Test runner

> *Stability: 1 - Experimental*

The `node:test` module facilitates the creation of JavaScript tests that report results in TAP format. To access it:

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

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

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

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

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

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

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

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

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

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

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

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

```
});

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

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

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

As a test file executes, TAP is written to the standard output of the Node.js process. This output can be interpreted by any test harness that understands the TAP format. If any tests fail, the process exit code is set to `1`.

Subtests

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

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

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

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

Skipping tests

Individual tests can be skipped by passing the `skip` option to the test, or by calling the test context's `skip()` method. Both of these options support including a message that is displayed in the TAP output 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');
});
```

`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');
});
```

Extraneous asynchronous activity

Once a test function finishes executing, the TAP results are output 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 in the top level of the file's TAP output.

The second `setImmediate()` creates an `uncaughtException` event. `uncaughtException` and `unhandledRejection` events originating from a completed test are handled by the `test` module and reported as diagnostic warnings in the top level of the file's TAP output.

```
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.
});
```

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

- `name` {string} The name of the test, which is displayed when reporting test results. **Default:** The `name` property of `fn`, or `'<anonymous>'` if `fn` does not have a name.
- `options` {Object} Configuration options for the test. The following properties are supported:
  - `concurrency` {number} The number of tests that can be run at the same time. If unspecified, subtests inherit this value from their parent. **Default:** `1`.
  - `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`.
  - `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`.

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

The `test()` function is the value imported from the `test` module. Each invocation of this function results in the creation of a test point in the TAP output.

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 TAP diagnostic information, or creating subtests.

`test()` returns a `Promise` that resolves once the test completes. 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);
    });
  });
});
```

Class: `TestContext`

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

`context.diagnostic(message)`

- `message` {string} Message to be displayed as a TAP diagnostic.

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

`context.runOnly(shouldRunOnlyTests)`

- `shouldRunOnlyTests` {boolean} Whether or not to run `only` tests.

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

`context.skip([message])`

- `message` {string} Optional skip message to be displayed in TAP output.

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

`context.todo([message])`

- `message` {string} Optional `TODO` message to be displayed in TAP output.

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

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

- `name` {string} The name of the subtest, which is displayed when reporting test results. **Default:** The `name` property of `fn`, or `'<anonymous>'` if `fn` does not have a name.
- `options` {Object} Configuration options for the subtest. The following properties are supported:
  - `concurrency` {number} The number of tests that can be run at the same time. If unspecified, subtests inherit this value from their parent. **Default:** `1`.
  - `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`.
  - `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`.
- `fn` {Function|AsyncFunction} The function under test. This first argument to this function is a `TestContext` object. If the test uses callbacks, the callback function is passed as the second argument. **Default:** A no-op function.
- Returns: {Promise} Resolved with `undefined` once the test completes.

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