How to write a test for the Node.js project

What is a test?

Most tests in Node.js core are JavaScript programs that exercise a functionality provided by Node.js and check that it behaves as expected. Tests should exit with code `0` on success. A test will fail if:

- It exits by setting `process.exitCode` to a non-zero number.
  - This is usually done by having an assertion throw an uncaught Error.
  - Occasionally, using `process.exit(code)` may be appropriate.
- It never exits. In this case, the test runner will terminate the test because it sets a maximum time limit.

Add tests when:

- Adding new functionality.
- Fixing regressions and bugs.
- Expanding test coverage.

Test directory structure

See directory structure overview for outline of existing test and locations. When deciding on whether to expand an existing test file or create a new one, consider going through the files related to the subsystem. For example, look for `test-streams` when writing a test for `lib/streams.js`.

Test structure

Let's analyze this basic test from the Node.js test suite:

```
'use strict';                                                          // 1
const common = require('../common');                                   // 2
const fixtures = require('../common/fixtures');                        // 3

// This test ensures that the http-parser can handle UTF-8 characters  // 5
// in the http header.                                                 // 6

const assert = require('assert');                                      // 8
const http = require('http');                                          // 9

const server = http.createServer(common.mustCall((req, res) => {       // 11
  res.end('ok');                                                       // 12
}));                                                                   // 13
server.listen(0, () => {                                               // 14
  http.get({                                                           // 15
    port: server.address().port,                                       // 16
    headers: { 'Test': 'Düsseldorf' }                                  // 17
```

```
  }, common.mustCall((res) => {                                    // 18
    assert.strictEqual(res.statusCode, 200);                       // 19
    server.close();                                                // 20
  }));                                                             // 21
});                                                                // 22
// ...                                                             // 23
```

Lines 1-3

```
'use strict';
const common = require('../common');
const fixtures = require('../common/fixtures');
```

The first line enables strict mode. All tests should be in strict mode unless the nature of the test requires that the test run without it.

The second line loads the `common` module. The `common` module is a helper module that provides useful tools for the tests. Some common functionality has been extracted into submodules, which are required separately like the fixtures module here.

Even if a test uses no functions or other properties exported by `common`, the test should still include the `common` module before any other modules. This is because the `common` module includes code that will cause a test to fail if the test leaks variables into the global space. In situations where a test uses no functions or other properties exported by `common`, include it without assigning it to an identifier:

```
require('../common');
```

Lines 5-6

```
// This test ensures that the http-parser can handle UTF-8 characters
// in the http header.
```

A test should start with a comment containing a brief description of what it is designed to test.

Lines 8-9

```
const assert = require('assert');
const http = require('http');
```

The test checks functionality in the `http` module.

Most tests use the `assert` module to confirm expectations of the test.

The require statements are sorted in ASCII order (digits, upper case, `_`, lower case).

Lines 11-22

```
const server = http.createServer(common.mustCall((req, res) => {
  res.end('ok');
}));
server.listen(0, () => {
  http.get({
    port: server.address().port,
    headers: { 'Test': 'Düsseldorf' }
  }, common.mustCall((res) => {
    assert.strictEqual(res.statusCode, 200);
    server.close();
  }));
});
```

This is the body of the test. This test is simple, it just tests that an HTTP server accepts `non-ASCII` characters in the headers of an incoming request. Interesting things to notice:

- If the test doesn't depend on a specific port number, then always use 0 instead of an arbitrary value, as it allows tests to run in parallel safely, as the operating system will assign a random port. If the test requires a specific port, for example if the test checks that assigning a specific port works as expected, then it is ok to assign a specific port number.
- The use of `common.mustCall` to check that some callbacks/listeners are called.
- The HTTP server closes once all the checks have run. This way, the test can exit gracefully. Remember that for a test to succeed, it must exit with a status code of 0.

General recommendations

Timers

Avoid timers unless the test is specifically testing timers. There are multiple reasons for this. Mainly, they are a source of flakiness. For a thorough explanation go here.

In the event a test needs a timer, consider using the `common.platformTimeout()` method. It allows setting specific timeouts depending on the platform:

```
const timer = setTimeout(fail, common.platformTimeout(4000));
```

will create a 4-second timeout on most platforms but a longer timeout on slower platforms.

The *common* API

Make use of the helpers from the `common` module as much as possible. Please refer to the common file documentation for the full details of the helpers.

common.mustCall One interesting case is `common.mustCall`. The use of `common.mustCall` may avoid the use of extra variables and the corresponding assertions. Let's explain this with a real test from the test suite.

```
'use strict';
require('../common');
const assert = require('assert');
const http = require('http');

let request = 0;
let listening = 0;
let response = 0;
process.on('exit', () => {
  assert.equal(request, 1, 'http server "request" callback was not called');
  assert.equal(listening, 1, 'http server "listening" callback was not called');
  assert.equal(response, 1, 'http request "response" callback was not called');
});

const server = http.createServer((req, res) => {
  request++;
  res.end();
}).listen(0, () => {
  listening++;
  const options = {
    agent: null,
    port: server.address().port
  };
  http.get(options, (res) => {
    response++;
    res.resume();
    server.close();
  });
});
```

This test could be greatly simplified by using `common.mustCall` like this:

```
'use strict';
const common = require('../common');
const http = require('http');

const server = http.createServer(common.mustCall((req, res) => {
  res.end();
```

```
})).listen(0, common.mustCall(() => {
  const options = {
    agent: null,
    port: server.address().port
  };
  http.get(options, common.mustCall((res) => {
    res.resume();
    server.close();
  }));
}));
```

Note: Many functions invoke their callback with an `err` value as the first argument. It is not a good idea to simply pass `common.mustCall()` to those because `common.mustCall()` will ignore the error. Use `common.mustSucceed()` instead.

Countdown module The common Countdown module provides a simple countdown mechanism for tests that require a particular action to be taken after a given number of completed tasks (for instance, shutting down an HTTP server after a specific number of requests).

```
const Countdown = require('../common/countdown');

const countdown = new Countdown(2, () => {
  console.log('.');
});

countdown.dec();
countdown.dec(); // The countdown callback will be invoked now.
```

Testing promises When writing tests involving promises, it is generally good to wrap the `onFulfilled` handler, otherwise the test could successfully finish if the promise never resolves (pending promises do not keep the event loop alive). Node.js automatically crashes - and hence, the test fails - in the case of an `unhandledRejection` event.

```
const common = require('../common');
const assert = require('assert');
const fs = require('fs').promises;

// Wrap the `onFulfilled` handler in `common.mustCall()`.
fs.readFile('test-file').then(
  common.mustCall(
    (content) => assert.strictEqual(content.toString(), 'test2')
  ));
```

Flags

Some tests will require running Node.js with specific command line flags set. To accomplish this, add a `// Flags:` comment in the preamble of the test followed by the flags. For example, to allow a test to require some of the `internal/*` modules, add the `--expose-internals` flag. A test that would require `internal/freelist` could start like this:

```
'use strict';

// Flags: --expose-internals

require('../common');
const assert = require('assert');
const freelist = require('internal/freelist');
```

In specific scenarios it may be useful to get a hold of `primordials` or `internalBinding()`. You can do so using

```
node --expose-internals -r internal/test/binding lib/fs.js
```

This only works if you preload `internal/test/binding` by command line flag.

Assertions

When writing assertions, prefer the strict versions:

- `assert.strictEqual()` over `assert.equal()`
- `assert.deepStrictEqual()` over `assert.deepEqual()`

When using `assert.throws()`, if possible, provide the full error message:

```
assert.throws(
  () => {
    throw new Error('Wrong value');
  },
  /^Error: Wrong value$/ // Instead of something like /Wrong value/
);
```

Console output

Output written by tests to stdout or stderr, such as with `console.log()` or `console.error()`, can be useful when writing tests, as well as for debugging them during later maintenance. The output will be suppressed by the test runner (`./tools/test.py`) unless the test fails, but will always be displayed when running tests directly with `node`. For failing tests, the test runner will include the output along with the failed test assertion in the test report.

Some output can help debugging by giving context to test failures. For example, when troubleshooting tests that timeout in CI. With no log statements, we have no idea where the test got hung up.

There have been cases where tests fail without `console.log()`, and then pass when its added, so be cautious about its use, particularly in tests of the I/O and streaming APIs.

Excessive use of console output is discouraged as it can overwhelm the display, including the Jenkins console and test report displays. Be particularly cautious of output in loops, or other contexts where output may be repeated many times in the case of failure.

In some tests, it can be unclear whether a `console.log()` statement is required as part of the test (message tests, tests that check output from child processes, etc.), or is there as a debug aide. If there is any chance of confusion, use comments to make the purpose clear.

ES.Next features

For performance considerations, we only use a selected subset of ES.Next features in JavaScript code in the `lib` directory. However, when writing tests, for the ease of backporting, it is encouraged to use those ES.Next features that can be used directly without a flag in all maintained branches. node.green lists available features in each release, such as:

- `let` and `const` over `var`
- Template literals over string concatenation
- Arrow functions when appropriate

Naming test files

Test files are named using kebab casing. The first component of the name is `test`. The second is the module or subsystem being tested. The third is usually the method or event name being tested. Subsequent components of the name add more information about what is being tested.

For example, a test for the `beforeExit` event on the `process` object might be named `test-process-before-exit.js`. If the test specifically checked that arrow functions worked correctly with the `beforeExit` event, then it might be named `test-process-before-exit-arrow-functions.js`.

Imported tests

Web platform tests

See `test/wpt` for more information.

C++ unit test

C++ code can be tested using Google Test. Most features in Node.js can be tested using the methods described previously in this document. But there are

cases where these might not be enough, for example writing code for Node.js that will only be called when Node.js is embedded.

Adding a new test

The unit test should be placed in `test/cctest` and be named with the prefix `test` followed by the name of unit being tested. For example, the code below would be placed in `test/cctest/test_env.cc`:

```
#include "gtest/gtest.h"
#include "node_test_fixture.h"
#include "env.h"
#include "node.h"
#include "v8.h"

static bool called_cb = false;
static void at_exit_callback(void* arg);

class EnvTest : public NodeTestFixture { };

TEST_F(EnvTest, RunAtExit) {
  v8::HandleScope handle_scope(isolate_);
  v8::Local<v8::Context> context = v8::Context::New(isolate_);
  node::IsolateData* isolateData = node::CreateIsolateData(isolate_, uv_default_loop());
  Argv argv{"node", "-e", ";"};
  auto env = node::CreateEnvironment(isolateData, context, 1, *argv, 2, *argv);
  node::AtExit(env, at_exit_callback);
  node::RunAtExit(env);
  EXPECT_TRUE(called_cb);
}

static void at_exit_callback(void* arg) {
  called_cb = true;
}
```

Next add the test to the `sources` in the `cctest` target in node.gyp:

```
'sources': [
  'test/cctest/test_env.cc',
  ...
],
```

The only sources that should be included in the cctest target are actual test or helper source files. There might be a need to include specific object files that are compiled by the `node` target and this can be done by adding them to the `libraries` section in the cctest target.

The test can be executed by running the `cctest` target:

```
$ make cctest
```

A filter can be applied to run single/multiple test cases:

```
$ make cctest GTEST_FILTER=EnvironmentTest.AtExitWithArgument
```

`cctest` can also be run directly which can be useful when debugging:

```
$ out/Release/cctest --gtest_filter=EnvironmentTest.AtExit\*
```

Node.js test fixture

There is a test fixture named `node_test_fixture.h` which can be included by unit tests. The fixture takes care of setting up the Node.js environment and tearing it down after the tests have finished.

It also contains a helper to create arguments to be passed into Node.js. It will depend on what is being tested if this is required or not.

Test coverage

To generate a test coverage report, see the Test Coverage section of the Building guide.

Nightly coverage reports for the Node.js master branch are available at https://coverage.nodejs.org/.