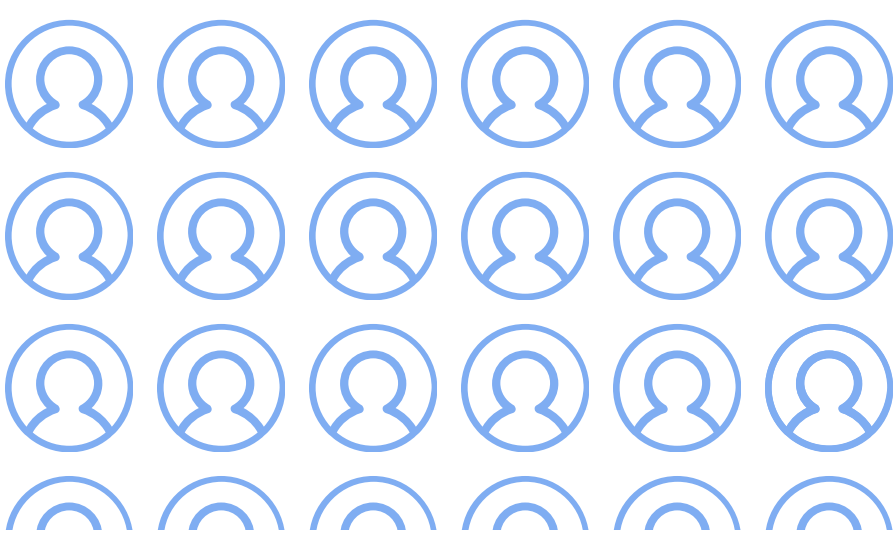
simple, flexible, fun

Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing *simple* and *fun*. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases. Hosted on GitHub.

BACKERS

Find Mocha helpful? Become a backer and support Mocha with a monthly donation.



SPONSORS

Use Mocha at Work? Ask your manager or marketing team if they'd help support our project. Your company's logo will also be displayed on npmjs.com and our GitHub repository.

FEATURES

- browser support
- simple async support, including promises
- test coverage reporting
- string diff support
- javascript API for running tests
- proper exit status for CI support etc
- auto-detects and disables coloring for non-ttys
- maps uncaught exceptions to the correct test case
- async test timeout support
- test retry support
- test-specific timeouts

- growl notification support
- reports test durations
- highlights slow tests
- file watcher support
- global variable leak detection
- optionally run tests that match a regexp
- auto-exit to prevent "hanging" with an active loop
- easily meta-generate suites & test-cases
- mocha.opts file support
- clickable suite titles to filter test execution
- node debugger support
- detects multiple calls to `done()`
- use any assertion library you want
- extensible reporting, bundled with 9+ reporters
- extensible test DSLs or "interfaces"
- before, after, before each, after each hooks
- arbitrary transpiler support (coffee-script etc)
- TextMate bundle
- and more!

TABLE OF CONTENTS

INSTALLATION

Install with npm globally:

```
$ npm install --global mocha
```

or as a development dependency for your project:

```
$ npm install --save-dev mocha
```

> Mocha currently requires Node.js v6.x or newer.

GETTING STARTED

```
$ npm install mocha
```

```
$ mkdir test
$ $EDITOR test/test.js # or open with your favorite edi
```

In your editor:

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present'
      assert.equal([1,2,3].indexOf(4), -1);
    });
  });
});
```

Back in the terminal:

```
$ ./node_modules/mocha/bin/mocha

  Array
    #indexOf()
      ✓ should return -1 when the value is not present



  1 passing (9ms)
```

Set up a test script in package.json:

```
"scripts": {
    "test": "mocha"
  }
```

Then run tests with:

```
$ npm test
```

DETECTS MULTIPLE CALLS TO `DONE()`

If you use callback-based async tests, Mocha will throw an error if `done()` is called multiple times. This is handy for catching accidental double callbacks.

```
it('double done', function(done) {
  // Calling `done()` twice is an error
  setImmediate(done);
  setImmediate(done);
});
```

Running the above test will give you the below error message:

```
$ ./node_modules/.bin/mocha mocha.test.js



  ✓ double done
  1) double done


  1 passing (6ms)
  1 failing


  1) double done:
     Error: done() called multiple times
      at Object.<anonymous> (mocha.test.js:1:63)
      at require (internal/module.js:11:18)
      at Array.forEach (<anonymous>)
```

```
      at startup (bootstrap_node.js:187:16)
      at bootstrap_node.js:608:3
```

ASSERTIONS

Mocha allows you to use any assertion library you wish. In the above example, we're using Node.js' built-in assert module–but generally, if it throws an `Error`, it will work! This means you can use libraries such as:

- should.js - BDD style shown throughout these docs
- expect.js - `expect()` style assertions
- chai - `expect()`, `assert()` and `should`-style assertions
- better-assert - C-style self-documenting `assert()`
- unexpected - "the extensible BDD assertion toolkit"

ASYNCHRONOUS CODE

Testing asynchronous code with Mocha could not be simpler! Simply invoke the callback when your test is complete. By adding a callback (usually named `done`) to `it()`, Mocha will know that it should wait for this function to be called to complete the test. This callback accepts both an `Error` instance (or subclass thereof) *or* a falsy value; anything else will cause a failed test.

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
```

```
      var user = new User('Luna');
      user.save(function(err) {
        if (err) done(err);
        else done();
      });
    });
  });
});
```

To make things even easier, the `done()` callback also accepts an `Error` instance (i.e. `new Error()`), so we may use this directly:

```
describe('User', function() {
  describe('#save()', function() {
    it('should save without error', function(done) {
      var user = new User('Luna');
      user.save(done);
    });
  });
});
```

WORKING WITH PROMISES

Alternately, instead of using the `done()` callback, you may return a Promise. This is useful if the APIs you are testing return promises instead of taking callbacks:

```
beforeEach(function() {
  return db.clear()
    .then(function() {
      return db.save([tobi, loki, jane]);
    });
```

```
});

describe('#find()', function() {
  it('respond with matching records', function() {
    return db.find({ type: 'User' }).should.eventually.
  });
});
```

> The latter example uses Chai as Promised for fluent promise assertions.

In Mocha v3.0.0 and newer, returning a `Promise` *and* calling `done()` will result in an exception, as this is generally a mistake:

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

it('should complete this test', function (done) {
  return new Promise(function (resolve) {
    assert.ok(true);
    resolve();
  })
    .then(done);
});
```

The above test will fail with `Error: Resolution method is overspecified. Specify a callback *or* return a Promise; not both.`. In versions older than v3.0.0, the call to `done()` is effectively ignored.

USING ASYNC / AWAIT

If your JS environment supports async / await you can also write asynchronous tests like this:

```
beforeEach(async function() {
  await db.clear();
  await db.save([tobi, loki, jane]);
});

describe('#find()', function() {
  it('responds with matching records', async function()
    const users = await db.find({ type: 'User' });
    users.should.have.length(3);
  });
});
```

SYNCHRONOUS CODE

When testing synchronous code, omit the callback and Mocha will automatically continue on to the next test.

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present'
      [1,2,3].indexOf(5).should.equal(-1);
      [1,2,3].indexOf(0).should.equal(-1);
    });
  });
});
```

ARROW FUNCTIONS

Passing arrow functions ("lambdas") to Mocha is discouraged.

Lambdas lexically bind `this` and cannot access the Mocha context. For example, the following code will fail:

```
describe('my suite', () => {
  it('my test', () => {
    // should set the timeout of this test to 1000 ms;
    this.timeout(1000);
    assert.ok(true);
  });
});
```

If you do not need to use Mocha's context, lambdas should work. However, the result will be more difficult to refactor if the need eventually arises.

HOOKS

With its default "BDD"-style interface, Mocha provides the hooks `before()`, `after()`, `beforeEach()`, and `afterEach()`. These should be used to set up preconditions and clean up after your tests.

```
describe('hooks', function() {

  before(function() {
    // runs before all tests in this block
  });

  after(function() {
    // runs after all tests in this block
  });
```

```
  beforeEach(function() {
    // runs before each test in this block
  });

  afterEach(function() {
    // runs after each test in this block
  });

  // test cases
});
```

> Tests can appear before, after, or interspersed with your hooks. Hooks will run in the order they are defined, as appropriate; all `before()` hooks run (once), then any `beforeEach()` hooks, tests, any `afterEach()` hooks, and finally `after()` hooks (once).

DESCRIBING HOOKS

Any hook can be invoked with an optional description, making it easier to pinpoint errors in your tests. If a hook is given a named function, that name will be used if no description is supplied.

```
beforeEach(function() {
  // beforeEach hook
});

beforeEach(function namedFun() {
  // beforeEach:namedFun
});
```

```
beforeEach('some description', function() {
  // beforeEach:some description
});
```

ASYNCHRONOUS HOOKS

All hooks (`before()`, `after()`, `beforeEach()`, `afterEach()`) may be sync or async as well, behaving much like a regular test-case. For example, you may wish to populate database with dummy content before each test:

```
describe('Connection', function() {
  var db = new Connection,
    tobi = new User('tobi'),
    loki = new User('loki'),
    jane = new User('jane');

  beforeEach(function(done) {
    db.clear(function(err) {
      if (err) return done(err);
      db.save([tobi, loki, jane], done);
    });
  });

  describe('#find()', function() {
    it('respond with matching records', function(done)
      db.find({type: 'User'}, function(err, res) {
        if (err) return done(err);
        res.should.have.length(3);
        done();
      });
    });
  });
```

```
});
```

ROOT-LEVEL HOOKS

You may also pick any file and add “root”-level hooks. For example, add `beforeEach()` outside of all `describe()` blocks. This will cause the callback to `beforeEach()` to run before any test case, regardless of the file it lives in (this is because Mocha has an *implied* `describe()` block, called the “root suite”).

```
beforeEach(function() {
  console.log('before every test in every file');
});
```

DELAYED ROOT SUITE

If you need to perform asynchronous operations before any of your suites are run, you may delay the root suite. Run `mocha` with the `--delay` flag. This will attach a special callback function, `run()`, to the global context:

```
setTimeout(function() {
  // do some setup

  describe('my suite', function() {
    // ...
  });

  run();
}, 5000);
```

PENDING TESTS

"Pending"–as in "someone should write these test cases eventually"–test-cases are simply those *without* a callback:

```
describe('Array', function() {
  describe('#indexOf()', function() {
    // pending test below
    it('should return -1 when the value is not present'
  });
});
```

Pending tests will be included in the test results, and marked as pending. A pending test is not considered a failed test.

EXCLUSIVE TESTS

The exclusivity feature allows you to run *only* the specified suite or test-case by appending `.only()` to the function. Here's an example of executing only a particular suite:

```
describe('Array', function() {
  describe.only('#indexOf()', function() {
    // ...
  });
});
```

Note: All nested suites will still be executed.

Here's an example of executing an individual test case:

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it.only('should return -1 unless present', function
      // ...
    });

    it('should return the index when present', function
      // ...
    });
  });
});
```

Previous to v3.0.0, `.only()` used string matching to decide which tests to execute. As of v3.0.0, this is no longer the case. In v3.0.0 or newer, `.only()` can be used multiple times to define a subset of tests to run:

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it.only('should return -1 unless present', function
      // this test will be run
    });

    it.only('should return the index when present', fun
      // this test will also be run
    });

    it('should return -1 if called with a non-Array con
      // this test will not be run
    });
  });
});
```

You may also choose multiple suites:

```
describe('Array', function() {
  describe.only('#indexOf()', function() {
    it('should return -1 unless present', function() {
      // this test will be run
    });

    it('should return the index when present', function
      // this test will also be run
    });
  });

  describe.only('#concat()', function () {
    it('should return a new Array', function () {
      // this test will also be run
    });
  });

  describe('#slice()', function () {
    it('should return a new Array', function () {
      // this test will not be run
    });
  });
});
```

But *tests will have precedence*:

```
describe('Array', function() {
  describe.only('#indexOf()', function() {
    it.only('should return -1 unless present', function
      // this test will be run
    });

    it('should return the index when present', function
```

```
      // this test will not be run
    });
  });
});
```

Note: Hooks, if present, will still be executed.

> Be mindful not to commit usages of `.only()` to version control, unless you really mean it! To do so one can run mocha with the option `--forbid-only` in the continuous integration test command (or in a git precommit hook).

INCLUSIVE TESTS

This feature is the inverse of `.only()`. By appending `.skip()`, you may tell Mocha to simply ignore these suite(s) and test case(s). Anything skipped will be marked as pending, and reported as such. Here's an example of skipping an entire suite:

```
describe('Array', function() {
  describe.skip('#indexOf()', function() {
    // ...
  });
});
```

Or a specific test-case:

```
describe('Array', function() {
```

```
  describe('#indexOf()', function() {
    it.skip('should return -1 unless present', function
      // this test will not be run
    });

    it('should return the index when present', function
      // this test will be run
    });
  });
});
```

> *Best practice*: Use `.skip()` instead of commenting tests out.

You may also skip *at runtime* using `this.skip()`. If a test needs an environment or configuration which cannot be detected beforehand, a runtime skip is appropriate. For example:

```
it('should only test in the correct environment', funct
  if (/* check test environment */) {
    // make assertions
  } else {
    this.skip();
  }
});
```

The above test will be reported as pending. It's also important to note that calling `this.skip()` will effectively *abort* the test.

> *Best practice*: To avoid confusion, do not execute further instructions in a test or hook after calling `this.skip()`.

Contrast the above test with the following code:

```
it('should only test in the correct environment', funct
  if (/* check test environment */) {
    // make assertions
  } else {
    // do nothing
  }
});
```

Because this test *does nothing*, it will be reported as *passing*.

> *Best practice*: Don't do nothing! A test should make an assertion or use `this.skip()`.

To skip *multiple* tests in this manner, use `this.skip()` in a "before" hook:

```
before(function() {
  if (/* check test environment */) {
    // setup code
  } else {
    this.skip();
  }
});
```

> Before Mocha v3.0.0, `this.skip()` was not supported in asynchronous tests and hooks.

RETRY TESTS

You can choose to retry failed tests up to a certain number of times. This feature is designed to handle end-to-end tests (functional tests/Selenium…) where resources cannot be easily mocked/stubbed. **It's not recommended to use this feature for unit tests.**

This feature does re-run `beforeEach/afterEach` hooks but not `before/after` hooks.

NOTE: Example below was written using Selenium webdriver (which overwrites global Mocha hooks for `Promise` chain).

```
describe('retries', function() {
  // Retry all tests in this suite up to 4 times
  this.retries(4);

  beforeEach(function () {
    browser.get('http://www.yahoo.com');
  });

  it('should succeed on the 3rd try', function () {
    // Specify this test to only retry up to 2 times
    this.retries(2);
    expect($('.foo').isDisplayed()).to.eventually.be.tr
  });
});
```

DYNAMICALLY GENERATING TESTS

Given Mocha's use of `Function.prototype.call` and function expressions to define suites and test cases, it's straightforward to generate your tests dynamically. No special syntax is required — plain ol' JavaScript can be used to achieve functionality similar to "parameterized" tests, which you may have seen in other frameworks.

Take the following example:

```
var assert = require('chai').assert;

function add() {
  return Array.prototype.slice.call(arguments).reduce(f
    return prev + curr;
  }, 0);
}

describe('add()', function() {
  var tests = [
    {args: [1, 2],       expected: 3},
    {args: [1, 2, 3],    expected: 6},
    {args: [1, 2, 3, 4], expected: 10}
  ];

  tests.forEach(function(test) {
    it('correctly adds ' + test.args.length + ' args',
      var res = add.apply(null, test.args);
      assert.equal(res, test.expected);
```

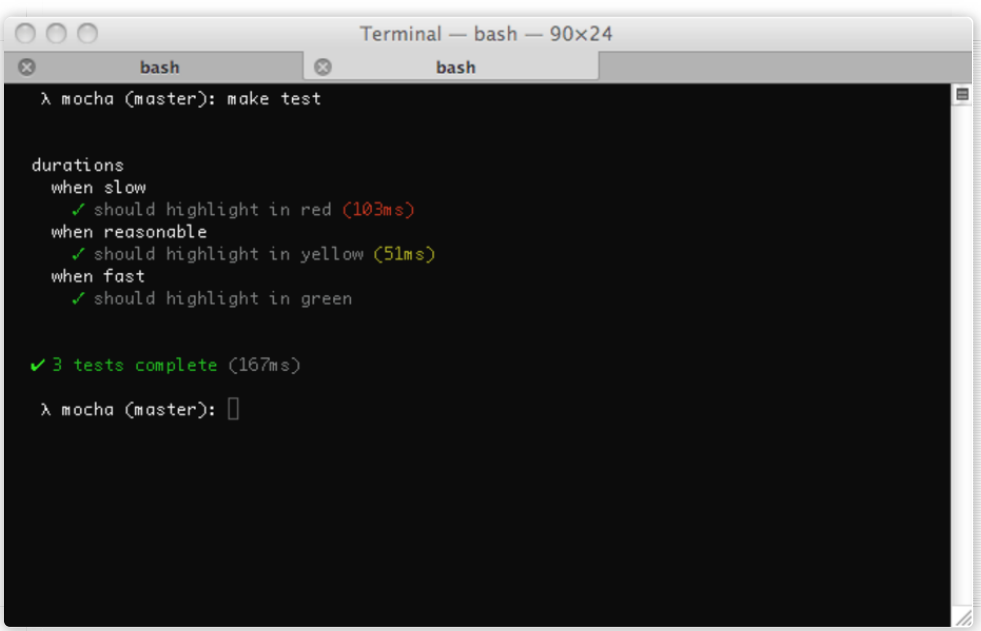
```
    });
  });
});
```

The above code will produce a suite with three specs:

```
$ mocha


  add()
    ✓ correctly adds 2 args
    ✓ correctly adds 3 args
    ✓ correctly adds 4 args
```

TEST DURATION

Many reporters will display test duration, as well as flagging tests that are slow, as shown here with the “spec” reporter:



To tweak what’s considered “slow”, you can use the `slow()` method:

```
describe('something slow', function() {
  this.slow(10000);

  it('should take long enough for me to go make a sandw
    // ...
  });
});
```

TIMEOUTS

SUITE-LEVEL

Suite-level timeouts may be applied to entire test "suites", or disabled via `this.timeout(0)`. This will be inherited by all nested suites and test-cases that do not override the value.

```
describe('a suite of tests', function() {
  this.timeout(500);

  it('should take less than 500ms', function(done){
    setTimeout(done, 300);
  });

  it('should take less than 500ms as well', function(do
    setTimeout(done, 250);
  });
})
```

TEST-LEVEL

Test-specific timeouts may also be applied, or the use of `this.timeout(0)` to disable timeouts all together:

```
it('should take less than 500ms', function(done){
  this.timeout(500);
  setTimeout(done, 300);
});
```

HOOK-LEVEL

Hook-level timeouts may also be applied:

```
describe('a suite of tests', function() {
  beforeEach(function(done) {
    this.timeout(3000); // A very long environment setu
    setTimeout(done, 2500);
  });
});
```

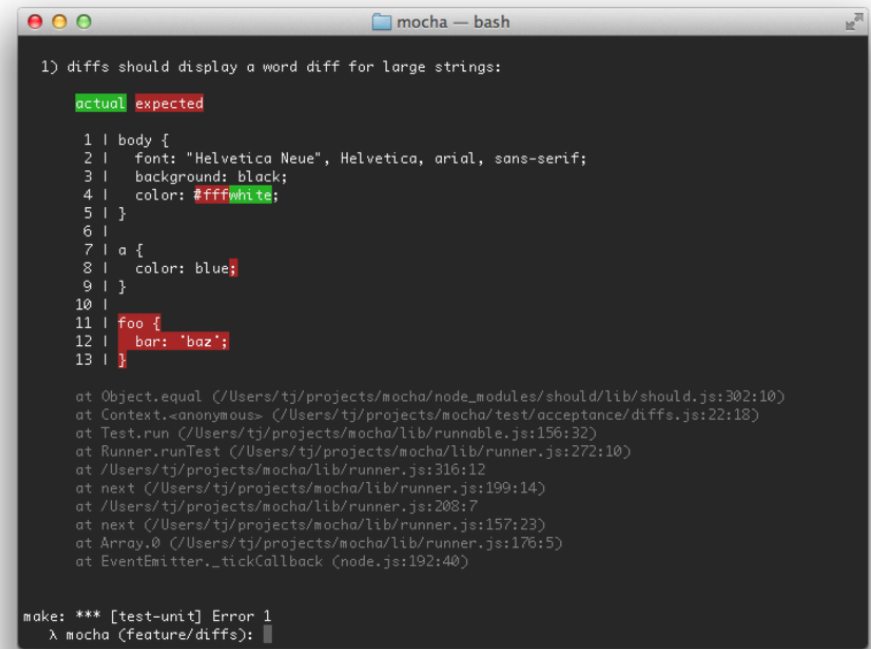
Again, use `this.timeout(0)` to disable the timeout for a hook.

> In v3.0.0 or newer, a parameter passed to `this.timeout()` greater than the maximum delay value will cause the timeout to be disabled.

DIFFS

Mocha supports the `err.expected` and `err.actual` properties of any thrown `AssertionError`s from an assertion library. Mocha will

attempt to display the difference between what was expected, and what the assertion actually saw. Here's an example of a "string" diff:



USAGE

```
Usage: mocha [debug] [options] [files]

Options:

  -V, --version                          output the
  -A, --async-only                       force all t
  -c, --colors                           force enabl
  -C, --no-colors                        force disab
  -G, --growl                            enable grow
  -O, --reporter-options <k=v,k2=v2,...> reporter-sp
  -R, --reporter <name>                  specify the
  -S, --sort                             sort test f
```

```
    -b, --bail                                bail after
    -d, --debug                               enable node
    -g, --grep <pattern>                      only run te
    -f, --fgrep <string>                      only run te
    -gc, --expose-gc                          expose gc e
    -i, --invert                              inverts --g
    -r, --require <name>                      require the
    -s, --slow <ms>                           "slow" test
    -t, --timeout <ms>                        set test-ca
    -u, --ui <name>                           specify use
    -w, --watch                               watch files
    --check-leaks                             check for g
    --full-trace                              display the
    --compilers <ext>:<module>,...            use the giv
    --debug-brk                               enable node
    --globals <names>                         allow the g
    --es_staging                              enable all
    --harmony<_classes,_generators,...>       all node --
    --preserve-symlinks                       Instructs t
    --icu-data-dir                            include ICU
    --inline-diffs                            display act
    --no-diff                                 do not show
    --inspect                                 activate de
    --inspect-brk                             activate de
    --interfaces                              display ava
    --no-deprecation                          silence dep
    --exit                                    force shutd
    --no-timeouts                             disables ti
    --no-warnings                             silence all
    --opts <path>                             specify opt
    --perf-basic-prof                         enable perf
    --napi-modules                            enable expe
    --prof                                    log statist
    --log-timer-events                        Time events
    --recursive                               include sub
```

```
    --reporters                              display ava
    --retries <times>                        set numbers
    --throw-deprecation                      throw an ex
    --trace                                  trace funct
    --trace-deprecation                      show stack
    --trace-warnings                         show stack
    --use_strict                             enforce str
    --watch-extensions <ext>,...             specify ext
    --delay                                  wait for as
    --allow-uncaught                         enable unca
    --forbid-only                            causes test
    --forbid-pending                         causes pend
    --file <file>                            include a f
    --exclude <file>                         a file or g
    -h, --help                               output usag

  Commands:

    init <path>  initialize a client-side mocha setup a
```

`-w, --watch`

Executes tests on changes to JavaScript in the CWD, and once initially.

`--exit` / `--no-exit`

> *Updated in Mocha v4.0.0*

Prior to **version v4.0.0,** *by default*, **Mocha would force its own process to exit once it was finished executing all tests. This behavior enables a set of potential problems; it's indicative of tests (or fixtures, harnesses, code under test, etc.) which don't clean up**

after themselves properly. Ultimately, “dirty” tests can (but not always) lead to *false positive* or *false negative* results.

“Hanging” most often manifests itself if a server is still listening on a port, or a socket is still open, etc. It can also be something like a runaway `setInterval()`, or even an errant `Promise` that never fulfilled.

The *default behavior* in v4.0.0 is `--no-exit`, where previously it was `--exit`.

The easiest way to “fix” the issue is to simply pass `--exit` to the Mocha process. It *can* be time-consuming to debug–because it’s not always obvious where the problem is–but it *is* recommended to do so.

To ensure your tests aren’t leaving messes around, here are some ideas to get started:

See the Node.js guide to debugging	Try something like why-is-node-running
Use the new `async_hooks` API (example)	Use `.only` until you find the test that causes Mocha to hang

`--compilers`

> *Updated in Mocha v4.0.0*

`--compilers` **is deprecated as of Mocha v4.0.0. See further explanation and workarounds.**

CoffeeScript is no longer supported out of the box. CS and similar transpilers may be used by mapping the file extensions (for use with `--watch`) and the module name. For example `--compilers coffee:coffee-script` with CoffeeScript 1.6- or `--compilers`

`coffee:coffee-script/register` with CoffeeScript 1.7+.

About Babel

If your ES6 modules have extension `.js`, you can `npm install --save-dev babel-register` and use `mocha --require babel-register`; `--compilers` is only necessary if you need to specify a file extension.

`-b, --bail`

Only interested in the first exception? use `--bail`!

`-d, --debug`

Enables node's debugger support, this executes your script(s) with `node debug <file ...>` allowing you to step through code and break with the `debugger` statement. Note the difference between `mocha debug` and `mocha --debug`: `mocha debug` will fire up node's built-in debug client, `mocha --debug` will allow you to use a different interface — such as the Blink Developer Tools. Implies `--no-timeouts`.

`--globals <names>`

Accepts a comma-delimited list of accepted global variable names. For example, suppose your app deliberately exposes a global named `app` and `YUI`, you may want to add `--globals app,YUI`. It also accepts wildcards. You could do `--globals '*bar'` and it would match `foobar`, `barbar`, etc. You can also simply pass in `'*'` to ignore all globals.

By using this option in conjunction with `--check-leaks`, you can specify a whitelist of known global variables that you would expect to leak into global scope.

`--check-leaks`

Use this option to have Mocha check for global variables that are leaked while running tests. Specify globals that are acceptable via the `--globals` option (for example: `--check-leaks --globals jQuery,MyLib`).

`-r, --require <module-name>`

The `--require` option is useful for libraries such as should.js, so you may simply `--require should` instead of manually invoking `require('should')` within each test file. Note that this works well for `should` as it augments `Object.prototype`, however if you wish to access a module's exports you will have to require them, for example `var should = require('should')`. Furthermore, it can be used with relative paths, e.g. `--require ./test/helper.js`

`-u, --ui <name>`

The `--ui` option lets you specify the interface to use, defaulting to "bdd".

`-R, --reporter <name>`

The `--reporter` option allows you to specify the reporter that will be used, defaulting to "spec". This flag may also be used to utilize third-party reporters. For example if you `npm install mocha-lcov-reporter` you may then do `--reporter mocha-lcov-reporter`.

`-t, --timeout <ms>`

Specifies the test-case timeout, defaulting to 2 seconds. To override you may pass the timeout in milliseconds, or a value with the `s` suffix, ex: `--timeout 2s` or `--timeout 2000` would be equivalent.

`--no-timeouts`

Disables timeouts. Equivalent to `--timeout 0`.

`-s, --slow <ms>`

Specify the “slow” test threshold, defaulting to 75ms. Mocha uses this to highlight test-cases that are taking too long.

`--file <file>`

Add a file you want included first in a test suite. This is useful if you have some generic setup code that must be included within the test suite. The file passed is not affected by any other flags (`--recursive` or `--sort` have no effect). Accepts multiple `--file` flags to include multiple files, the order in which the flags are given are the order in which the files are included in the test suite. Can also be used in `mocha.opts`.

`-g, --grep <pattern>`

The `--grep` option when specified will trigger mocha to only run tests matching the given `pattern` which is internally compiled to a `RegExp`.

Suppose, for example, you have “api” related tests, as well as “app” related tests, as shown in the following snippet; One could use `--grep api` or `--grep app` to run one or the other. The same goes for any other part of a suite or test-case title, `--grep users` would be valid as well, or even `--grep GET`.

```
describe('api', function() {
  describe('GET /api/users', function() {
    it('respond with an array of users', function() {
```

```
      // ...
    });
  });
});

describe('app', function() {
  describe('GET /users', function() {
    it('respond with an array of users', function() {
      // ...
    });
  });
});
```

INTERFACES

Mocha's "interface" system allows developers to choose their style of DSL. Mocha has **BDD**, **TDD**, **Exports**, **QUnit** and **Require**-style interfaces.

BDD

The **BDD** interface provides `describe()`, `context()`, `it()`, `specify()`, `before()`, `after()`, `beforeEach()`, and `afterEach()`.

`context()` is just an alias for `describe()`, and behaves the same way; it just provides a way to keep tests easier to read and organized. Similarly, `specify()` is an alias for `it()`.

> All of the previous examples were written using the **BDD** interface.

```
describe('Array', function() {
  before(function() {
    // ...
  });

  describe('#indexOf()', function() {
    context('when not present', function() {
      it('should not throw an error', function() {
        (function() {
          [1,2,3].indexOf(4);
        }).should.not.throw();
      });
      it('should return -1', function() {
        [1,2,3].indexOf(4).should.equal(-1);
      });
    });
    context('when present', function() {
      it('should return the index where the element f
        [1,2,3].indexOf(3).should.equal(2);
      });
    });
  });
});
```

TDD

The TDD interface provides `suite()`, `test()`, `suiteSetup()`, `suiteTeardown()`, `setup()`, **and** `teardown()`:

```
suite('Array', function() {
  setup(function() {
    // ...
  });
```

```
  suite('#indexOf()', function() {
    test('should return -1 when not present', function(
      assert.equal(-1, [1,2,3].indexOf(4));
    });
  });
});
```

EXPORTS

The **Exports** interface is much like Mocha's predecessor expresso. The keys `before`, `after`, `beforeEach`, and `afterEach` are special-cased, object values are suites, and function values are test-cases:

```
module.exports = {
  before: function() {
    // ...
  },

  'Array': {
    '#indexOf()': {
      'should return -1 when not present': function() {
        [1,2,3].indexOf(4).should.equal(-1);
      }
    }
  }
};
```

QUNIT

The QUnit-inspired interface matches the "flat" look of QUnit, where the test suite title is simply defined before the test-cases. Like TDD, it uses `suite()` and `test()`, but resembling BDD, it also contains

`before()`, `after()`, `beforeEach()`, and `afterEach()`.

```
function ok(expr, msg) {
  if (!expr) throw new Error(msg);
}

suite('Array');

test('#length', function() {
  var arr = [1,2,3];
  ok(arr.length == 3);
});

test('#indexOf()', function() {
  var arr = [1,2,3];
  ok(arr.indexOf(1) == 0);
  ok(arr.indexOf(2) == 1);
  ok(arr.indexOf(3) == 2);
});

suite('String');

test('#length', function() {
  ok('foo'.length == 3);
});
```

REQUIRE

The `require` interface allows you to require the `describe` and friend words directly using `require` and call them whatever you want. This interface is also useful if you want to avoid global variables in your tests.

Note: The `require` interface cannot be run via the `node` executable, and must be run via `mocha`.

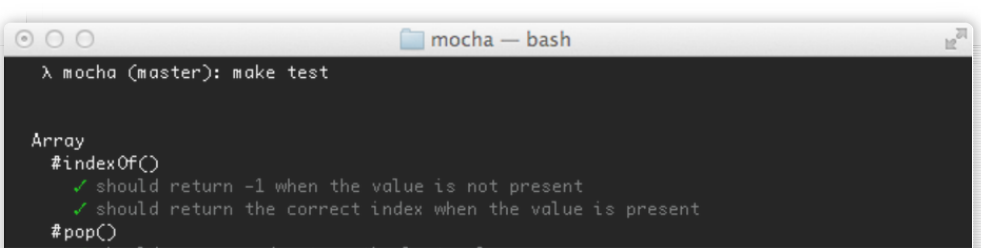
```
var testCase = require('mocha').describe;
var pre = require('mocha').before;
var assertions = require('mocha').it;
var assert = require('chai').assert;

testCase('Array', function() {
  pre(function() {
    // ...
  });

  testCase('#indexOf()', function() {
    assertions('should return -1 when not present', fun
      assert.equal([1,2,3].indexOf(4), -1);
    });
  });
});
```

REPORTERS

Mocha reporters adjust to the terminal window, and always disable ANSI-escape coloring when the stdio streams are not associated with a TTY.

SPEC

This is the default reporter. The “spec” reporter outputs a hierarchical view nested just as the test cases are.



```
    ✓ should remove and return the last value


  ✔ 3 tests completed (5ms)

   λ mocha (master):
```

```
mocha — bash
   λ mocha (master): make test


  Array
    #indexOf()
      0) should return -1 when the value is not present
      ✓ should return the correct index when the value is present
    #pop()
      ✓ should remove and return the last value


  ✖ 1 of 3 tests failed:

  0) Array #indexOf() should return -1 when the value is not present: AssertionError: expe
cted -1 to equal 1

  at Object.equal (/Users/tj/projects/mocha/node_modules/should/lib/should.js:306:10)
      at Test.fn (/Users/tj/projects/mocha/test/interfaces/bdd.js:6:33)
      at Test.run (/Users/tj/projects/mocha/lib/test.js:80:29)
      at Array.0 (/Users/tj/projects/mocha/lib/runner.js:187:12)
      at EventEmitter._tickCallback (node.js:192:40)


make: *** [test-unit] Error 1
```

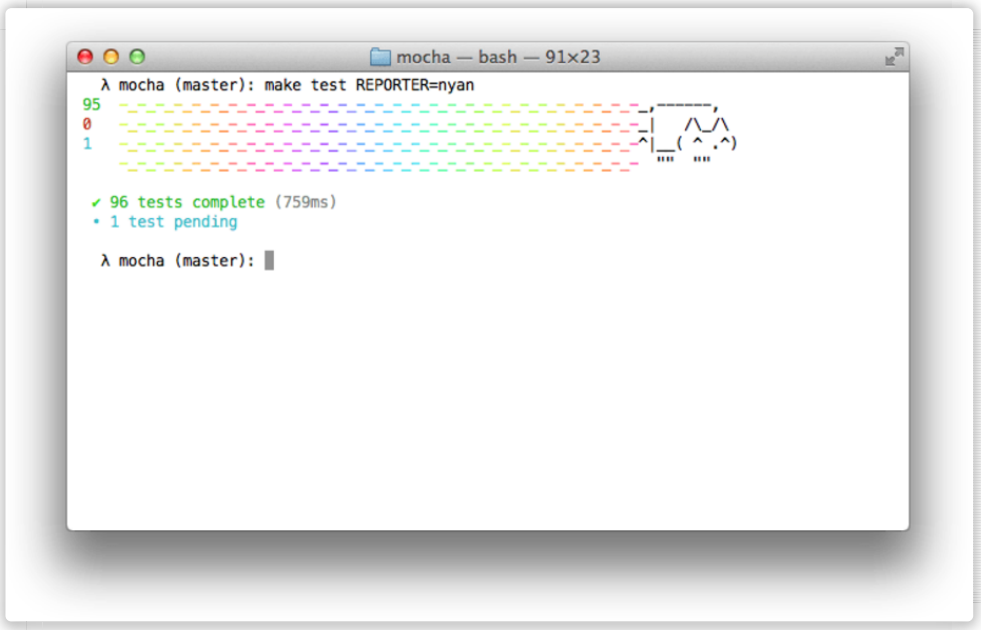
DOT MATRIX

The dot matrix (or “dot”) reporter is simply a series of characters which represent test cases. Failures highlight in red exclamation marks (`!`), pending tests with a blue comma (`,`), and slow tests as yellow. Good if you prefer minimal output.

```
$ mocha test/ --reporter dot


  ..........................................................,...............................
  ..........................................................!...............................
  ..

  176 passing (549ms)
  1 pending
  1 failing
```

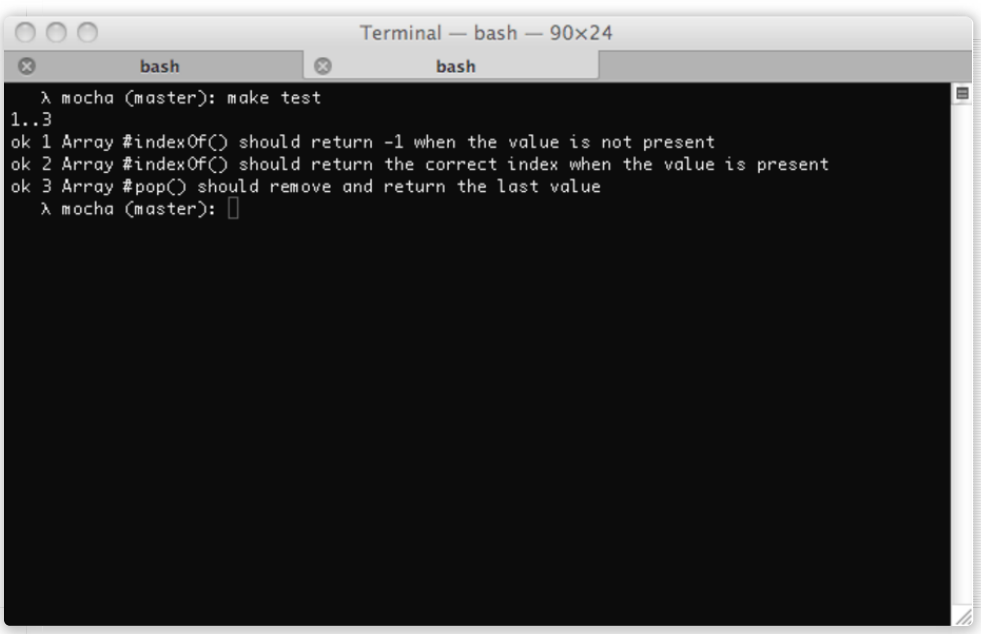
NYAN

The “nyan” reporter is exactly what you might expect:



TAP

The TAP reporter emits lines for a Test-Anything-Protocol consumer.



LANDING STRIP

The Landing Strip (`landing`) reporter is a gimmicky test reporter

simulating a plane landing :) unicode ftw

```
mocha — bash
λ mocha (master): make test

------------------------------------------------------------------
.................................................................✈
------------------------------------------------------------------

✔ 3 tests completed (3ms)

λ mocha (master):
```

```
mocha — bash
λ mocha (master): make test

------------------------------------------------------------------
......................✈.........................................
------------------------------------------------------------------

✘ 1 of 3 tests failed:

0) Array #indexOf() should return -1 when the value is not present: AssertionError: expe
cted -1 to equal 1

at Object.equal (/Users/tj/projects/mocha/node_modules/should/lib/should.js:306:10)
    at Test.fn (/Users/tj/projects/mocha/test/interfaces/bdd.js:6:33)
    at Test.run (/Users/tj/projects/mocha/lib/test.js:80:29)
    at Array.0 (/Users/tj/projects/mocha/lib/runner.js:187:12)
    at EventEmitter._tickCallback (node.js:192:40)


make: *** [test-unit] Error 1
λ mocha (master):
```

LIST

The “list” reporter outputs a simple specifications list as test cases pass or fail, outputting the failure details at the bottom of the output.

```
mocha — bash
λ mocha (master): make test

✓ Array #indexOf() should return -1 when the value is not present: 1ms
✓ Array #indexOf() should return the correct index when the value is present: 0ms
✓ Array #pop() should remove and return the last value: 1ms

✔ 3 tests completed (4ms)
```

```
λ mocha (master):
```

PROGRESS

The “progress” reporter implements a simple progress-bar:

```
Terminal — bash — 90×24
bash    bash
λ mocha (master): make test

[————————————————————————————————————]

✔ 32 tests complete (193ms)

λ mocha (master):
```

JSON

The “JSON” reporter outputs a single large JSON object when the tests have completed (failures or not).

```
mocha — bash
λ mocha (master): make test
{"stats":{"suites":4,"tests":3,"passes":3,"failures":0,"start":"2011-11-19T20:27:09.040Z",
"end":"2011-11-19T20:27:09.042Z","duration":2},"tests":[{"title":"should return -1 when th
e value is not present","fullTitle":"Array #indexOf() should return -1 when the value is n
ot present","duration":1},{"title":"should return the correct index when the value is pres
ent","fullTitle":"Array #indexOf() should return the correct index when the value is prese
nt","duration":0},{"title":"should remove and return the last value","fullTitle":"Array #p
op() should remove and return the last value","duration":1}],"failures":[],"passes":[{"tit
le":"should return -1 when the value is not present","fullTitle":"Array #indexOf() should
return -1 when the value is not present","duration":1},{"title":"should return the correct
 index when the value is present","fullTitle":"Array #indexOf() should return the correct
index when the value is present","duration":0},{"title":"should remove and return the last
```

```
value","fullTitle":"Array #pop() should remove and return the last value","duration":1}]}
  λ mocha (master):
```

JSON STREAM

The “JSON stream” reporter outputs newline-delimited JSON “events” as they occur, beginning with a “start” event, followed by test passes or failures, and then the final “end” event.

```
mocha — bash
  λ mocha (master): make test
["start",{"total":3}]
["pass",{"title":"should return -1 when the value is not present","fullTitle":"Array #inde
xOf() should return -1 when the value is not present","duration":0}]
["pass",{"title":"should return the correct index when the value is present","fullTitle":"
Array #indexOf() should return the correct index when the value is present","duration":0}]
["pass",{"title":"should remove and return the last value","fullTitle":"Array #pop() shoul
d remove and return the last value","duration":1}]
["end",{"suites":4,"tests":3,"passes":3,"failures":0,"start":"2011-11-19T20:27:27.538Z","e
nd":"2011-11-19T20:27:27.542Z","duration":4}]   λ mocha (master):
  λ mocha (master):
```

MIN

The “min” reporter displays the summary only, while still outputting errors on failure. This reporter works great with `--watch` as it clears the terminal in order to keep your test summary at the top.

```
mocha — node — 90×22
bash | bash | node

✔ 62 tests complete (63ms)

watching
```

DOC

The “doc” reporter outputs a hierarchical HTML body representation of your tests. Wrap it with a header, footer, and some styling, then you have some fantastic documentation!

```
mocha — bash
  λ mocha (master): make test
   <section class="suite">
     <h1>Array</h1>
     <dl>
       <section class="suite">
         <h1>#indexOf()</h1>
         <dl>
           <dt>should return -1 when the value is not present</dt>
           <dd><pre><code>
[1,2,3].indexOf(5).should.equal(-1);
[1,2,3].indexOf(0).should.equal(-1);</code></pre></dd>
           <dt>should return the correct index when the value is present</dt>
           <dd><pre><code>
[1,2,3].indexOf(1).should.equal(0);
[1,2,3].indexOf(2).should.equal(1);
[1,2,3].indexOf(3).should.equal(2);</code></pre></dd>
         </dl>
       </section>
       <section class="suite">
         <h1>#pop()</h1>
         <dl>
           <dt>should remove and return the last value</dt>
           <dd><pre><code>
var arr = [1,2,3];
```

For example, suppose you have the following JavaScript:

```
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present'
      [1,2,3].indexOf(5).should.equal(-1);
      [1,2,3].indexOf(0).should.equal(-1);
    });
  });
```

```
});
```

The command `mocha --reporter doc array` would yield:

```
<section class="suite">
  <h1>Array</h1>
  <dl>
    <section class="suite">
      <h1>#indexOf()</h1>
      <dl>
      <dt>should return -1 when the value is not presen
      <dd><pre><code>[1,2,3].indexOf(5).should.equal(-1
[1,2,3].indexOf(0).should.equal(-1);</code></pre></dd>
      </dl>
    </section>
  </dl>
</section>
```

The SuperAgent request library test documentation was generated with Mocha's doc reporter using this Bash command:

```
$ mocha --reporter=doc | cat docs/head.html - docs/tail
```

View SuperAgent's Makefile for reference.

MARKDOWN

The "markdown" reporter generates a markdown TOC and body for your test suite. This is great if you want to use the tests as documentation within a Github wiki page, or a markdown file in the repository that Github can render. For example here is the Connect test output.

XUNIT

The `xunit` reporter is also available. It outputs an XUnit-compatible XML document, often applicable in CI servers.

By default, it will output to the console. To write directly to a file, use `--reporter-options output=filename.xml`.

THIRD-PARTY REPORTERS

Mocha allows you to define custom reporters. For more information see the wiki. An example is the TeamCity reporter.

HTML REPORTER

The HTML reporter is not intended for use on the command-line.

RUNNING MOCHA IN THE BROWSER

Mocha runs in the browser. Every release of Mocha will have new builds of `./mocha.js` and `./mocha.css` for use in the browser.

BROWSER-SPECIFIC METHODS

The following method(s) *only* function in a browser context:

`mocha.allowUncaught()` : If called, uncaught errors will not be absorbed by the error handler.

A typical setup might look something like the following, where we call `mocha.setup('bdd')` to use the BDD interface before loading the test scripts, running them `onload` with `mocha.run()`.

```
<html>
```

```
<head>
  <meta charset="utf-8">
  <title>Mocha Tests</title>
  <link href="https://unpkg.com/mocha@5.2.0/mocha.css"
</head>
<body>
  <div id="mocha"></div>

  <script src="https://unpkg.com/chai/chai.js"></script
  <script src="https://unpkg.com/mocha@5.2.0/mocha.js">

  <script>mocha.setup('bdd')</script>
  <script src="test.array.js"></script>
  <script src="test.object.js"></script>
  <script src="test.xhr.js"></script>
  <script>
    mocha.checkLeaks();
    mocha.run();
  </script>
</body>
</html>
```

GREP

The browser may use the `--grep` as functionality. Append a query-string to your URL: `?grep=api`.

BROWSER CONFIGURATION

Mocha options can be set via `mocha.setup()`. Examples:

```
// Use "tdd" interface.  This is a shortcut to setting
// any other options must be passed via an object.
```

```
mocha.setup('tdd');

// This is equivalent to the above.
mocha.setup({
  ui: 'tdd'
});

// Use "tdd" interface, ignore leaks, and force all tes
mocha.setup({
  ui: 'tdd',
  ignoreLeaks: true,
  asyncOnly: true
});
```

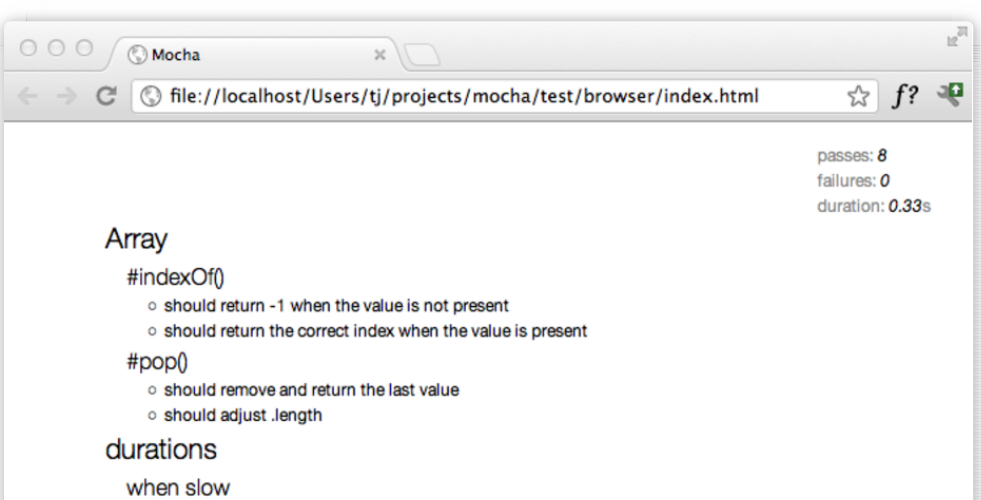
BROWSER-SPECIFIC OPTION(S)

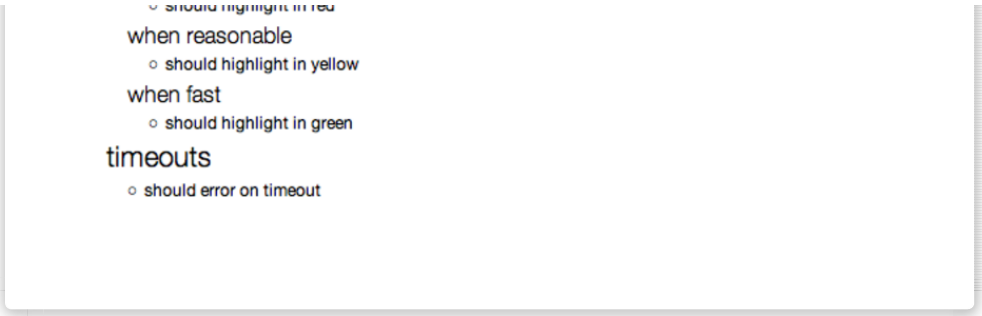
The following option(s) *only* function in a browser context:

`noHighlighting`: **If set to `true`, do not attempt to use syntax highlighting on output test code.**

REPORTING

The "HTML" reporter is what you see when running Mocha in the browser. It looks like this:





Mochawesome is a great alternative to the default HTML reporter.

MOCHA.OPTS

Back on the server, Mocha will attempt to load `"./test/mocha.opts"` as a Run-Control file of sorts.

Beginning-of-line comment support is available; any line *starting* with a hash (#) symbol will be considered a comment. Blank lines may also be used. Any other line will be treated as a command-line argument (along with any associated option value) to be used as a default setting. Settings should be specified one per line.

The lines in this file are prepended to any actual command-line arguments. As such, actual command-line arguments will take precedence over the defaults.

For example, suppose you have the following `mocha.opts` file:

```
# mocha.opts

  --require should
  --reporter dot
  --ui bdd
```

The settings above will default the reporter to `dot`, require the `should` library, and use `bdd` as the interface. With this, you may then

invoke `mocha` with additional arguments, here enabling Growl support, and changing the reporter to `list`:

```
$ mocha --reporter list --growl
```

THE `TEST/` DIRECTORY

By default, `mocha` looks for the glob `./test/*.js`, so you may want to put your tests in `test/` folder. If you want to include sub directories, pass the `--recursive` option.

To configure where `mocha` looks for tests, you may pass your own glob:

```
$ mocha --recursive "./spec/*.js"
```

Some shells support recursive matching by using the ** wildcard in a glob. Bash >= 4.3 supports this with the `globstar` option which must be enabled to get the same results as passing the `--recursive` option (ZSH and Fish support this by default). With recursive matching enabled, the following is the same as passing `--recursive`:

```
$ mocha "./spec/**/*.js"
```

Note: Double quotes around the glob are recommended for portability.

EDITOR PLUGINS

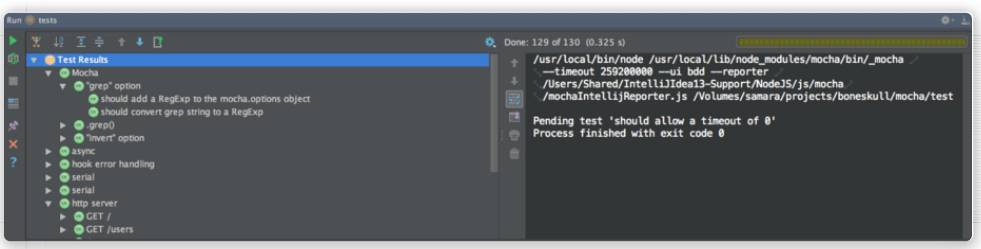
The following editor-related packages are available:

TEXTMATE

The Mocha TextMate bundle includes snippets to make writing tests quicker and more enjoyable.

JETBRAINS

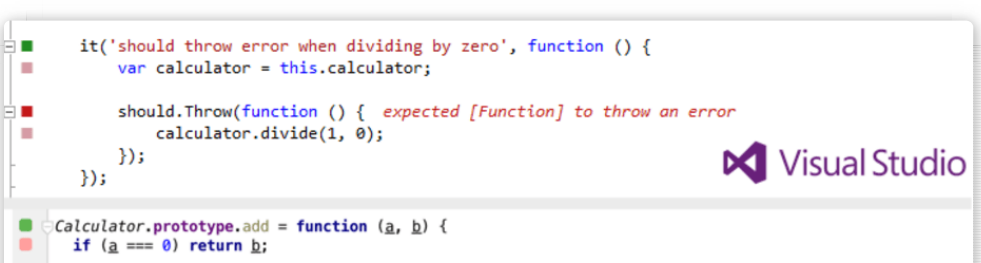
JetBrains provides a NodeJS plugin for its suite of IDEs (IntelliJ IDEA, WebStorm, etc.), which contains a Mocha test runner, among other things.



The plugin is titled **NodeJS**, and can be installed via **Preferences > Plugins**, assuming your license allows it.

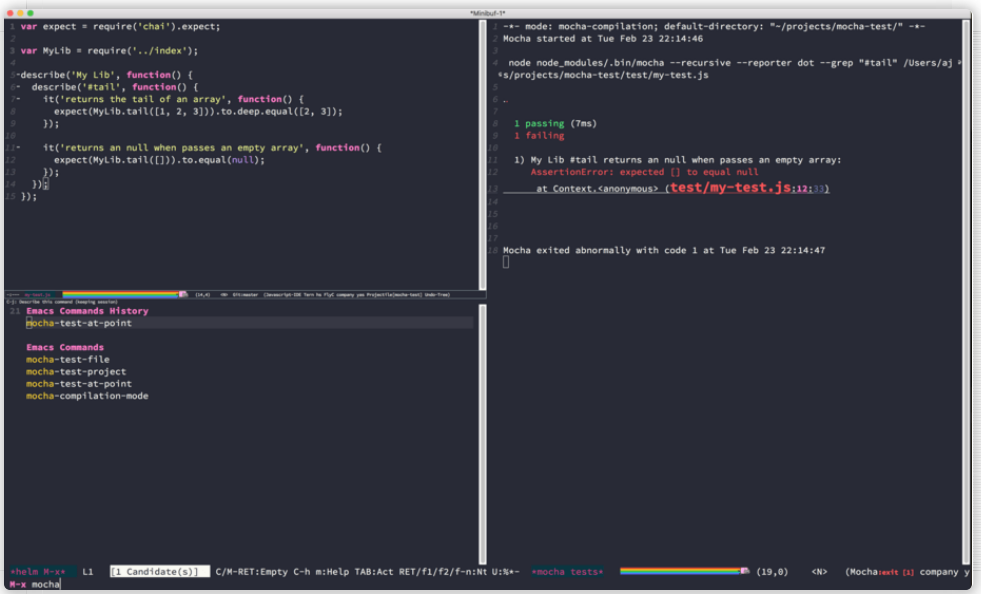
WALLABY.JS

Wallaby.js is a continuous testing tool that enables real-time code coverage for Mocha with any assertion library in VS Code, Atom, JetBrains IDEs (IntelliJ IDEA, WebStorm, etc.), Sublime Text and Visual Studio for both browser and node.js projects.



EMACS

Emacs support for running Mocha tests is available via a 3rd party package mocha.el. The package is available on MELPA, and can be installed via `M-x package-install mocha`.

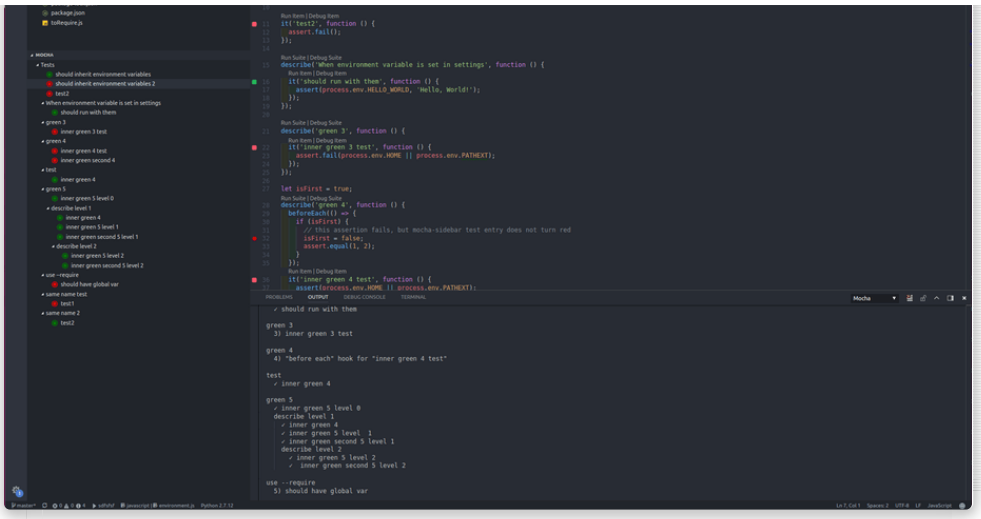


MOCHA SIDEBAR (VS CODE)

Mocha sidebar is the most complete mocha extension for vs code.

Features

- see all tests in VS Code sidebar menu
- run & debug tests for each level hierarchy from all tests to a single test (and each describe of course)
- auto run tests on file save
- see tests results directly in the code editor



EXAMPLES

Real live example code:

- Express
- Connect
- SuperAgent
- WebSocket.io
- Mocha

TESTING MOCHA

To run Mocha's tests, you will need GNU Make or compatible; Cygwin should work.

```
$ cd /path/to/mocha
$ npm install
$ npm test
```

To use a different reporter:

```
$ REPORTER=nyan npm test
```

MORE INFORMATION

In addition to chatting with us on Gitter, for additional information such as using spies, mocking, and shared behaviours be sure to check out the Mocha Wiki on GitHub. For discussions join the Google Group. For a running example of Mocha, view example/tests.html. For the JavaScript API, view the API documentation or the source.

Last updated: Sat Aug 11 06:56:36 2018