Test runner

▶ History

Stability: 2 - Stable

Source Code: lib/test.js

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

MJS modules

```
import test from 'node:test';
```

CJS modules

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

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

MJS modules

```
import test from 'test';
```

CJS modules

```
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` fulfills.
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 settled and 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'));
  });
}); copy
```

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

Subtests

The test context's `test()` method allows subtests to be created. It allows you to structure your tests in a hierarchical manner, where you can create nested tests within a larger test. 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);
  });
}); copy
```

Note: `beforeEach` and `afterEach` hooks are triggered between each subtest execution.

In this example, `await` is used to ensure that both subtests have completed. This is necessary because tests do not wait for their subtests to complete, unlike tests created within suites. 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 additional logic.
  t.skip();
});

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

TODO tests

Individual tests can be marked as flaky or incomplete by passing the `todo` option to the test, or by calling the test context's `todo()` method, as shown in the following example. These tests represent a pending implementation or bug that needs to be fixed. TODO tests are executed, but are not treated as test failures, and therefore do not affect the process exit code. If a test is marked as both TODO and skipped, the TODO option is ignored.

```
// The todo option is used, but no message is provided.
test('todo option', { todo: true }, (t) => {
  // This code is executed, but not treated as a failure.
  throw new Error('this does not fail the test');
});

// The todo option is used, and a message is provided.
test('todo option with message', { todo: 'this is a todo test' }, (t) => {
  // This code is executed.
});

test('todo() method', (t) => {
  t.todo();
});

test('todo() method with message', (t) => {
  t.todo('this is a todo test and is not treated as a failure');
  throw new Error('this does not fail the test');
}); copy
```

`describe()` and `it()` aliases

Suites and tests can also be written using the `describe()` and `it()` functions. `describe()` is an alias for `suite()`, and `it()` is an alias 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);
    });
  });
}); copy
```

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

MJS modules

```
import { describe, it } from 'node:test';
```

CJS modules

```
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 'only' 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');
}); copy
```

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');
}); copy
```

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.
}); copy
```

Watch mode

Added in: v19.2.0, v18.13.0

Stability: 1 - Experimental

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

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

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 copy
```

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/ copy
```

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. The maximum number of child processes running at any time is controlled by the `--test-concurrency` flag. 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

Stability: 1 - Experimental

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 */ copy
```

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');
} copy
```

Coverage reporters

The tap and spec reporters will print a summary of the coverage statistics. There is also an lcov reporter that will generate an lcov file which can be used as an in depth coverage report.

```
node --test --experimental-test-coverage --test-reporter=lcov --test-reporter-destination=lcov.info copy
```

Limitations

The test runner's code coverage functionality does not support excluding specific files or directories from the coverage report.

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.

MJS modules

```
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();
});
```

CJS modules

```
'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);
}); copy
```

Timers

Mocking timers is a technique commonly used in software testing to simulate and control the behavior of timers, such as `setInterval` and `setTimeout`, without actually waiting for the specified time intervals.

Refer to the `MockTimers` class for a full list of methods and features.

This allows developers to write more reliable and predictable tests for time-dependent functionality.

The example below shows how to mock `setTimeout`. Using `.enable({ apis: ['setTimeout'] });` it will mock the `set-Timeout` functions in the node:timers and node:timers/promises modules, as well as from the Node.js global context.

Note: Destructuring functions such as `import { setTimeout } from 'node:timers'` is currently not supported by this API.

MJS modules

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

test('mocks setTimeout to be executed synchronously without having to actually wait for it', () => {
  const fn = mock.fn();

  // Optionally choose what to mock
  mock.timers.enable({ apis: ['setTimeout'] });
  setTimeout(fn, 9999);
  assert.strictEqual(fn.mock.callCount(), 0);

  // Advance in time
  mock.timers.tick(9999);
  assert.strictEqual(fn.mock.callCount(), 1);

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

  // If you call reset mock instance, it will also reset timers instance
  mock.reset();
});
```

CJS modules

```
const assert = require('node:assert');
const { mock, test } = require('node:test');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', () => {
  const fn = mock.fn();

  // Optionally choose what to mock
  mock.timers.enable({ apis: ['setTimeout'] });
  setTimeout(fn, 9999);
  assert.strictEqual(fn.mock.callCount(), 0);

  // Advance in time
  mock.timers.tick(9999);
  assert.strictEqual(fn.mock.callCount(), 1);

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

  // If you call reset mock instance, it will also reset timers instance
  mock.reset();
});
```

The same mocking functionality is also exposed in the mock property on the `TestContext` object of each test. The benefit of mocking via the test context is that the test runner will automatically restore all mocked timers functionality once the test finishes.

MJS modules

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

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();

  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout'] });
  setTimeout(fn, 9999);
  assert.strictEqual(fn.mock.callCount(), 0);

  // Advance in time
  context.mock.timers.tick(9999);
  assert.strictEqual(fn.mock.callCount(), 1);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();

  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout'] });
  setTimeout(fn, 9999);
  assert.strictEqual(fn.mock.callCount(), 0);

  // Advance in time
  context.mock.timers.tick(9999);
  assert.strictEqual(fn.mock.callCount(), 1);
});
```

Dates

The mock timers API also allows the mocking of the `Date` object. This is a useful feature for testing time-dependent functionality, or to simulate internal calendar functions such as `Date.now()`.

The dates implementation is also part of the `MockTimers` class. Refer to it for a full list of methods and features.

Note: Dates and timers are dependent when mocked together. This means that if you have both the `Date` and `setTimeout` mocked, advancing the time will also advance the mocked date as they simulate a single internal clock.

The example below show how to mock the `Date` object and obtain the current `Date.now()` value.

MJS modules

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

test('mocks the Date object', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['Date'] });
  // If not specified, the initial date will be based on 0 in the UNIX epoch
  assert.strictEqual(Date.now(), 0);

  // Advance in time will also advance the date
  context.mock.timers.tick(9999);
  assert.strictEqual(Date.now(), 9999);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks the Date object', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['Date'] });
  // If not specified, the initial date will be based on 0 in the UNIX epoch
  assert.strictEqual(Date.now(), 0);

  // Advance in time will also advance the date
  context.mock.timers.tick(9999);
  assert.strictEqual(Date.now(), 9999);
});
```

If there is no initial epoch set, the initial date will be based on 0 in the Unix epoch. This is January 1st, 1970, 00:00:00 UTC. You can set an initial date by passing a `now` property to the `.enable()` method. This value will be used as the initial date for the mocked `Date` object. It can either be a positive integer, or another Date object.

MJS modules

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

test('mocks the Date object with initial time', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['Date'], now: 100 });
  assert.strictEqual(Date.now(), 100);

  // Advance in time will also advance the date
  context.mock.timers.tick(200);
  assert.strictEqual(Date.now(), 300);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks the Date object with initial time', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['Date'], now: 100 });
  assert.strictEqual(Date.now(), 100);

  // Advance in time will also advance the date
  context.mock.timers.tick(200);
  assert.strictEqual(Date.now(), 300);
});
```

You can use the `.setTime()` method to manually move the mocked date to another time. This method only accepts a positive integer.

Note: This method will execute any mocked timers that are in the past from the new time.

In the below example we are setting a new time for the mocked date.

MJS modules

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

test('sets the time of a date object', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['Date'], now: 100 });
  assert.strictEqual(Date.now(), 100);

  // Advance in time will also advance the date
  context.mock.timers.setTime(1000);
  context.mock.timers.tick(200);
  assert.strictEqual(Date.now(), 1200);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('sets the time of a date object', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['Date'], now: 100 });
  assert.strictEqual(Date.now(), 100);
```

```
  // Advance in time will also advance the date
  context.mock.timers.setTime(1000);
  context.mock.timers.tick(200);
  assert.strictEqual(Date.now(), 1200);
});
```

If you have any timer that's set to run in the past, it will be executed as if the `.tick()` method has been called. This is useful if you want to test time-dependent functionality that's already in the past.

MJS modules

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

test('runs timers as setTime passes ticks', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const fn = context.mock.fn();
  setTimeout(fn, 1000);

  context.mock.timers.setTime(800);
  // Timer is not executed as the time is not yet reached
  assert.strictEqual(fn.mock.callCount(), 0);
  assert.strictEqual(Date.now(), 800);

  context.mock.timers.setTime(1200);
  // Timer is executed as the time is now reached
  assert.strictEqual(fn.mock.callCount(), 1);
  assert.strictEqual(Date.now(), 1200);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('runs timers as setTime passes ticks', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const fn = context.mock.fn();
  setTimeout(fn, 1000);

  context.mock.timers.setTime(800);
  // Timer is not executed as the time is not yet reached
  assert.strictEqual(fn.mock.callCount(), 0);
  assert.strictEqual(Date.now(), 800);

  context.mock.timers.setTime(1200);
  // Timer is executed as the time is now reached
  assert.strictEqual(fn.mock.callCount(), 1);
  assert.strictEqual(Date.now(), 1200);
});
```

Using `.runAll()` will execute all timers that are currently in the queue. This will also advance the mocked date to the time of the last timer that was executed as if the time has passed.

MJS modules

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

test('runs timers as setTime passes ticks', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const fn = context.mock.fn();
  setTimeout(fn, 1000);
  setTimeout(fn, 2000);
  setTimeout(fn, 3000);

  context.mock.timers.runAll();
  // All timers are executed as the time is now reached
  assert.strictEqual(fn.mock.callCount(), 3);
  assert.strictEqual(Date.now(), 3000);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('runs timers as setTime passes ticks', (context) => {
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const fn = context.mock.fn();
  setTimeout(fn, 1000);
  setTimeout(fn, 2000);
  setTimeout(fn, 3000);

  context.mock.timers.runAll();
  // All timers are executed as the time is now reached
  assert.strictEqual(fn.mock.callCount(), 3);
  assert.strictEqual(Date.now(), 3000);
});
```

Test reporters

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

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`.
- `junit` The junit reporter outputs test results in a jUnit XML format
- `lcov` The `lcov` reporter outputs test coverage when used with the `--experimental-test-coverage` flag.

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:

MJS modules

```
import { tap, spec, dot, junit, lcov } from 'node:test/reporters';
```

CJS modules

```
const { tap, spec, dot, junit, lcov } = 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>`:

MJS modules

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

const customReporter = new Transform({
  writableObjectMode: true,
  transform(event, encoding, callback) {
    switch (event.type) {
      case 'test:dequeue':
        callback(null, `test ${event.data.name} dequeued`);
        break;
      case 'test:enqueue':
        callback(null, `test ${event.data.name} enqueued`);
        break;
      case 'test:watch:drained':
        callback(null, 'test watch queue drained');
        break;
      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':
      case 'test:stderr':
      case 'test:stdout':
        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;
```

CJS modules

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

const customReporter = new Transform({
  writableObjectMode: true,
  transform(event, encoding, callback) {
    switch (event.type) {
      case 'test:dequeue':
        callback(null, `test ${event.data.name} dequeued`);
        break;
      case 'test:enqueue':
        callback(null, `test ${event.data.name} enqueued`);
        break;
      case 'test:watch:drained':
        callback(null, 'test watch queue drained');
        break;
      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':
      case 'test:stderr':
      case 'test:stdout':
```

```
        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:

MJS modules

```
export default async function * customReporter(source) {
  for await (const event of source) {
    switch (event.type) {
      case 'test:dequeue':
        yield `test ${event.data.name} dequeued`;
        break;
      case 'test:enqueue':
        yield `test ${event.data.name} enqueued`;
        break;
      case 'test:watch:drained':
        yield 'test watch queue drained';
        break;
      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 ${event.data.name} failed\n`;
        break;
      case 'test:plan':
        yield 'test plan';
        break;
      case 'test:diagnostic':
      case 'test:stderr':
      case 'test:stdout':
        yield `${event.data.message}\n`;
        break;
      case 'test:coverage': {
        const { totalLineCount } = event.data.summary.totals;
        yield `total line count: ${totalLineCount}\n`;
```

```
        break;
      }
    }
  }
}
```

CJS modules

```
module.exports = async function * customReporter(source) {
  for await (const event of source) {
    switch (event.type) {
      case 'test:dequeue':
        yield `test ${event.data.name} dequeued`;
        break;
      case 'test:enqueue':
        yield `test ${event.data.name} enqueued`;
        break;
      case 'test:watch:drained':
        yield 'test watch queue drained';
        break;
      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 ${event.data.name} failed\n`;
        break;
      case 'test:plan':
        yield 'test plan\n';
        break;
      case 'test:diagnostic':
      case 'test:stderr':
      case 'test:stdout':
        yield `${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-destination=stdout --test-reporter-destination=file.txt copy
```

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

```
run([options])
```

▶ History

- `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.
  - `forceExit`: <boolean> Configures the test runner to exit the process once all known tests have finished executing even if the event loop would otherwise remain active. **Default:** `false`.
  - `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`.
  - `only`: <boolean> If truthy, the test context will only run tests that have the `only` option set
  - `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`.
  - `shard` <Object> Running tests in a specific shard. **Default:** `undefined`.
    - `index` <number> is a positive integer between 1 and `<total>` that specifies the index of the shard to run. This option is *required*.

    - `total` <number> is a positive integer that specifies the total number of shards to split the test files to. This option is *required*.
- Returns: <TestsStream>

Note: `shard` is used to horizontally parallelize test running across machines or processes, ideal for large-scale executions across varied environments. It's incompatible with `watch` mode, tailored for rapid code iteration by automatically rerunning tests on file changes.

MJS modules

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

run({ files: [path.resolve('./tests/test.js')] })
 .on('test:fail', () => {
   process.exitCode = 1;
 })
 .compose(tap)
 .pipe(process.stdout);
```

CJS modules

```
const { tap } = require('node:test/reporters');
const { run } = require('node:test');
const path = require('node:path');

run({ files: [path.resolve('./tests/test.js')] })
 .on('test:fail', () => {
   process.exitCode = 1;
 })
 .compose(tap)
 .pipe(process.stdout);
```

```
suite([name][, options][, fn])
```

Added in: v20.13.0

- `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> Optional configuration options for the suite. This supports the same options as `test([name][, options][, fn])`.
- `fn` <Function> | <AsyncFunction> The suite function declaring nested tests and suites. The first argument to this function is a `SuiteContext` object. **Default:** A no-op function.
- Returns: <Promise> Immediately fulfilled with `undefined`.

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

```
suite.skip([name][, options][, fn])
```

Shorthand for skipping a suite. This is the same as `suite([name], { skip: true }[, fn])`. Added in: v20.13.0

```
suite.todo([name][, options][, fn])
```

Shorthand for marking a suite as `TODO`. This is the same as `suite([name], { todo: true }[, fn])`. Added in: v20.13.0

```
suite.only([name][, options][, fn])
```

Shorthand for marking a suite as `only`. This is the same as `suite([name], { only: true }[, fn])`. Added in: v20.13.0

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

▶ History

- `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`.
  - `plan` <number> The number of assertions and subtests expected to be run in the test. If the number of assertions run in the test does not match the number specified in the plan, the test will fail. **Default:** `undefined`.
- `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> Fulfilled with `undefined` once the test completes, or immediately if the test runs within a suite.

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 fulfills once the test completes. if `test()` is called within a suite, it fulfills 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 outlive 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);
    });
  });
}); copy
```

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])
```

Alias for `suite()`.

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

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

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

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

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

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

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

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

Added in: v19.8.0, v18.15.0

Alias for `test()`.

▶ History

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])`

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

Added in: v19.8.0, v18.15.0

`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 creates a hook that runs before executing a suite.

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

`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 creates a hook that runs after executing a suite.

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

Note: The `after` hook is guaranteed to run, even if tests within the suite fail.

```
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 creates a hook that runs before each test in the current suite.

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

```
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 creates a hook that runs after each test in the current suite. The `afterEach()` hook is run even if the test fails.

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

```
}); copy
```

Class: `MockFunctionContext`

The `MockFunctionContext` class is used to inspect or manipulate the behavior of mocks created via the `MockTracker` APIs. Added in: v19.1.0, v18.13.0

`ctx.calls`

- `<Array>`

Added in: v19.1.0, v18.13.0

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()`

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

Added in: v19.1.0, v18.13.0

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)`

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

Added in: v19.1.0, v18.13.0

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);
}); copy
```

```
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);
}); copy
```

`ctx.resetCalls()`

Resets the call history of the mock function. Added in: v19.3.0, v18.13.0

`ctx.restore()`

Resets the implementation of the mock function to its original behavior. The mock can still be used after calling this function. Added in: v19.1.0, v18.13.0

Class: `MockTracker`

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. Added in: v19.1.0, v18.13.0

`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);
}); copy
```

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

This function is syntax sugar for `MockTracker.method` with `options.getter` set to `true`. Added in: v19.3.0, v18.13.0

```
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);
}); copy
```

```
mock.reset()
```

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. Added in: v19.1.0, v18.13.0

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()
```

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. Added in: v19.1.0, v18.13.0

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

This function is syntax sugar for `MockTracker.method` with `options.setter` set to `true`. Added in: v19.3.0, v18.13.0

Class: `MockTimers`

Added in: v20.4.0

Stability: 1 - Experimental

Mocking timers is a technique commonly used in software testing to simulate and control the behavior of timers, such as `setInterval` and `setTimeout`, without actually waiting for the specified time intervals.

MockTimers is also able to mock the `Date` object.

The `MockTracker` provides a top-level `timers` export which is a `MockTimers` instance.

```
timers.enable([enableOptions])
```

Enables timer mocking for the specified timers. ▶ History

- `enableOptions` <Object> Optional configuration options for enabling timer mocking. The following properties are supported:
  - `apis` <Array> An optional array containing the timers to mock. The currently supported timer values are `'setInterval'`, `'setTimeout'`, `'setImmediate'`, and `'Date'`. **Default:** `['setInterval', 'setTimeout', 'setImmediate', 'Date']`. If no array is provided, all time related APIs (`'setInterval'`, `'clearInterval'`, `'setTimeout'`, `'clearTimeout'`, and `'Date'`) will be mocked by default.
  - `now` <number> | <Date> An optional number or Date object representing the initial time (in milliseconds) to use as the value for `Date.now()`. **Default:** `0`.

Note: When you enable mocking for a specific timer, its associated clear function will also be implicitly mocked.

Note: Mocking `Date` will affect the behavior of the mocked timers as they use the same internal clock.

Example usage without setting initial time:

MJS modules

```
import { mock } from 'node:test';
mock.timers.enable({ apis: ['setInterval'] });
```

CJS modules

```
const { mock } = require('node:test');
mock.timers.enable({ apis: ['setInterval'] });
```

The above example enables mocking for the `setInterval` timer and implicitly mocks the `clearInterval` function. Only the `setInterval` and `clearInterval` functions from node:timers, node:timers/promises, and `globalThis` will be mocked.

Example usage with initial time set

MJS modules

```
import { mock } from 'node:test';
mock.timers.enable({ apis: ['Date'], now: 1000 });
```

CJS modules

```
const { mock } = require('node:test');
mock.timers.enable({ apis: ['Date'], now: 1000 });
```

Example usage with initial Date object as time set

MJS modules

```
import { mock } from 'node:test';
mock.timers.enable({ apis: ['Date'], now: new Date() });
```

CJS modules

```
const { mock } = require('node:test');
mock.timers.enable({ apis: ['Date'], now: new Date() });
```

Alternatively, if you call `mock.timers.enable()` without any parameters:

All timers (`'setInterval'` , `'clearInterval'` , `'setTimeout'` , and `'clearTimeout'`) will be mocked. The `setInterval` , `clearInterval` , `setTimeout` , and `clearTimeout` functions from `node:timers` , `node:timers/promises` , and `globalThis` will be mocked. As well as the global `Date` object.

```
timers.reset()
```

This function restores the default behavior of all mocks that were previously created by this `MockTimers` instance and disassociates the mocks from the `MockTracker` instance. Added in: v20.4.0

Note: After each test completes, this function is called on the test context's `MockTracker` .

MJS modules

```
import { mock } from 'node:test';
mock.timers.reset();
```

CJS modules

```
const { mock } = require('node:test');
mock.timers.reset();
```

```
timers[Symbol.dispose]()
```

Calls `timers.reset()`.

```
timers.tick(milliseconds)
```

Advances time for all mocked timers. Added in: v20.4.0

- `milliseconds` <number> The amount of time, in milliseconds, to advance the timers.

Note: This diverges from how `setTimeout` in Node.js behaves and accepts only positive numbers. In Node.js, `setTimeout` with negative numbers is only supported for web compatibility reasons.

The following example mocks a `setTimeout` function and by using `.tick` advances in time triggering all pending timers.

MJS modules

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

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();

  context.mock.timers.enable({ apis: ['setTimeout'] });

  setTimeout(fn, 9999);

  assert.strictEqual(fn.mock.callCount(), 0);

  // Advance in time
  context.mock.timers.tick(9999);

  assert.strictEqual(fn.mock.callCount(), 1);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();
  context.mock.timers.enable({ apis: ['setTimeout'] });

  setTimeout(fn, 9999);
  assert.strictEqual(fn.mock.callCount(), 0);
```

```
  // Advance in time
  context.mock.timers.tick(9999);

  assert.strictEqual(fn.mock.callCount(), 1);
});
```

Alternatively, the `.tick` function can be called many times

MJS modules

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

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();
  context.mock.timers.enable({ apis: ['setTimeout'] });
  const nineSecs = 9000;
  setTimeout(fn, nineSecs);

  const twoSeconds = 3000;
  context.mock.timers.tick(twoSeconds);
  context.mock.timers.tick(twoSeconds);
  context.mock.timers.tick(twoSeconds);

  assert.strictEqual(fn.mock.callCount(), 1);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();
  context.mock.timers.enable({ apis: ['setTimeout'] });
  const nineSecs = 9000;
  setTimeout(fn, nineSecs);

  const twoSeconds = 3000;
  context.mock.timers.tick(twoSeconds);
  context.mock.timers.tick(twoSeconds);
  context.mock.timers.tick(twoSeconds);

  assert.strictEqual(fn.mock.callCount(), 1);
});
```

Advancing time using `.tick` will also advance the time for any `Date` object created after the mock was enabled (if `Date` was also set to be mocked).

MJS modules

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

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();

  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  setTimeout(fn, 9999);

  assert.strictEqual(fn.mock.callCount(), 0);
  assert.strictEqual(Date.now(), 0);

  // Advance in time
  context.mock.timers.tick(9999);
  assert.strictEqual(fn.mock.callCount(), 1);
  assert.strictEqual(Date.now(), 9999);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });

  setTimeout(fn, 9999);
  assert.strictEqual(fn.mock.callCount(), 0);
  assert.strictEqual(Date.now(), 0);

  // Advance in time
  context.mock.timers.tick(9999);
  assert.strictEqual(fn.mock.callCount(), 1);
  assert.strictEqual(Date.now(), 9999);
});
```

Using clear functions

As mentioned, all clear functions from timers (`clearTimeout` and `clearInterval`) are implicity mocked. Take a look at this example using `setTimeout` :

MJS modules

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

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();
```

```
  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout'] });
  const id = setTimeout(fn, 9999);

  // Implicity mocked as well
  clearTimeout(id);
  context.mock.timers.tick(9999);

  // As that setTimeout was cleared the mock function will never be called
  assert.strictEqual(fn.mock.callCount(), 0);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', (context) => {
  const fn = context.mock.fn();

  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout'] });
  const id = setTimeout(fn, 9999);

  // Implicity mocked as well
  clearTimeout(id);
  context.mock.timers.tick(9999);

  // As that setTimeout was cleared the mock function will never be called
  assert.strictEqual(fn.mock.callCount(), 0);
});
```

Working with Node.js timers modules

Once you enable mocking timers, node:timers, node:timers/promises modules, and timers from the Node.js global context are enabled:

Note: Destructuring functions such as `import { setTimeout } from 'node:timers'` is currently not supported by this API.

MJS modules

```
import assert from 'node:assert';
import { test } from 'node:test';
import nodeTimers from 'node:timers';
import nodeTimersPromises from 'node:timers/promises';

test('mocks setTimeout to be executed synchronously without having to actually wait for it', async (context) => {
  const globalTimeoutObjectSpy = context.mock.fn();
  const nodeTimerSpy = context.mock.fn();
  const nodeTimerPromiseSpy = context.mock.fn();

  // Optionally choose what to mock
```

```
  context.mock.timers.enable({ apis: ['setTimeout'] });
  setTimeout(globalTimeoutObjectSpy, 9999);
  nodeTimers.setTimeout(nodeTimerSpy, 9999);

  const promise = nodeTimersPromises.setTimeout(9999).then(nodeTimerPromiseSpy);

  // Advance in time
  context.mock.timers.tick(9999);
  assert.strictEqual(globalTimeoutObjectSpy.mock.callCount(), 1);
  assert.strictEqual(nodeTimerSpy.mock.callCount(), 1);
  await promise;
  assert.strictEqual(nodeTimerPromiseSpy.mock.callCount(), 1);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');
const nodeTimers = require('node:timers');
const nodeTimersPromises = require('node:timers/promises');

test('mocks setTimeout to be executed synchronously without having to actually wait for it', async (con-
text) => {
  const globalTimeoutObjectSpy = context.mock.fn();
  const nodeTimerSpy = context.mock.fn();
  const nodeTimerPromiseSpy = context.mock.fn();

  // Optionally choose what to mock
  context.mock.timers.enable({ apis: ['setTimeout'] });
  setTimeout(globalTimeoutObjectSpy, 9999);
  nodeTimers.setTimeout(nodeTimerSpy, 9999);

  const promise = nodeTimersPromises.setTimeout(9999).then(nodeTimerPromiseSpy);

  // Advance in time
  context.mock.timers.tick(9999);
  assert.strictEqual(globalTimeoutObjectSpy.mock.callCount(), 1);
  assert.strictEqual(nodeTimerSpy.mock.callCount(), 1);
  await promise;
  assert.strictEqual(nodeTimerPromiseSpy.mock.callCount(), 1);
});
```

In Node.js, `setInterval` from node:timers/promises is an `AsyncGenerator` and is also supported by this API:

MJS modules

```
import assert from 'node:assert';
import { test } from 'node:test';
import nodeTimersPromises from 'node:timers/promises';
test('should tick five times testing a real use case', async (context) => {
  context.mock.timers.enable({ apis: ['setInterval'] });

  const expectedIterations = 3;
```

```
  const interval = 1000;
  const startedAt = Date.now();
  async function run() {
    const times = [];
    for await (const time of nodeTimersPromises.setInterval(interval, startedAt)) {
      times.push(time);
      if (times.length === expectedIterations) break;
    }
    return times;
  }

  const r = run();
  context.mock.timers.tick(interval);
  context.mock.timers.tick(interval);
  context.mock.timers.tick(interval);

  const timeResults = await r;
  assert.strictEqual(timeResults.length, expectedIterations);
  for (let it = 1; it < expectedIterations; it++) {
    assert.strictEqual(timeResults[it - 1], startedAt + (interval * it));
  }
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');
const nodeTimersPromises = require('node:timers/promises');
test('should tick five times testing a real use case', async (context) => {
  context.mock.timers.enable({ apis: ['setInterval'] });

  const expectedIterations = 3;
  const interval = 1000;
  const startedAt = Date.now();
  async function run() {
    const times = [];
    for await (const time of nodeTimersPromises.setInterval(interval, startedAt)) {
      times.push(time);
      if (times.length === expectedIterations) break;
    }
    return times;
  }

  const r = run();
  context.mock.timers.tick(interval);
  context.mock.timers.tick(interval);
  context.mock.timers.tick(interval);

  const timeResults = await r;
```

```
  assert.strictEqual(timeResults.length, expectedIterations);
  for (let it = 1; it < expectedIterations; it++) {
    assert.strictEqual(timeResults[it - 1], startedAt + (interval * it));
  }
});
```

```
timers.runAll()
```

Triggers all pending mocked timers immediately. If the `Date` object is also mocked, it will also advance the `Date` object to the furthest timer's time. Added in: v20.4.0

The example below triggers all pending timers immediately, causing them to execute without any delay.

MJS modules

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

test('runAll functions following the given order', (context) => {
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const results = [];
  setTimeout(() => results.push(1), 9999);

  // Notice that if both timers have the same timeout,
  // the order of execution is guaranteed
  setTimeout(() => results.push(3), 8888);
  setTimeout(() => results.push(2), 8888);

  assert.deepStrictEqual(results, []);

  context.mock.timers.runAll();
  assert.deepStrictEqual(results, [3, 2, 1]);
  // The Date object is also advanced to the furthest timer's time
  assert.strictEqual(Date.now(), 9999);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('runAll functions following the given order', (context) => {
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const results = [];
  setTimeout(() => results.push(1), 9999);

  // Notice that if both timers have the same timeout,
  // the order of execution is guaranteed
  setTimeout(() => results.push(3), 8888);
  setTimeout(() => results.push(2), 8888);

  assert.deepStrictEqual(results, []);
```

```
  context.mock.timers.runAll();
  assert.deepStrictEqual(results, [3, 2, 1]);
  // The Date object is also advanced to the furthest timer's time
  assert.strictEqual(Date.now(), 9999);
});
```

Note: The `runAll()` function is specifically designed for triggering timers in the context of timer mocking. It does not have any effect on real-time system clocks or actual timers outside of the mocking environment.

```
timers.setTime(milliseconds)
```

Sets the current Unix timestamp that will be used as reference for any mocked `Date` objects. Added in: v20.11.0

MJS modules

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

test('runAll functions following the given order', (context) => {
  const now = Date.now();
  const setTime = 1000;
  // Date.now is not mocked
  assert.deepStrictEqual(Date.now(), now);

  context.mock.timers.enable({ apis: ['Date'] });
  context.mock.timers.setTime(setTime);
  // Date.now is now 1000
  assert.strictEqual(Date.now(), setTime);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('setTime replaces current time', (context) => {
  const now = Date.now();
  const setTime = 1000;
  // Date.now is not mocked
  assert.deepStrictEqual(Date.now(), now);

  context.mock.timers.enable({ apis: ['Date'] });
  context.mock.timers.setTime(setTime);
  // Date.now is now 1000
  assert.strictEqual(Date.now(), setTime);
});
```

Dates and Timers working together

Dates and timer objects are dependent on each other. If you use `setTime()` to pass the current time to the mocked `Date` object, the set timers with `setTimeout` and `setInterval` will **not** be affected.

However, the `tick` method **will** advanced the mocked `Date` object.

MJS modules

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

test('runAll functions following the given order', (context) => {
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const results = [];
  setTimeout(() => results.push(1), 9999);

  assert.deepStrictEqual(results, []);
  context.mock.timers.setTime(12000);
  assert.deepStrictEqual(results, []);
  // The date is advanced but the timers don't tick
  assert.strictEqual(Date.now(), 12000);
});
```

CJS modules

```
const assert = require('node:assert');
const { test } = require('node:test');

test('runAll functions following the given order', (context) => {
  context.mock.timers.enable({ apis: ['setTimeout', 'Date'] });
  const results = [];
  setTimeout(() => results.push(1), 9999);

  assert.deepStrictEqual(results, []);
  context.mock.timers.setTime(12000);
  assert.deepStrictEqual(results, []);
  // The date is advanced but the timers don't tick
  assert.strictEqual(Date.now(), 12000);
});
```

Class: `TestsStream`

- Extends <Readable>

▶ History

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

Some of the events are guaranteed to be emitted in the same order as the tests are defined, while others are emitted in the order that the tests execute.

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.
      - `functions` <Array> An array of functions representing function coverage.
        - `name` <string> The name of the function.
        - `line` <number> The line number where the function is defined.
        - `count` <number> The number of times the function was called.
      - `branches` <Array> An array of branches representing branch coverage.
        - `line` <number> The line number where the branch is defined.
        - `count` <number> The number of times the branch was taken.
      - `lines` <Array> An array of lines representing line numbers and the number of times they were covered.
        - `line` <number> The line number.
        - `count` <number> The number of times the line was covered.
    - `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:complete'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `details` <Object> Additional execution metadata.
    - `passed` <boolean> Whether the test passed or not.
    - `duration_ms` <number> The duration of the test in milliseconds.
    - `error` <Error> | <undefined> An error wrapping the error thrown by the test if it did not pass.
      - `cause` <Error> The actual error thrown by the test.
    - `type` <string> | <undefined> The type of the test, used to denote whether this is a suite.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `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 completes its execution. This event is not emitted in the same order as the tests are defined. The corresponding declaration ordered events are `'test:pass'` and `'test:fail'`.

Event: `'test:dequeue'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `name` <string> The test name.
  - `nesting` <number> The nesting level of the test.

Emitted when a test is dequeued, right before it is executed. This event is not guaranteed to be emitted in the same order as the tests are defined. The corresponding declaration ordered event is `'test:start'`.

Event: `'test:diagnostic'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `message` <string> The diagnostic message.
  - `nesting` <number> The nesting level of the test.

Emitted when `context.diagnostic` is called. This event is guaranteed to be emitted in the same order as the tests are defined.

Event: `'test:enqueue'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `name` <string> The test name.
  - `nesting` <number> The nesting level of the test.

Emitted when a test is enqueued for execution.

Event: `'test:fail'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `details` <Object> Additional execution metadata.
    - `duration_ms` <number> The duration of the test in milliseconds.
    - `error` <Error> An error wrapping the error thrown by the test.
      - `cause` <Error> The actual error thrown by the test.
    - `type` <string> | <undefined> The type of the test, used to denote whether this is a suite.

- `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
- `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
- `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. This event is guaranteed to be emitted in the same order as the tests are defined. The corresponding execution ordered event is `'test:complete'`.

Event: `'test:pass'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `details` <Object> Additional execution metadata.
    - `duration_ms` <number> The duration of the test in milliseconds.
    - `type` <string> | <undefined> The type of the test, used to denote whether this is a suite.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `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. This event is guaranteed to be emitted in the same order as the tests are defined. The corresponding execution ordered event is `'test:complete'`.

Event: `'test:plan'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.

- `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. This event is guaranteed to be emitted in the same order as the tests are defined.

Event: `'test:start'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> | <undefined> The path of the test file, `undefined` if test was run through the REPL.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `name` <string> The test name.
  - `nesting` <number> The nesting level of the test.

Emitted when a test starts reporting its own and its subtests status. This event is guaranteed to be emitted in the same order as the tests are defined. The corresponding execution ordered event is `'test:dequeue'`.

Event: `'test:stderr'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> The path of the test file.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.
  - `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. This event is not guaranteed to be emitted in the same order as the tests are defined.

Event: `'test:stdout'`

- `data` <Object>
  - `column` <number> | <undefined> The column number where the test is defined, or `undefined` if the test was run through the REPL.
  - `file` <string> The path of the test file.
  - `line` <number> | <undefined> The line number where the test is defined, or `undefined` if the test was run through the REPL.

- `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. This event is not guaranteed to be emitted in the same order as the tests are defined.

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. ▶ History

`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');
    },
  );
}); copy
```

```
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:** `Infinity` .

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');
}); copy
```

```
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');
    },
  );
}); copy
```

```
context.diagnostic(message)
```

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

Added in: v18.0.0, v16.17.0

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');
}); copy
```

```
context.name
```

The name of the test.

Added in: v18.8.0, v16.18.0

```
context.plan(count)
```

Added in: v20.15.0

Stability: 1 - Experimental

- `count` <number> The number of assertions and subtests that are expected to run.

This function is used to set the number of assertions and subtests that are expected to run within the test. If the number of assertions and subtests that run does not match the expected count, the test will fail.

> Note: To make sure assertions are tracked, `t.assert` must be used instead of `assert` directly.

```
test('top level test', (t) => {
  t.plan(2);
  t.assert.ok('some relevant assertion here');
  t.subtest('subtest', () => {});
}); copy
```

When working with asynchronous code, the `plan` function can be used to ensure that the correct number of assertions are run:

```
test('planning with streams', (t, done) => {
  function* generate() {
    yield 'a';
    yield 'b';
    yield 'c';
  }
  const expected = ['a', 'b', 'c'];
  t.plan(expected.length);
  const stream = Readable.from(generate());
  stream.on('data', (chunk) => {
```

```
    t.assert.strictEqual(chunk, expected.shift());
  });

  stream.on('end', () => {
    done();
  });
}); copy
```

```
context.runOnly(shouldRunOnlyTests)
```

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

Added in: v18.0.0, v16.17.0

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 }),
  ]);
}); copy
```

```
context.signal
```

- Type: <AbortSignal>

Added in: v18.7.0, v16.17.0

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 });
}); copy
```

```
context.skip([message])
```

- `message` <string> Optional skip message.

Added in: v18.0.0, v16.17.0

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 additional logic.
  t.skip('this is skipped');
}); copy
```

```
context.todo([message])
```

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

Added in: v18.0.0, v16.17.0

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');
}); copy
```

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

▶ History

- `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:** `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`.
  - `plan` <number> The number of assertions and subtests expected to be run in the test. If the number of assertions run in the test does not match the number specified in the plan, the test will fail. **Default:** `undefined`.
- `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> Fulfilled 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, plan: 4 },
    (t) => {
```

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

Class: `SuiteContext`

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. Added in: v18.7.0, v16.17.0

`context.name`

The name of the suite. Added in: v18.8.0, v16.18.0

`context.signal`

- Type: <AbortSignal>

Added in: v18.7.0, v16.17.0

Can be used to abort test subtasks when the test has been aborted.