
  it('is a subtest', () => {
```

```
    assert.ok('some relevant assertion here');
  });
});
```

after([fn][, options])#

Added in: v18.8.0, v16.18.0

- `fn` <Function> | <AsyncFunction> The hook function. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running after running a suite.

```
describe('tests', async () => {
  after(() => console.log('finished running tests'));
  it('is a subtest', () => {
    assert.ok('some relevant assertion here');
  });
});
```

beforeEach([fn][, options])#

Added in: v18.8.0, v16.18.0

- `fn` <Function> | <AsyncFunction> The hook function. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running before each subtest of the current suite.

```
describe('tests', async () => {
  beforeEach(() => console.log('about to run a test'));
  it('is a subtest', () => {
    assert.ok('some relevant assertion here');
  });
});
```

`afterEach([fn][, options])#`

Added in: v18.8.0, v16.18.0

- `fn` <Function> | <AsyncFunction> The hook function. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running after each subtest of the current test.

```
describe('tests', async () => {
  afterEach(() => console.log('finished running a test'));
  it('is a subtest', () => {
    assert.ok('some relevant assertion here');
  });
});
```

Class: `MockFunctionContext`#

Added in: v19.1.0, v18.13.0

The `MockFunctionContext` class is used to inspect or manipulate the behavior of mocks created via the `MockTracker` APIs.

`ctx.calls`#

Added in: v19.1.0, v18.13.0

- <Array>

A getter that returns a copy of the internal array used to track calls to the mock. Each entry in the array is an object with the following properties.

- `arguments` <Array> An array of the arguments passed to the mock function.
- `error` <any> If the mocked function threw then this property contains the thrown value. **Default:** `undefined`.
- `result` <any> The value returned by the mocked function.
- `stack` <Error> An `Error` object whose stack can be used to determine the callsite of the mocked function invocation.
- `target` <Function> | <undefined> If the mocked function is a constructor, this field contains the class being constructed. Otherwise this will be `undefined`.
- `this` <any> The mocked function's `this` value.

`ctx.callCount()`#

Added in: v19.1.0, v18.13.0

- Returns: <integer> The number of times that this mock has been invoked.

This function returns the number of times that this mock has been invoked. This function is more efficient than checking `ctx.calls.length` because `ctx.calls` is a getter that creates a copy of the internal call tracking array.

`ctx.mockImplementation(implementation)`#

Added in: v19.1.0, v18.13.0

- `implementation` <Function> | <AsyncFunction> The function to be used as the mock's new implementation.

This function is used to change the behavior of an existing mock.

The following example creates a mock function using `t.mock.fn()`, calls the mock function, and then changes the mock implementation to a different function.

```
test('changes a mock behavior', (t) => {
  let cnt = 0;
```

```
  function addOne() {
    cnt++;
    return cnt;
  }

  function addTwo() {
    cnt += 2;
    return cnt;
  }

  const fn = t.mock.fn(addOne);

  assert.strictEqual(fn(), 1);
  fn.mock.mockImplementation(addTwo);
  assert.strictEqual(fn(), 3);
  assert.strictEqual(fn(), 5);
});
```

`ctx.mockImplementationOnce(implementation[, onCall])#`

Added in: v19.1.0, v18.13.0

- `implementation` <Function> | <AsyncFunction> The function to be used as the mock's implementation for the invocation number specified by `onCall`.
- `onCall` <integer> The invocation number that will use `implementation`. If the specified invocation has already occurred then an exception is thrown. **Default:** The number of the next invocation.

This function is used to change the behavior of an existing mock for a single invocation. Once invocation `onCall` has occurred, the mock will revert to whatever behavior it would have used had `mockImplementationOnce()` not been called.

The following example creates a mock function using `t.mock.fn()`, calls the mock function, changes the mock implementation to a different function for the next invocation, and then resumes its previous behavior.

```
test('changes a mock behavior once', (t) => {
  let cnt = 0;

  function addOne() {
    cnt++;
    return cnt;
  }

  function addTwo() {
    cnt += 2;
    return cnt;
```

```
  }

  const fn = t.mock.fn(addOne);

  assert.strictEqual(fn(), 1);
  fn.mock.mockImplementationOnce(addTwo);
  assert.strictEqual(fn(), 3);
  assert.strictEqual(fn(), 4);
});
```

`ctx.resetCalls()`#

Added in: v19.3.0, v18.13.0

Resets the call history of the mock function.

`ctx.restore()`#

Added in: v19.1.0, v18.13.0

Resets the implementation of the mock function to its original behavior. The mock can still be used after calling this function.

Class: `MockTracker`#

Added in: v19.1.0, v18.13.0

The `MockTracker` class is used to manage mocking functionality. The test runner module provides a top level `mock` export which is a `MockTracker` instance. Each test also provides its own `MockTracker` instance via the test context's `mock` property.

`mock.fn([original[, implementation]][, options])`#

Added in: v19.1.0, v18.13.0

- `original` <Function> | <AsyncFunction> An optional function to create a mock on. **Default:** A no-op function.
- `implementation` <Function> | <AsyncFunction> An optional function used as the mock implementation for `original`. This is useful for creating mocks that exhibit one behavior for a specified number of calls and then restore the behavior of `original`. **Default:** The function specified by `original`.
- `options` <Object> Optional configuration options for the mock function. The

following properties are supported:

- `times` <integer> The number of times that the mock will use the behavior of `implementation`. Once the mock function has been called `times` times, it will automatically restore the behavior of `original`. This value must be an integer greater than zero. **Default:** `Infinity`.

- Returns: <Proxy> The mocked function. The mocked function contains a special `mock` property, which is an instance of `MockFunctionContext`, and can be used for inspecting and changing the behavior of the mocked function.

This function is used to create a mock function.

The following example creates a mock function that increments a counter by one on each invocation. The `times` option is used to modify the mock behavior such that the first two invocations add two to the counter instead of one.

```
test('mocks a counting function', (t) => {
  let cnt = 0;

  function addOne() {
    cnt++;
    return cnt;
  }

  function addTwo() {
    cnt += 2;
    return cnt;
  }

  const fn = t.mock.fn(addOne, addTwo, { times: 2 });

  assert.strictEqual(fn(), 2);
  assert.strictEqual(fn(), 4);
  assert.strictEqual(fn(), 5);
  assert.strictEqual(fn(), 6);
});
```

`mock.getter(object, methodName[, implementation][, options])`#

Added in: v19.3.0, v18.13.0

This function is syntax sugar for `MockTracker.method` with `options.getter` set to `true`.

`mock.method(object, methodName[, implementation][, options])`#

Added in: v19.1.0, v18.13.0

- `object` <Object> The object whose method is being mocked.
- `methodName` <string> | <symbol> The identifier of the method on `object` to mock. If `object[methodName]` is not a function, an error is thrown.
- `implementation` <Function> | <AsyncFunction> An optional function used as the mock implementation for `object[methodName]`. **Default:** The original method specified by `object[methodName]`.
- `options` <Object> Optional configuration options for the mock method. The following properties are supported:
  - `getter` <boolean> If `true`, `object[methodName]` is treated as a getter. This option cannot be used with the `setter` option. **Default:** false.
  - `setter` <boolean> If `true`, `object[methodName]` is treated as a setter. This option cannot be used with the `getter` option. **Default:** false.
  - `times` <integer> The number of times that the mock will use the behavior of `implementation`. Once the mocked method has been called `times` times, it will automatically restore the original behavior. This value must be an integer greater than zero. **Default:** `Infinity`.
- Returns: <Proxy> The mocked method. The mocked method contains a special `mock` property, which is an instance of `MockFunctionContext`, and can be used for inspecting and changing the behavior of the mocked method.

This function is used to create a mock on an existing object method. The following example demonstrates how a mock is created on an existing object method.

```
test('spies on an object method', (t) => {
  const number = {
    value: 5,
    subtract(a) {
      return this.value - a;
    },
  };

  t.mock.method(number, 'subtract');
  assert.strictEqual(number.subtract.mock.calls.length, 0);
  assert.strictEqual(number.subtract(3), 2);
  assert.strictEqual(number.subtract.mock.calls.length, 1);

  const call = number.subtract.mock.calls[0];

  assert.deepStrictEqual(call.arguments, [3]);
```

```
  assert.strictEqual(call.result, 2);
  assert.strictEqual(call.error, undefined);
  assert.strictEqual(call.target, undefined);
  assert.strictEqual(call.this, number);
});
```

`mock.reset()`#

Added in: v19.1.0, v18.13.0

This function restores the default behavior of all mocks that were previously created by this `MockTracker` and disassociates the mocks from the `MockTracker` instance. Once disassociated, the mocks can still be used, but the `MockTracker` instance can no longer be used to reset their behavior or otherwise interact with them.

After each test completes, this function is called on the test context's `MockTracker`. If the global `MockTracker` is used extensively, calling this function manually is recommended.

`mock.restoreAll()`#

Added in: v19.1.0, v18.13.0

This function restores the default behavior of all mocks that were previously created by this `MockTracker`. Unlike `mock.reset()`, `mock.restoreAll()` does not disassociate the mocks from the `MockTracker` instance.

`mock.setter(object, methodName[, implementation][, options])`#

Added in: v19.3.0, v18.13.0

This function is syntax sugar for `MockTracker.method` with `options.setter` set to `true`.

Class: `TestsStream`#

Added in: v18.9.0, v16.19.0

- Extends <ReadableStream>

A successful call to `run()` method will return a new <TestsStream> object, streaming a series of events representing the execution of the tests. `TestsStream`

will emit events, in the order of the tests definition

Event: `'test:coverage'`#

- `data` <Object>
  - `summary` <Object> An object containing the coverage report.
    - `files` <Array> An array of coverage reports for individual files. Each report is an object with the following schema:
      - `path` <string> The absolute path of the file.
      - `totalLineCount` <number> The total number of lines.
      - `totalBranchCount` <number> The total number of branches.
      - `totalFunctionCount` <number> The total number of functions.
      - `coveredLineCount` <number> The number of covered lines.
      - `coveredBranchCount` <number> The number of covered branches.
      - `coveredFunctionCount` <number> The number of covered functions.
      - `coveredLinePercent` <number> The percentage of lines covered.
      - `coveredBranchPercent` <number> The percentage of branches covered.
      - `coveredFunctionPercent` <number> The percentage of functions covered.
      - `uncoveredLineNumbers` <Array> An array of integers representing line numbers that are uncovered.
    - `totals` <Object> An object containing a summary of coverage for all files.
      - `totalLineCount` <number> The total number of lines.
      - `totalBranchCount` <number> The total number of branches.
      - `totalFunctionCount` <number> The total number of functions.
      - `coveredLineCount` <number> The number of covered lines.
      - `coveredBranchCount` <number> The number of covered branches.
      - `coveredFunctionCount` <number> The number of covered functions.
      - `coveredLinePercent` <number> The percentage of lines covered.
      - `coveredBranchPercent` <number> The percentage of branches covered.

        - `coveredFunctionPercent` <number> The percentage of functions covered.

      - `workingDirectory` <string> The working directory when code coverage began. This is useful for displaying relative path names in case the tests changed the working directory of the Node.js process.

    - `nesting` <number> The nesting level of the test.

Emitted when code coverage is enabled and all tests have completed.

Event: `'test:diagnostic'`#

- `data` <Object>
  - `file` <string> | <undefined> The path of the test file, undefined if test is not ran through a file.
  - `message` <string> The diagnostic message.
  - `nesting` <number> The nesting level of the test.

Emitted when `context.diagnostic` is called.

Event: `'test:fail'`#

- `data` <Object>
  - `details` <Object> Additional execution metadata.
    - `duration` <number> The duration of the test in milliseconds.
    - `error` <Error> The error thrown by the test.
  - `file` <string> | <undefined> The path of the test file, undefined if test is not ran through a file.
  - `name` <string> The test name.
  - `nesting` <number> The nesting level of the test.
  - `testNumber` <number> The ordinal number of the test.
  - `todo` <string> | <boolean> | <undefined> Present if `context.todo` is called
  - `skip` <string> | <boolean> | <undefined> Present if `context.skip` is called

Emitted when a test fails.

Event: `'test:pass'`#

- `data` <Object>
  - `details` <Object> Additional execution metadata.
    - `duration` <number> The duration of the test in milliseconds.
  - `file` <string> | <undefined> The path of the test file, undefined if test is not ran through a file.
  - `name` <string> The test name.
  - `nesting` <number> The nesting level of the test.
  - `testNumber` <number> The ordinal number of the test.
  - `todo` <string> | <boolean> | <undefined> Present if `context.todo` is called
  - `skip` <string> | <boolean> | <undefined> Present if `context.skip` is called

Emitted when a test passes.

Event: `'test:plan'`#

- `data` <Object>
  - `file` <string> | <undefined> The path of the test file, undefined if test is not ran through a file.
  - `nesting` <number> The nesting level of the test.
  - `count` <number> The number of subtests that have ran.

Emitted when all subtests have completed for a given test.

Event: `'test:start'`#

- `data` <Object>
  - `file` <string> | <undefined> The path of the test file, undefined if test is not ran through a file.
  - `name` <string> The test name.
  - `nesting` <number> The nesting level of the test.

Emitted when a test starts.

Event: `'test:stderr'`#

- `data` <Object>
  - `file` <string> The path of the test file.
  - `message` <string> The message written to `stderr`.

Emitted when a running test writes to `stderr`. This event is only emitted if `--test` flag is passed.

Event: `'test:stdout'`#

- `data` <Object>
  - `file` <string> The path of the test file.
  - `message` <string> The message written to `stdout`.

Emitted when a running test writes to `stdout`. This event is only emitted if `--test` flag is passed.

Event: `'test:watch:drained'`#

Emitted when no more tests are queued for execution in watch mode.

Class: `TestContext`#

An instance of `TestContext` is passed to each test function in order to interact with the test runner. However, the `TestContext` constructor is not exposed as part of the API.

`context.before([fn][, options])`#

Added in: v20.1.0

- `fn` <Function> | <AsyncFunction> The hook function. The first argument to this function is a `TestContext` object. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running before subtest of the current test.

`context.beforeEach([fn][, options])#`

Added in: v18.8.0, v16.18.0

- `fn` <Function> | <AsyncFunction> The hook function. The first argument to this function is a `TestContext` object. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running before each subtest of the current test.

```
test('top level test', async (t) => {
  t.beforeEach((t) => t.diagnostic(`about to run ${t.name}`));
  await t.test(
    'This is a subtest',
    (t) => {
      assert.ok('some relevant assertion here');
    },
  );
});
```

`context.after([fn][, options])#`

Added in: v19.3.0, v18.13.0

- `fn` <Function> | <AsyncFunction> The hook function. The first argument to this function is a `TestContext` object. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinit`

y.

This function is used to create a hook that runs after the current test finishes.

```
test('top level test', async (t) => {
  t.after((t) => t.diagnostic(`finished running ${t.name}`));
  assert.ok('some relevant assertion here');
});
```

context.afterEach([fn][, options])#

Added in: v18.8.0, v16.18.0

- `fn` <Function> | <AsyncFunction> The hook function. The first argument to this function is a `TestContext` object. If the hook uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- `options` <Object> Configuration options for the hook. The following properties are supported:
  - `signal` <AbortSignal> Allows aborting an in-progress hook.
  - `timeout` <number> A number of milliseconds the hook will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

This function is used to create a hook running after each subtest of the current test.

```
test('top level test', async (t) => {
  t.afterEach((t) => t.diagnostic(`finished running ${t.name}`));
  await t.test(
    'This is a subtest',
    (t) => {
      assert.ok('some relevant assertion here');
    },
  );
});
```

context.diagnostic(message)#

Added in: v18.0.0, v16.17.0

- `message` <string> Message to be reported.

This function is used to write diagnostics to the output. Any diagnostic information is included at the end of the test's results. This function does not return a value.

```
test('top level test', (t) => {
  t.diagnostic('A diagnostic message');
});
```

context.name#

Added in: v18.8.0, v16.18.0

The name of the test.

context.runOnly(shouldRunOnlyTests)#

Added in: v18.0.0, v16.17.0

- `shouldRunOnlyTests` <boolean> Whether or not to run `only` tests.

If `shouldRunOnlyTests` is truthy, the test context will only run tests that have the `only` option set. Otherwise, all tests are run. If Node.js was not started with the `--test-only` command-line option, this function is a no-op.

```
test('top level test', (t) => {
  // The test context can be set to run subtests with the 'only' option.
  t.runOnly(true);
  return Promise.all([
    t.test('this subtest is now skipped'),
    t.test('this subtest is run', { only: true }),
  ]);
});
```

context.signal#

Added in: v18.7.0, v16.17.0

- <AbortSignal> Can be used to abort test subtasks when the test has been aborted.

```
test('top level test', async (t) => {
  await fetch('some/uri', { signal: t.signal });
});
```

context.skip([message])#

Added in: v18.0.0, v16.17.0

- `message` <string> Optional skip message.

This function causes the test's output to indicate the test as skipped. If `message` is provided, it is included in the output. Calling `skip()` does not terminate execution of the test function. This function does not return a value.

```
test('top level test', (t) => {
  // Make sure to return here as well if the test contains additio
nal logic.
  t.skip('this is skipped');
});
```

`context.todo([message])`#

Added in: v18.0.0, v16.17.0

- `message` <string> Optional `TODO` message.

This function adds a `TODO` directive to the test's output. If `message` is provided, it is included in the output. Calling `todo()` does not terminate execution of the test function. This function does not return a value.

```
test('top level test', (t) => {
  // This test is marked as `TODO`
  t.todo('this is a todo');
});
```

`context.test([name][, options][, fn])`#

- `name` <string> The name of the subtest, which is displayed when reporting test results. **Default:** The `name` property of `fn`, or `'<anonymous>'` if `fn` does not have a name.
- `options` <Object> Configuration options for the subtest. The following properties are supported:
  - `concurrency` <number> | <boolean> | <null> If a number is provided, then that many tests would run in parallel within the application thread. If `true`, it would run all subtests in parallel. If `false`, it would only run one test at a time. If unspecified, subtests inherit this value from their parent. **Default:** `null`.
  - `only` <boolean> If truthy, and the test context is configured to run `only` tests, then this test will be run. Otherwise, the test is skipped. **Default:** `fal`

se.
    - `signal` <AbortSignal> Allows aborting an in-progress test.
    - `skip` <boolean> | <string> If truthy, the test is skipped. If a string is provided, that string is displayed in the test results as the reason for skipping the test. **Default:** `false`.
    - `todo` <boolean> | <string> If truthy, the test marked as `TODO`. If a string is provided, that string is displayed in the test results as the reason why the test is `TODO`. **Default:** `false`.
    - `timeout` <number> A number of milliseconds the test will fail after. If unspecified, subtests inherit this value from their parent. **Default:** `Infinity`.

- `fn` <Function> | <AsyncFunction> The function under test. The first argument to this function is a `TestContext` object. If the test uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- Returns: <Promise> Resolved with `undefined` once the test completes.

This function is used to create subtests under the current test. This function behaves in the same fashion as the top level `test()` function.

```
test('top level test', async (t) => {
  await t.test(
    'This is a subtest',
    { only: false, skip: false, concurrency: 1, todo: false },
    (t) => {
      assert.ok('some relevant assertion here');
    },
  );
});
```

Class: `SuiteContext`#

Added in: v18.7.0, v16.17.0

An instance of `SuiteContext` is passed to each suite function in order to interact with the test runner. However, the `SuiteContext` constructor is not exposed as part of the API.

`context.name`#

Added in: v18.8.0, v16.18.0

The name of the suite.

`context.signal#`

Added in: v18.7.0, v16.17.0

- <AbortSignal> Can be used to abort test subtasks when the test has been aborted.