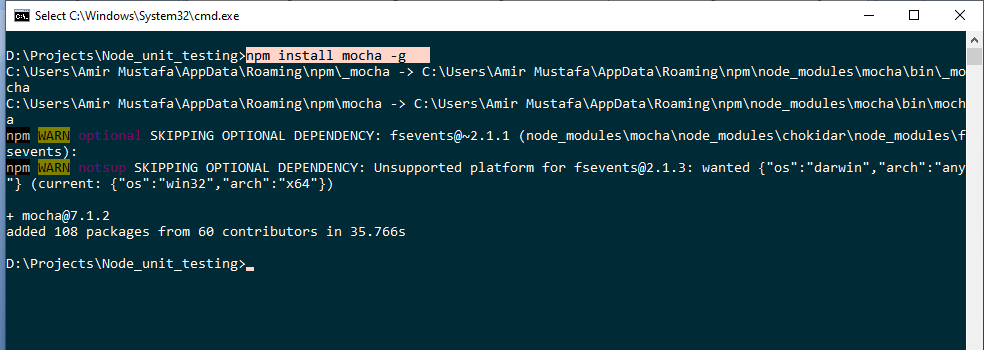
**Node JS – Unit Testing**

**Create basic Node js project:**

Npm init 🡪hit enter – enter 🡪 package.json file will be created

Install Mocha – JS Test framework

Npm install mocha -g



Inside node project created above: create directory project/test/test.js

test.js

describe("file to be tested", () => { // file to test

    context("function to be tested", () => { // function to test

        it("should do something", () =. { // each test have it block

        })

    })

});

* Node JS comes with default assertion function,but it is not that powerful to test real world application. It’s purpose was to do small application

Lets test that

Test.js

Path - Project/test/test.js1

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    it("should do something", () => {

      assert.equal(1, 2);

    });

  });

});

To run test from terminal – go to project/test and run mocha test



Test 2:

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    it("should do something", () => {

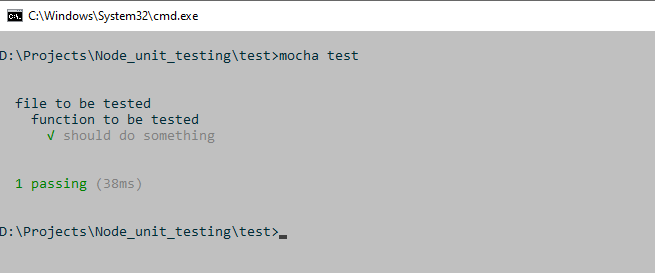
      assert.equal(1, 1);

    });

  });

});

Mocha test



Test 3 – Writing second test

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    it("should do something", () => {

      assert.equal(1, 1);

    });

    it("should do something else", () => {

      assert.deepEqual({ name: "amir" }, { name: "john" });

    });

  });

});

Mocha test



Down below it will show what is expected and what the actual value is (just like git)

Test 4 – making name same in second function

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    it("should do something", () => {

      assert.equal(1, 1);

    });

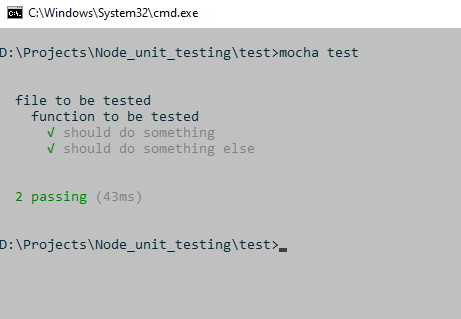
    it("should do something else", () => {

      assert.deepEqual({ name: "amir" }, { name: "amir" });

    });

  });

});



Test 5 – Pending test

Sometimes a function is not written, or not ready to be tested. So this is the **reminder from mocha to go back and finish it in the future.**

Test.js

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    it("should do something", () => {

      assert.equal(1, 1);

    });

    it("should do something else", () => {

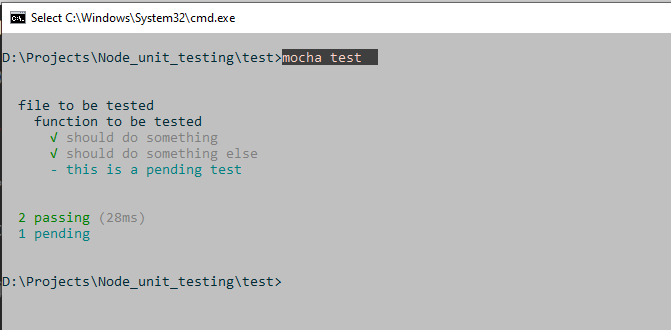
      assert.deepEqual({ name: "amir" }, { name: "amir" });

    });

    it("this is a pending test");

  });

});



Before and After ,Before Each and After Each

Before – this function runs before first test

After – this function runs after last test

NOTE – pending test is included in after and before case

Test.js

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    before(() => {

      console.log("=========== before function called============");

    });

    after(() => {

      console.log("=========== after function called ============");

    });

    it("should do something", () => {

      assert.equal(1, 1);

    });

    it("should do something else", () => {

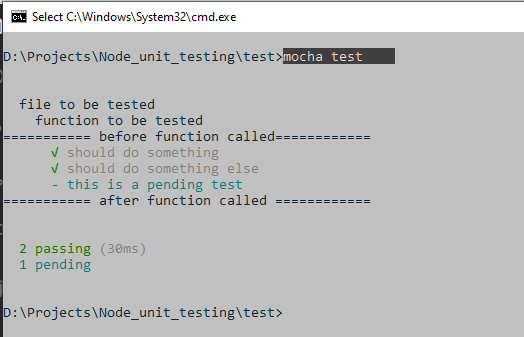
      assert.deepEqual({ name: "amir" }, { name: "amir" });

    });

    it("this is a pending test");

  });

});



beforeEach – this function runs before each test

afterEach – this function runs after each test

NOTE – pending test is left in afterEach and beforeEach case

Test.js

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    before(() => {

      console.log("=========== before function called============");

    });

    after(() => {

      console.log("=========== after function called ============");

    });

    beforeEach(() => {

      console.log("=============== beforeEach function called ==============");

    });

    afterEach(() => {

      console.log("=============== afterEach function called ==============");

    });

    it("should do something", () => {

      assert.equal(1, 1);

    });

    it("should do something else", () => {

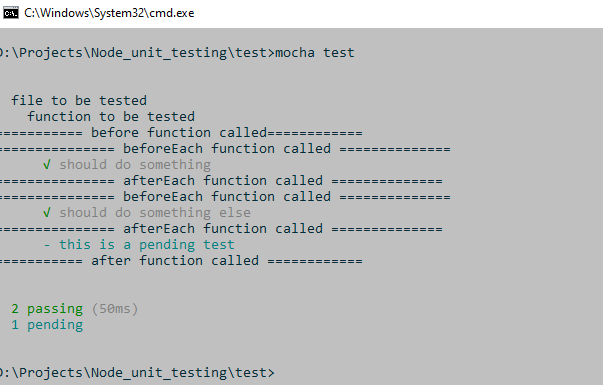
      assert.deepEqual({ name: "amir" }, { name: "amir" });

    });

    it("this is a pending test");

  });

});



Multiple Context:

* We can add multiple context in describe
* No. of context = No. of functions

A

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

it("should do something", () => {

      assert.equal(1, 1);

    });

    it("should do something else", () => {

      assert.deepEqual({ name: "amir" }, { name: "amir" });

    });

    it("this is a pending test");

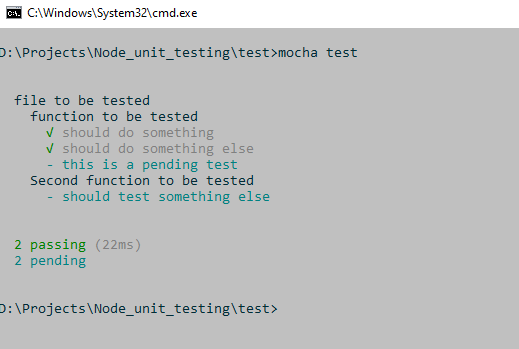
  });

  context("Second function to be tested", () => {

    it("should test something else");

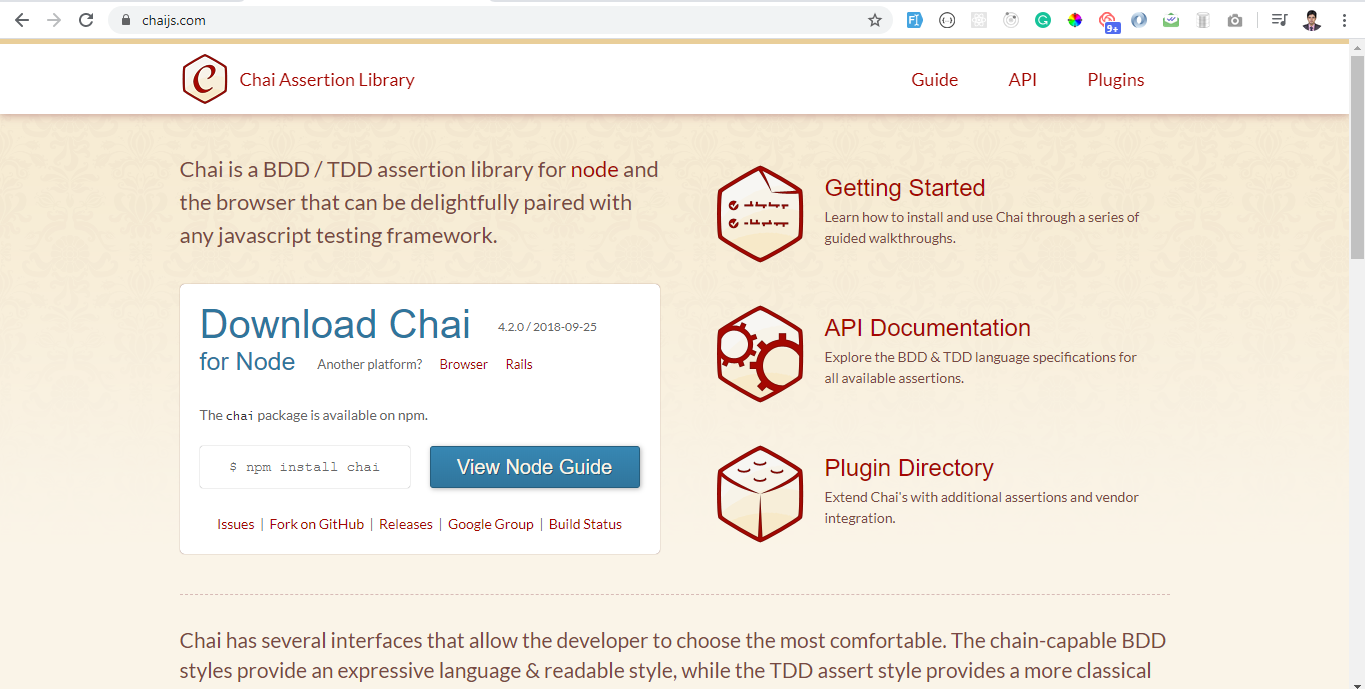
  });

});



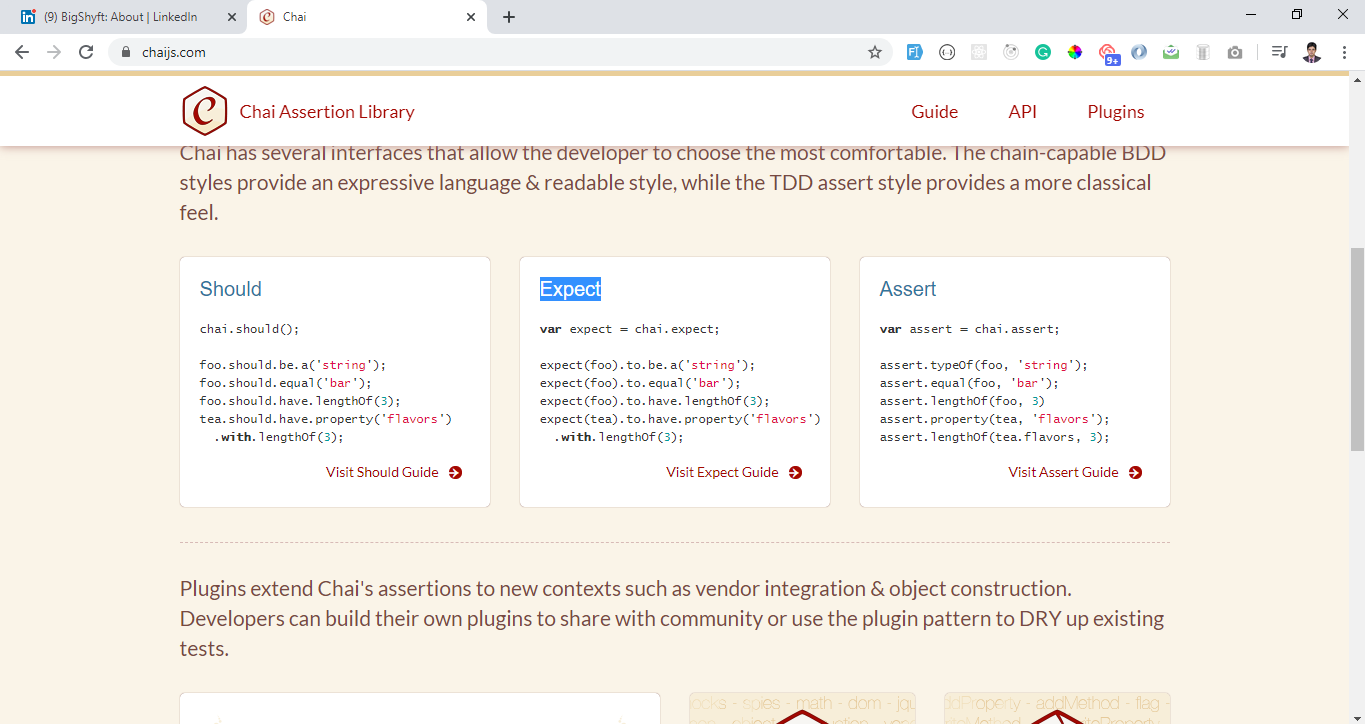
Chai.js

* Chai is the assertion library. Node js has the assertion library. It is very limited. Chai is the best standard in market for assertion

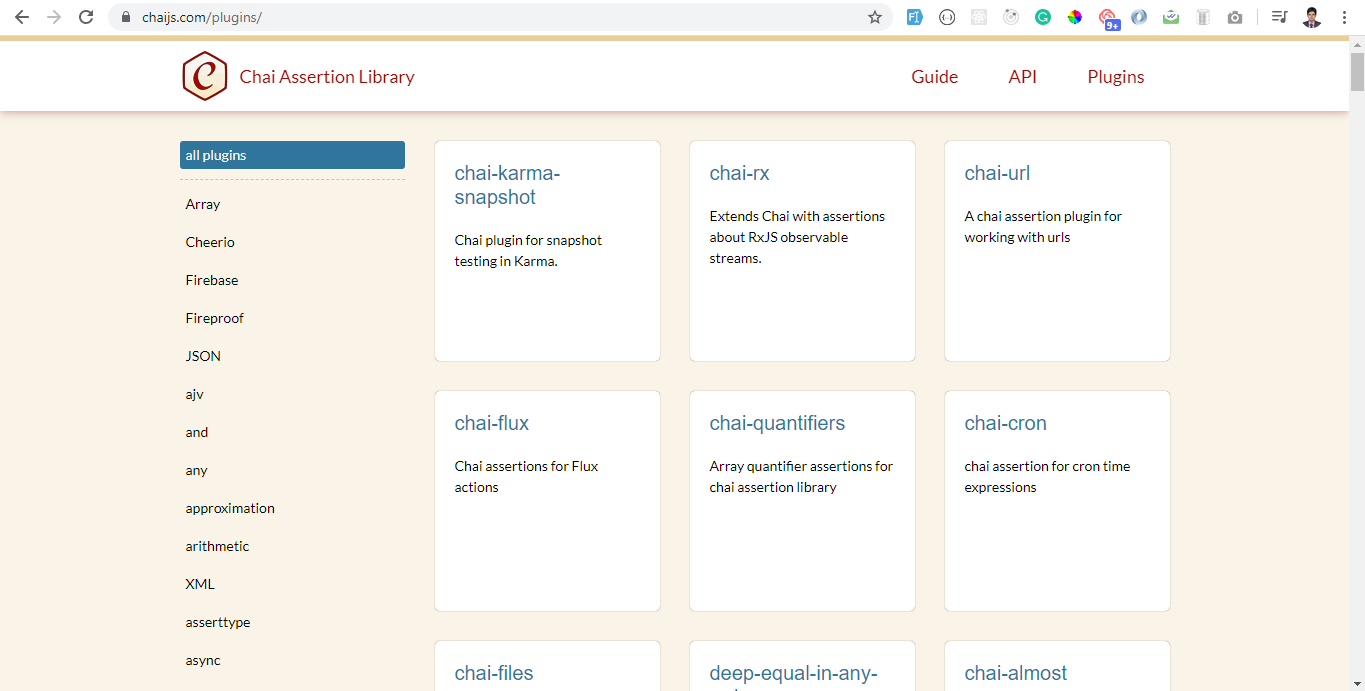


* Chai has 3 ways you can assert same code – should, expect and assert

Expect is the best method.

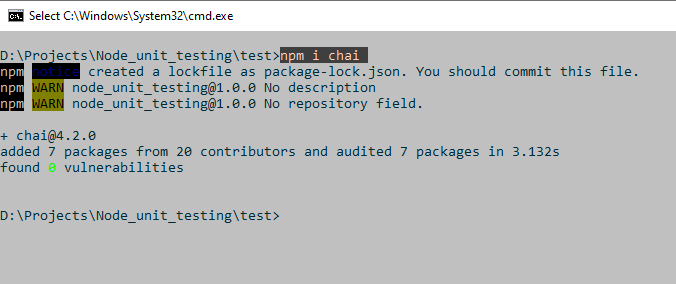


* Another important thing in chai is that it has many plugins which is helpful in many tests and make our life easier.



Chai Basics

Npm i chai



Eg1

chai-test.js

path – project/test/chai-test.js

const chai = require("chai");

const expect = chai.expect; // we use expect style of assertion

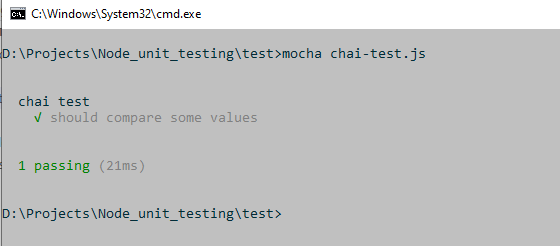
describe("chai test", () => {

  it("should compare some values", () => {

    expect(1).to.equal(1);

  });

});



Eg2:

const chai = require("chai");

const expect = chai.expect;

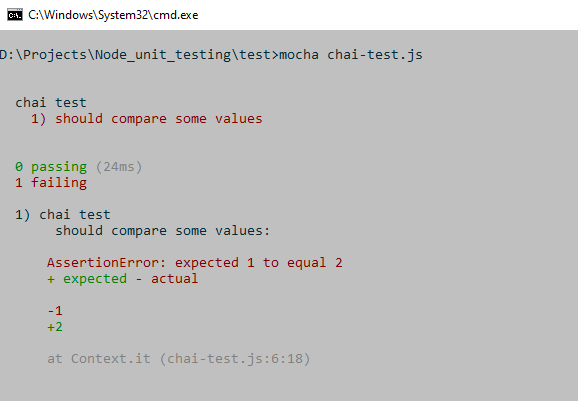
describe("chai test", () => {

  it("should compare some values", () => {

    expect(1).to.equal(2);

  });

});



Eg3:

const chai = require("chai");

const expect = chai.expect;

describe("chai test", () => {

  it("should compare some values", () => {

    expect(1).to.equal(1);

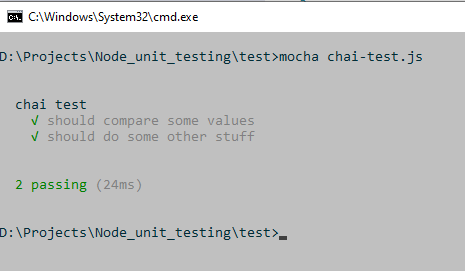
  });

  it("should do some other stuff", () => {

    expect({ name: "Amir" }).to.deep.equal({ name: "Amir" });

  });

});



Eg 4

const chai = require("chai");

const expect = chai.expect;

describe("chai test", () => {

  it("should compare some values", () => {

    expect(1).to.equal(1);

  });

  it("should do some other stuff", () => {

    expect({ name: "Amir" }).to.deep.equal({ name: "Amir" });

    expect({ name: "foo" }).to.have.property("name").to.equal("foo");

    expect(5 > 8).to.be.false;

    expect({}).to.be.a("object");

    expect("foo").to.be.a("string");

    expect(3).to.be.a("number");

    expect("bar").to.be.a("string").with.lengthOf(3);

    expect([1, 3, 5].length).to.equal(3);

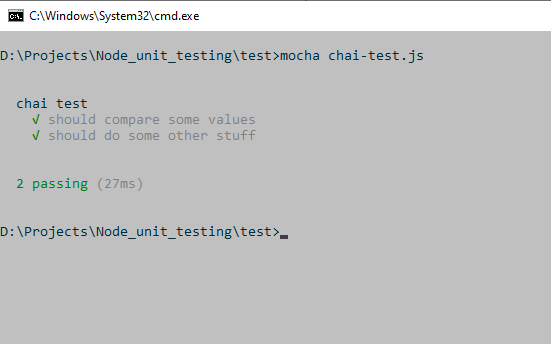
    expect(null).to.be.null;

    expect(undefined).to.not.exist;

    expect(1).to.exist;

  });

});



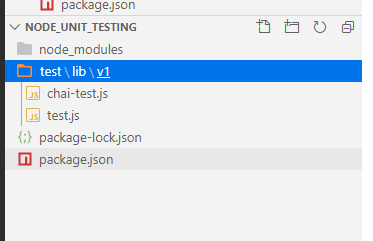
CLI and Test Commands

* So there are two common ways to setup directory structure for test files

Style 1

1. Inside test/lib/v1/<your\_files>

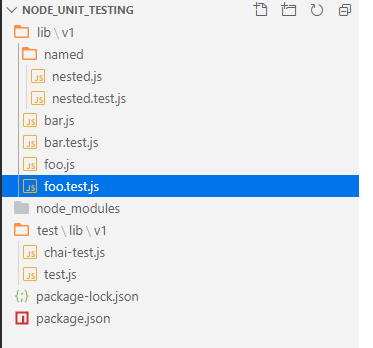
You can maintain versions of different tests



Style 2

1. Beside each page you create exactly same name with \_test.js

Eg. cart.js, cart\_test.js



Run all the tests in directory

mocha <folder\_to\_test> --recursive

**eg. mocha test –recursive**

* Now let’s create multiple test files

eg. project/test/lib/v1/foo.js

console.log("This should - NOT RUN");

project/test/lib/v1/foo.test.js

const chai = require("chai");

const expect = chai.expect;

describe("chai test", () => {

  it("should compare some values", () => {

    expect(1).to.equal(1);

  });

  it("should do some other stuff", () => {

    expect({ name: "Amir" }).to.deep.equal({ name: "Amir" });

    expect({ name: "foo" }).to.have.property("name").to.equal("foo");

    expect(5 > 8).to.be.false;

    expect({}).to.be.a("object");

    expect("foo").to.be.a("string");

    expect(3).to.be.a("number");

    expect("bar").to.be.a("string").with.lengthOf(3);

    expect([1, 3, 5].length).to.equal(3);

    expect(null).to.be.null;

    expect(undefined).to.not.exist;

    expect(1).to.exist;

  });

});

project/test/lib/v1/bar.js

console.log("Hello from bar.js");

project/test/lib/v1/bar.test.js

const assert = require("assert");

describe("file to be tested", () => {

  context("function to be tested", () => {

    // before(() => {

    //   console.log("=========== before function called============");

    // });

    // after(() => {

    //   console.log("=========== after function called ============");

    // });

    // beforeEach(() => {

    //   console.log("=============== beforeEach function called ==============");

    // });

    // afterEach(() => {

    //   console.log("=============== afterEach function called ==============");

    // });

    it("should do something", () => {

      assert.equal(1, 1);

    });

    it("should do something else", () => {

      assert.deepEqual({ name: "amir" }, { name: "amir" });

    });

    it("this is a pending test");

  });

  context("Second function to be tested", () => {

    it("should test something else");

  });

});

Project/test/lib/v1/names/nested.js

console.log("=========== NESTED - This should NOT RUN");

Project/test/lib/v1/names/nested.test.js

const chai = require("chai");

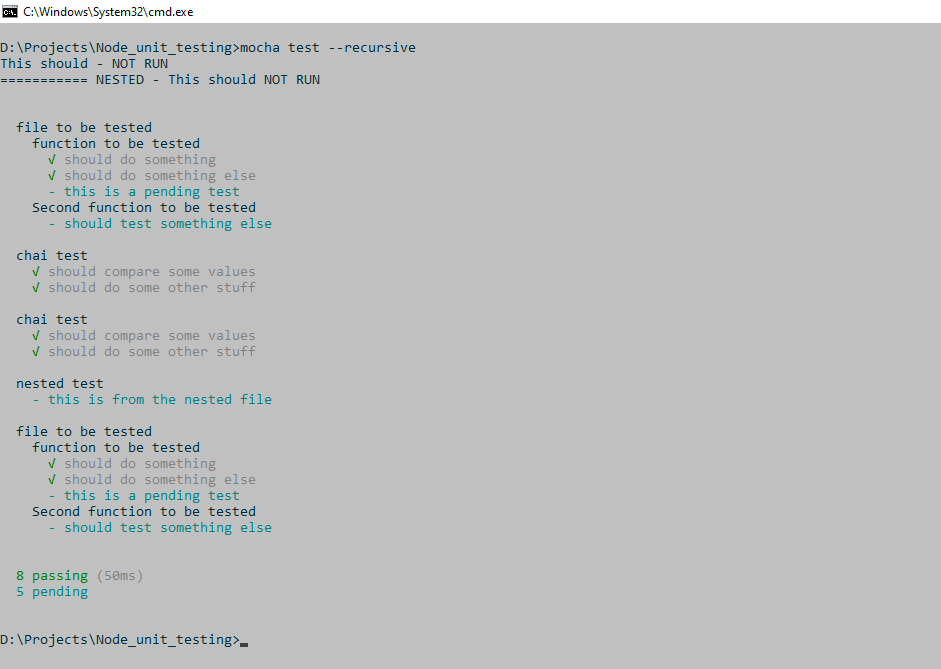
const expect = chai.expect;

describe("nested test", () => {

  it("this is from the nested file");

});

* Now run mocha test—recursive. All tests inside test directory run



Search by pattern way

mocha ./lib/\*\*/\*.test.js // executes all the .test.js files inside lib dir

Sometime it gives wrong result, therefore always put single quotes in path

mocha ‘./lib/\*\*/\*.test.js ‘ // executes all the .test.js files inside lib dir

only .test.js runs, all the .js file are left

* Now the we will write the same code in package.json file and run from npm

package.json

{

  "name": "node\_unit\_testing",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "mocha './lib/\*\*/\*.test.js'"

  },

  "author": "Amir Mustafa",

  "license": "ISC",

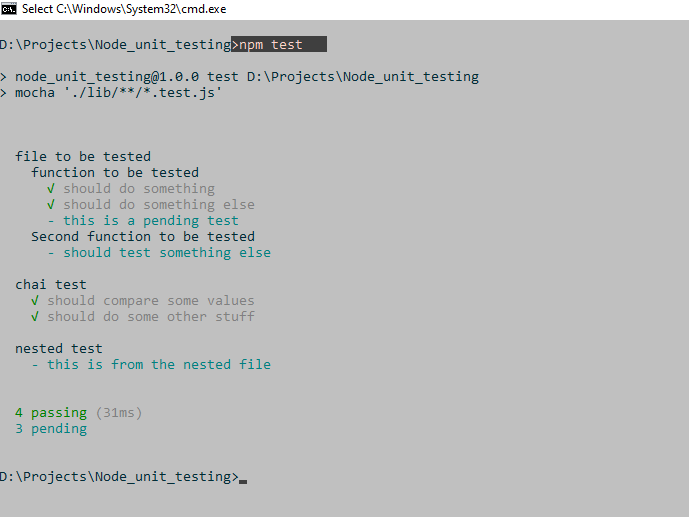
  "dependencies": {

    "chai": "^4.2.0"

  }

}

Run this command to execute - npm test or npm run test



Environmental Variables and cross-env

* When suppose in test file you are using env data,

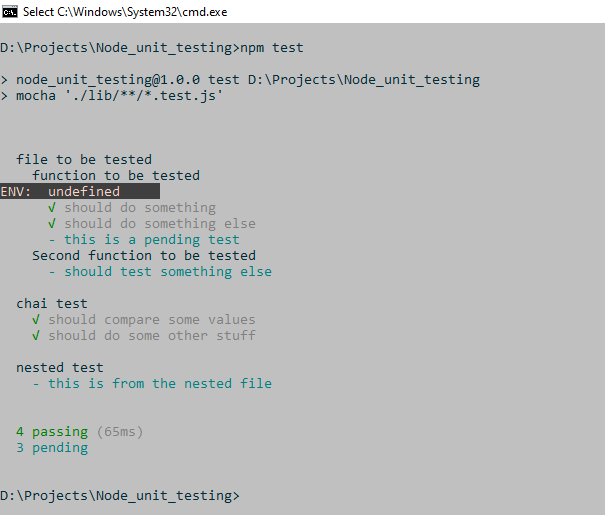
Eg.

it("should do something", () => {

      assert.equal(1, 1);

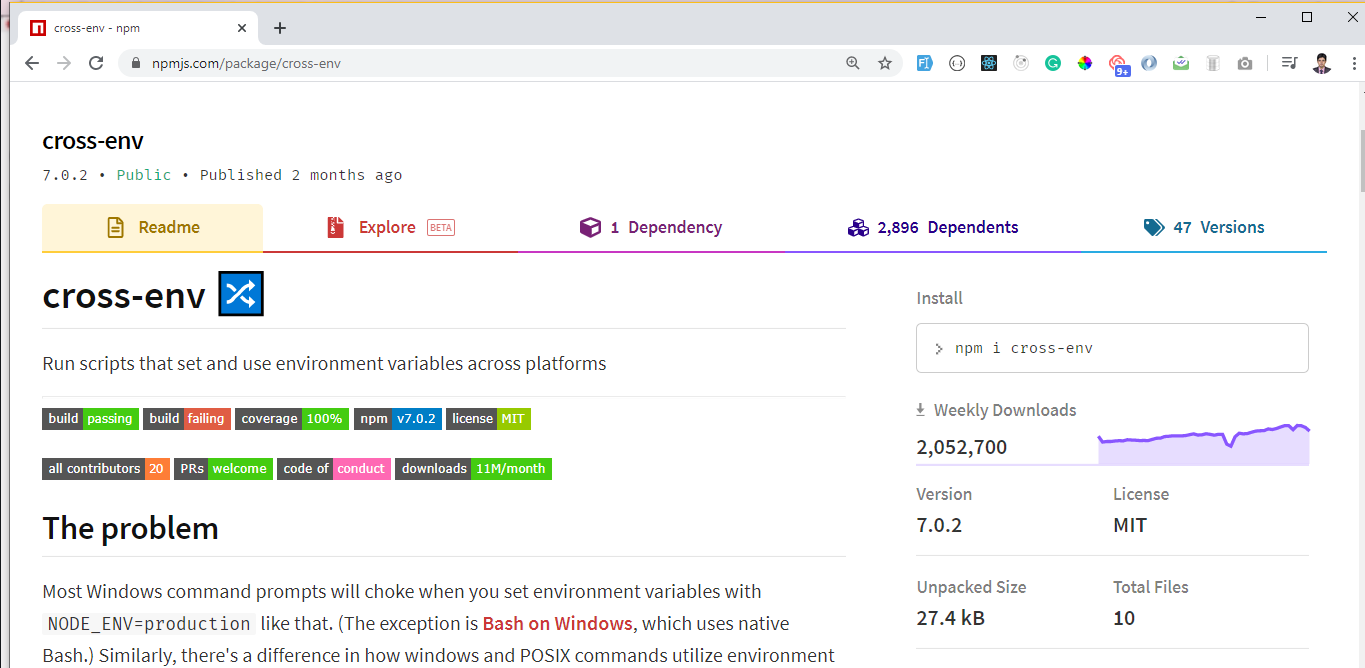
      console.log("ENV: ", process.env.NODE\_ENV); // prints undefined

    });



So to read it – package called cross-env

npm i cross-env



In the package.json file add line in test

{

  "name": "node\_unit\_testing",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "cross-env NODE\_ENV=development mocha './lib/\*\*/\*.test.js'"

  },

  "author": "Amir Mustafa",

  "license": "ISC",

  "dependencies": {

    "chai": "^4.2.0",

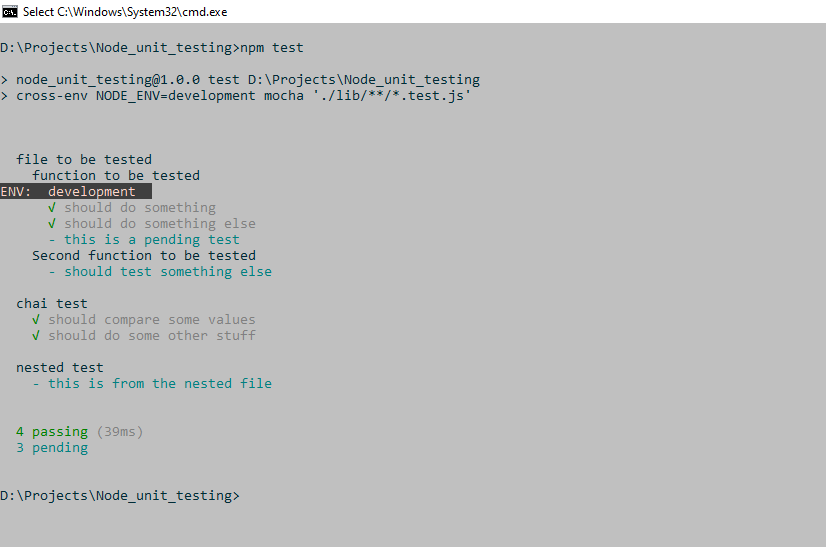
    "cross-env": "^7.0.2"

  }

}

Now run the tests:

npm test



Eg2: bar.test.js

it("should do something", () => {

      assert.equal(1, 1);

      // console.log("ENV: ", process.env.NODE\_ENV);

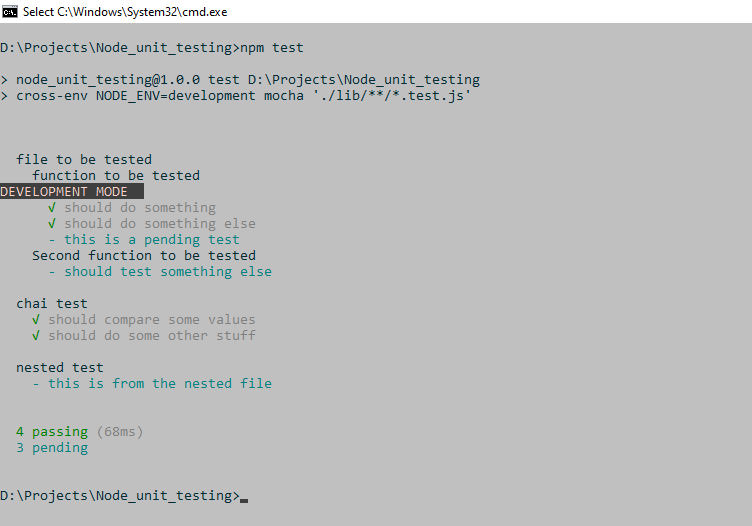
      if (process.env.NODE\_ENV === "development") {

        // run some variable here for testing

        console.log("DEVELOPMENT MODE");

      }

    });



NOTE: Never use testing in production (there are some db insert/delete code etc.)

**Basics of testing**

1. **Standard Functions:**

project/lib/demo.js

exports.add = function (a, b) {

    return a + b

}

project/lib/demo.test.js

const chai = require("chai");

const expect = chai.expect;

var demo = require("./demo");

describe("demo", () => {

  context("add", () => {

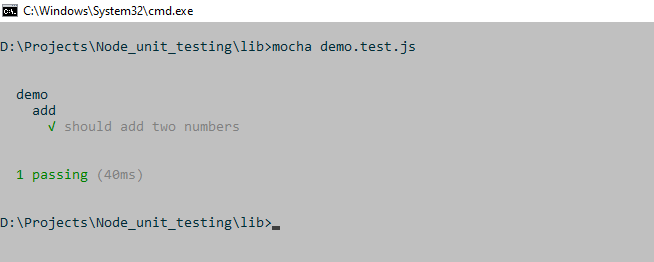
    it("should add two numbers", () => {

      expect(demo.add(1, 2)).to.equal(3);

    });

  });

});



1. **Callback Functions:**

demo.js

exports.addCallback = function (a, b, callback) {

  setTimeout(() => {

    return callback(null, a + b);

  }, 500);

};

demo.test.js

const chai = require("chai");

const expect = chai.expect;

var demo = require("./demo");

describe("demo", () => {

  context("add", () => {

    it("should add two numbers", () => {

      expect(demo.add(1, 2)).to.equal(3);

    });

  });

  context("callback add", () => {

    it("should test the callback", (done) => { // Parameter added

      demo.addCallback(1, 2, (err, result) => {

        expect(err).to.not.exist;

        expect(result).to.equal(3);

        done(); // must for callback

      });

    });

  });

});

1. **Promises**

demo.js

exports.addPromise = function (a, b) {

  // return Promise.reject(new Error('fake'))

  return Promise.resolve(a + b);

};

demo.test.js

context("test promise", () => {

    it("should add with the promise callback", (done) => {

      demo.addPromise(1, 2).then((result) => {

        expect(result).to.equal(3);

        done();

      });

    });

  });

Eg2. Catch the error if any

demo.js

exports.addPromise = function (a, b) {

  return Promise.reject(new Error("fake"));

  // return Promise.resolve(a + b);

};

demo.test.js

context("test promise", () => {

    it("should add with the promise callback", (done) => {

      demo

        .addPromise(1, 2)

        .then((result) => {

          expect(result).to.equal(3);

          done();

        })

        .catch((err) => {

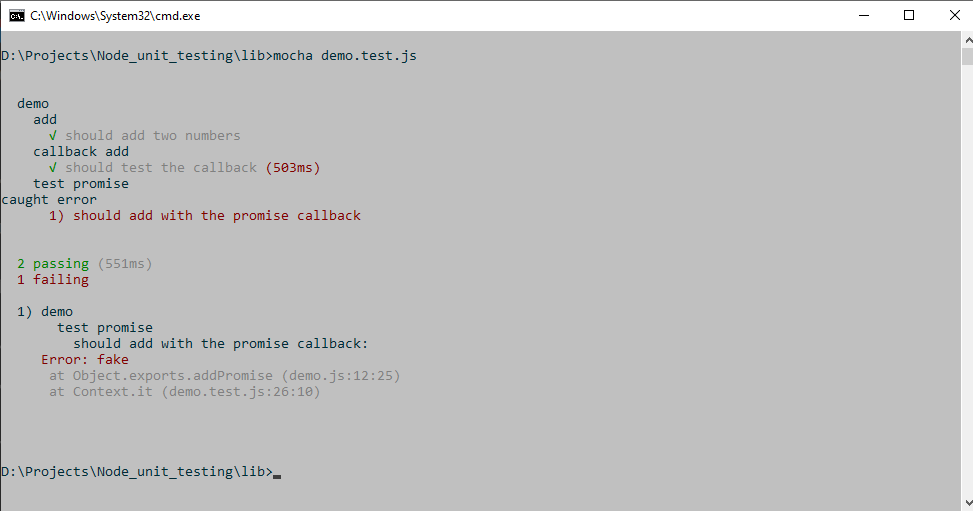
          console.log("caught error");

          done(err);

        });

    });

  });



Eg 3 – Using return style

demo.test.js

context("test promise", () => {

    // it("should add with the promise callback", (done) => {

    //   demo

    //     .addPromise(1, 2)

    //     .then((result) => {

    //       expect(result).to.equal(3);

    //       done();

    //     })

    //     .catch((err) => {

    //       console.log("caught error");

    //       done(err);

    //     });

    // });

    it("should test promise with return", () => {

      return demo.addPromise(1, 2).then((result) => {

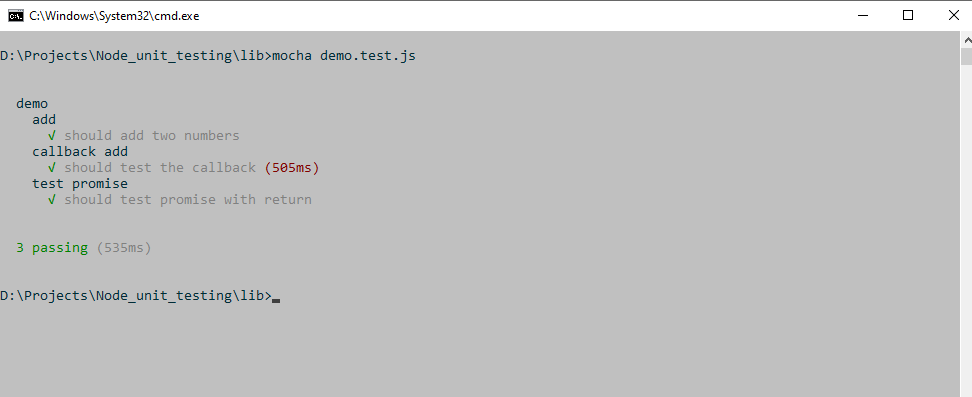
        expect(result).to.be.equal(3);

      });

    });

  });

NOTE: No need to write any catch for errors, return auto detects and throws like catch



Eg4 – calling promise by async await

demo.js

exports.addPromise = function (a, b) {

  // return Promise.reject(new Error('fake'))

  return Promise.resolve(a + b);

};

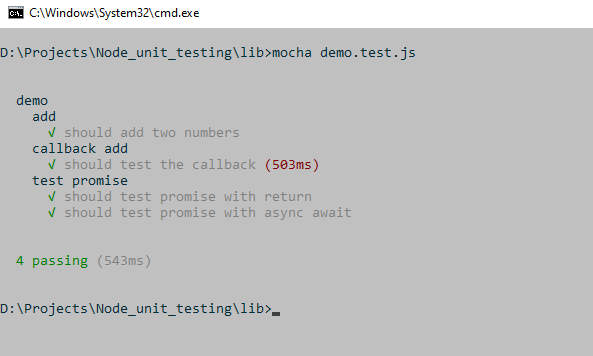
demo.test.js

it("should test promise with async await", async () => {

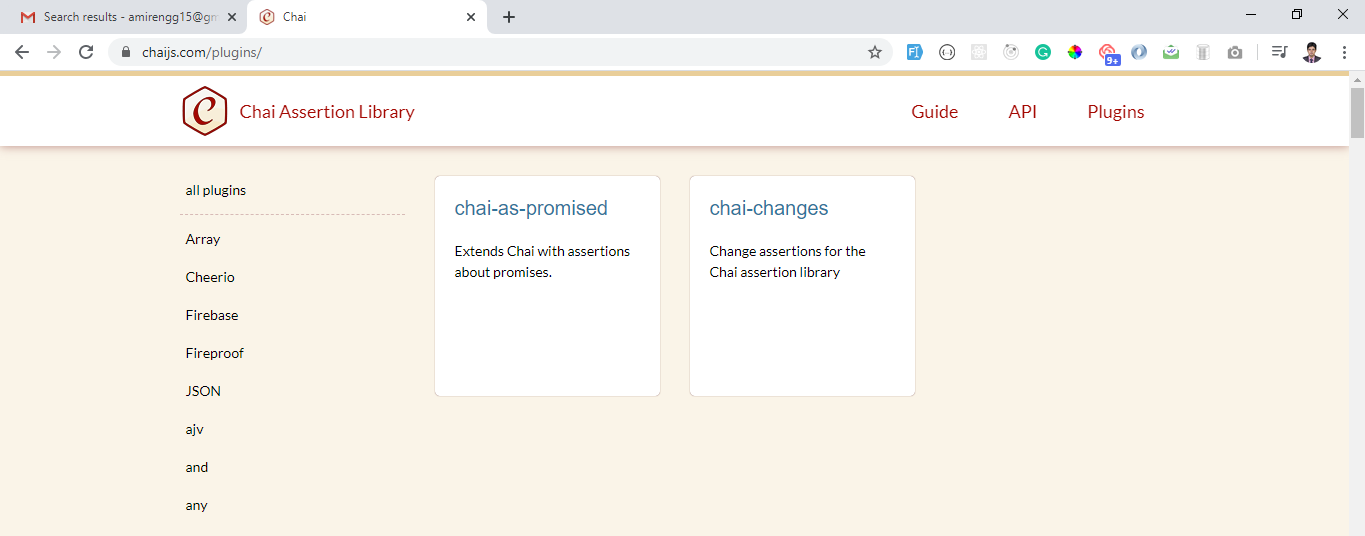
   let result = await demo.addPromise(1, 2);

   expect(result).to.equal(3);

});



Eg 5 – Calling by chai plugin – chai-as-promised



npm i chai-as-promised // this also uses async-await

demo.test.js

const chai = require("chai");

const expect = chai.expect;

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

var demo = require("./demo");

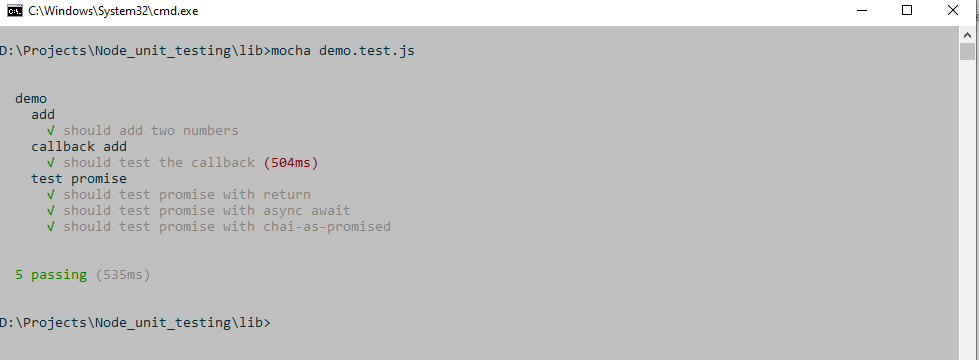
…

…

it("should test promise with chai-as-promised", async () => {

     await expect(demo.addPromise(1, 2)).to.eventually.equal(3);

});



Sinon

Sinon helps eliminate complexity in tests by allowing you to easily create so called ***test-doubles***.

Test-doubles are, like the name suggests, replacements for pieces of code used in your tests. Looking back at the Ajax example, instead of setting up a server, we would replace the Ajax call with a test-double. With the time example, we would use test-doubles to allow us to “travel forwards in time”.

Test doubles are like functions we can inject into our programs from spies or stubs, they will let us know what is happening behind the scene.

Eg.

demo.js

//spy on log

exports.foo = () => {

  //some operation

  console.log("console.log was called");

  console.warn("console.warn was called");

  return;

};

demo.test.js

const chai = require("chai");

const expect = chai.expect;

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

const sinon = require("sinon"); // for way 1

const sinonChai = require("sinon-chai"); //for way 2

chai.use(sinonChai); //way 2

var demo = require("./demo");

…

…

context("test doubles", () => {

    it("should spy on log", () => {

      let spy = sinon.spy(console, "log");

      demo.foo();

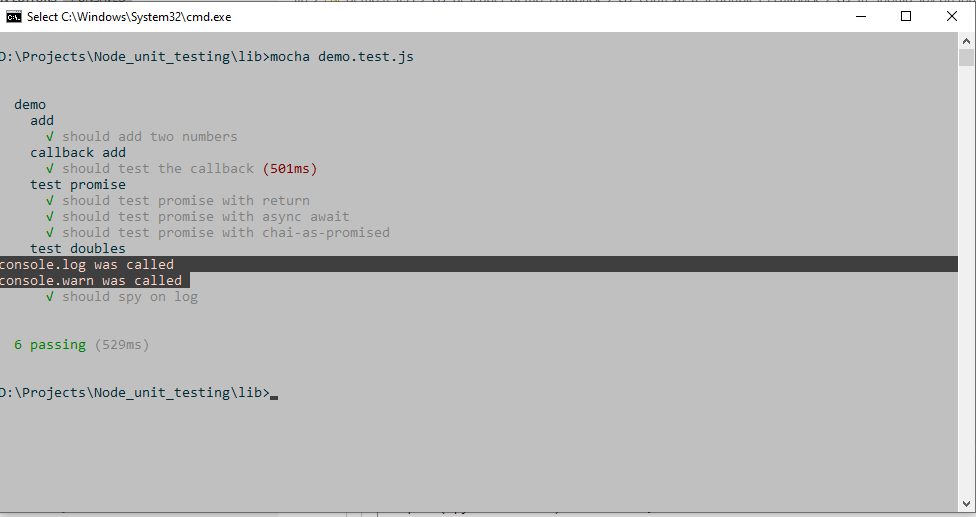
      expect(spy.calledOnce).to.be.true; // way 1 using sinon

      expect(spy).to.have.been.calledOnce; // way 2 using sinon-chai, (both are same)

      spy.restore();

    });

  });



Code Isolation / Stubs

* One of the important part of unit testing is code isolation and stubs. For example there are functions which require saving in database and console something, we are do not have to actually save in database and pass that save. Therefore stubs is used.
* So actual code is checked whether running or not but not executed.

demo.js

//stub on warn

exports.foo = () => {

  //some operation

  console.log("console.log was called");

  console.warn("console.warn was called");

  return;

};

demo.test.js

it("should stub console warn", () => {

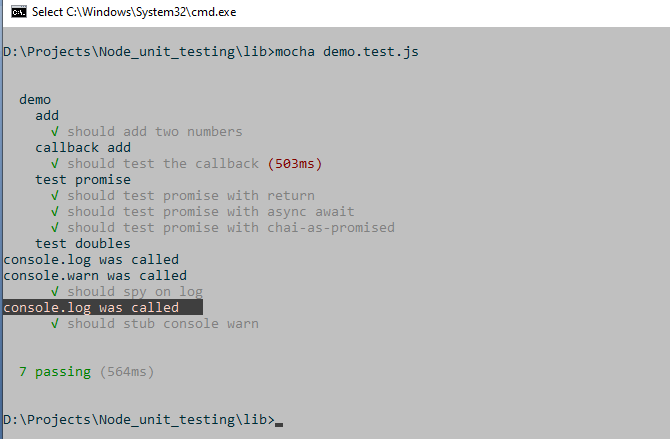
   let stub = sinon.stub(console, "warn");// console.warn will not print, but test will pass

   demo.foo();

   expect(stub).to.have.been.calledOnce;

});

// console.warn was not called for stub



Eg 2 – sending my message instead of stubbed function message

it("should stub console warn", () => {

      let stub = sinon.stub(console, "warn").callsFake(() => {

        console.log("message from stub");

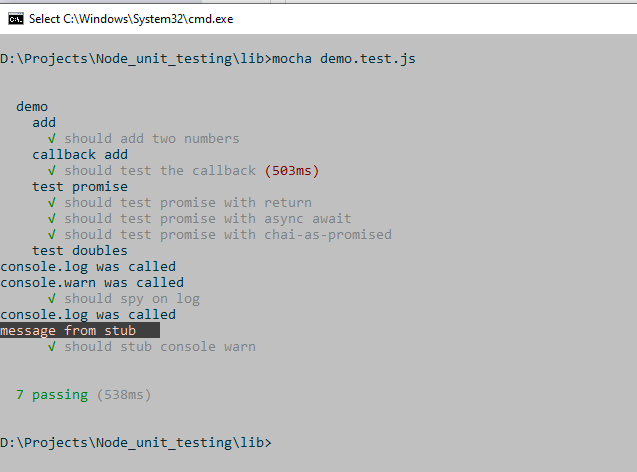
      });

      demo.foo();

      expect(stub).to.have.been.calledOnce;

stub.restore(); // this is mandatory to close for next stub or spy call

});



* In real case, if there is a database query and you do not want to run it, after running one can return fake object to testing.

Eg3 – To test whether we have stubbed function o/p

it("should stub console warn", () => {

      let stub = sinon.stub(console, "warn").callsFake(() => {

        console.log("message from stub");

      });

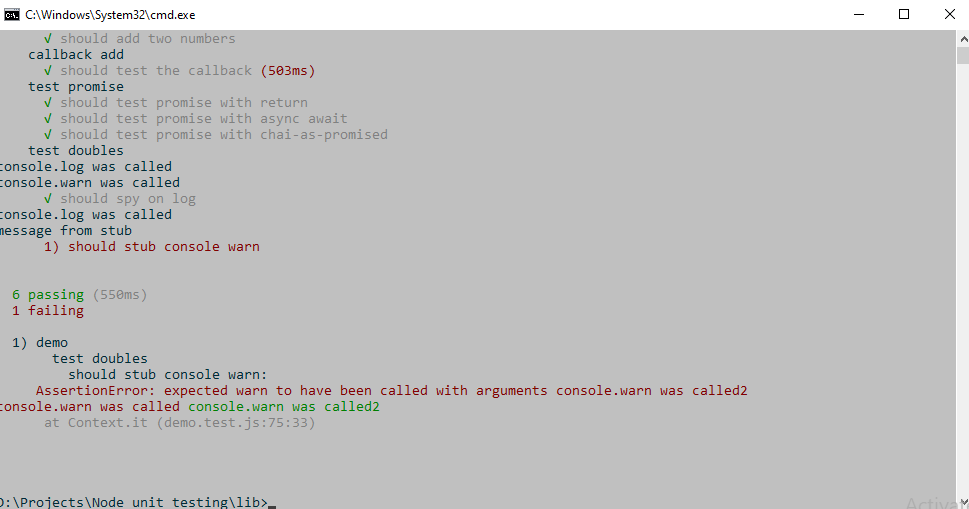
      demo.foo();

      expect(stub).to.have.been.calledOnce;

      expect(stub).to.have.been.calledWith("console.warn was called2"); // this string should o/p of actual function we stubbed - 2 should not be there

stub.restore();

});



This is useful when we want o/p should have certain **name property, etc**

Rewire

Rewire adds a special setter and getter to modules so you can modify their behaviour for better unit testing. You may

* inject mocks for other modules or globals like process
* inspect private variables
* override variables within the module.

Ref - <https://github.com/jhnns/rewire>

Whenever there is call to private function, inside a function, we use rewire to call it

TRICK

1. Load the rewire module
2. Replace require with rewire for the file to test

Eg. demo.js

//stub createfile

exports.bar = async (fileName) => {

  await exports.createFile(fileName);

  let result = await callDB(fileName); // private function call – therefore use rewire

  return result;

};

demo.test.js

const chai = require("chai");

const expect = chai.expect;

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

const sinon = require("sinon");

const sinonChai = require("sinon-chai");

chai.use(sinonChai);

const rewire = require("rewire");

var demo = rewire("./demo"); // replace require with rewire

…

…

context("stub private functions with rewire", () => {

    it("should stub createFile", async () => {

      let createStub = sinon.stub(demo, "createFile").resolves("create\_stub");

      let callStub = sinon.stub().resolves("calldb\_stub");

      demo.\_\_set\_\_("callDB", callStub);

      let result = await demo.bar("test.txt");

      expect(result).to.equal("calldb\_stub");

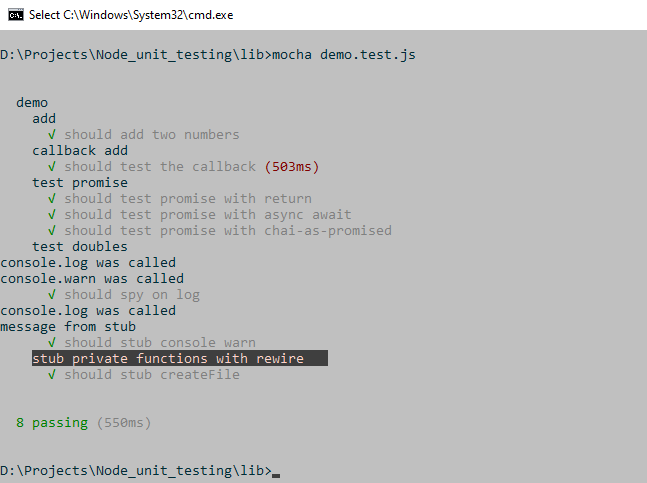
      expect(createStub).to.be.calledOnce;

      expect(createStub).to.have.been.calledWith("test.txt");

      expect(callStub).to.be.calledOnce;

    });

  });



Important Reminder

Please remember to use **var** instead of **const**when using rewire to import a module, it's easy to miss when you use const everything at the top of your code.

This is because rewire will inject the rewired versions, and we reset that during teardown. It's ok to use const on requiring rewire itself, but use var for everything else.

**const**rewire = require('rewire');

**var** myModule = rewire('../path/to/custom/module');

1. Users.get()

model/users.js

var mongoose = require('mongoose');

var UserSchema = mongoose.Schema({

    name: {type: String, required: true},

    email: {type: String, required: true},

    age: Number

}, {

    collection: 'users'

}); //overrides default collection name auto created

module.exports = mongoose.model('User', UserSchema);

Users.js

exports.get = function (id, callback) {

    if (!id) {

        return callback(new Error('Invalid user id'));

    }

    User.findById(id, function (err, result) {

        if (err) {

            return callback(err);

        }

        return callback(null, result);

    });

}

users.test.js

const chai = require("chai");

const expect = chai.expect;

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

const sinon = require("sinon");

const sinonChai = require("sinon-chai");

chai.use(sinonChai);

const rewire = require("rewire");

var mongoose = require("mongoose");

var users = require("./users");

var User = require("./models/user");

var sandbox = sinon.createSandbox();

describe("users", () => {

  let findStub;

  let sampleArgs;

  let sampleUser;

  beforeEach(() => {

    sampleUser = {

      // this data should be similar to model schema data

      id: 123,

      name: "amir",

      email: "amirengg15@gmail.com",

      age: 27,

    };

    findStub = sandbox.stub(mongoose.Model, "findById").resolves(sampleUser);

  });

  afterEach(() => {

    sandbox.restore(); // this equal to stub.restore();  but for sandbox version

  });

context("get", () => { // call user.get

    it("should check for an id", (done) => {

      users.get(null, (err, result) => {

        expect(err).to.exist;

        expect(err.message).to.equal("Invalid user id"); // this err msg is exact same as in function testing

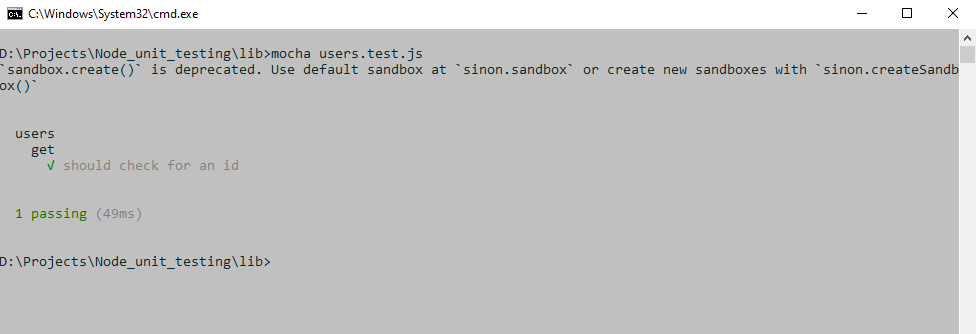
        // expect(err.message).to.equal("Invalid user idasa"); // Err msg not same as function ==> fail test

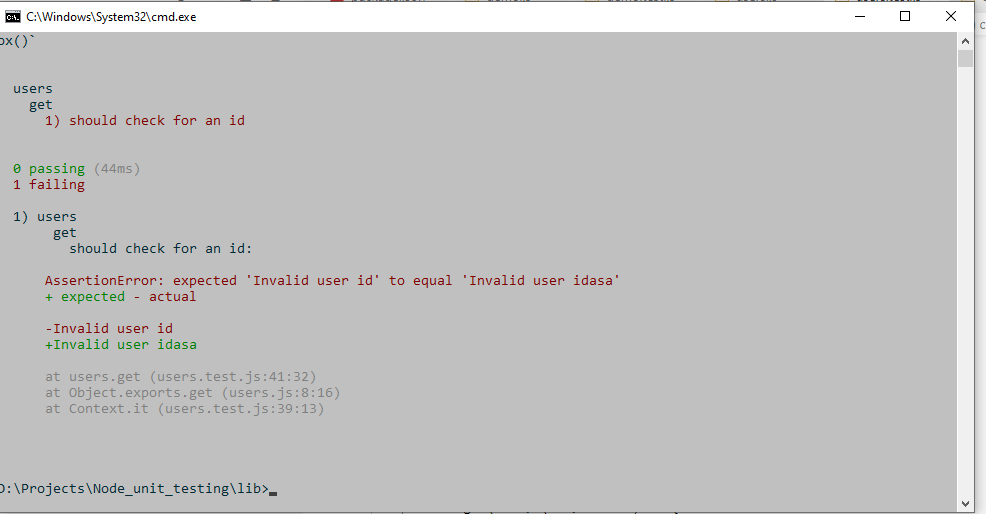
        done();

      });

    });

Run mocha **users.test.js**





context("get", () => {

    it("should check for an id", (done) => {

      users.get(null, (err, result) => {

        expect(err).to.exist;

        expect(err.message).to.equal("Invalid user id"); // this err msg is exact same as in function testing

        done();

      });

    });

    it("should call findUserById with id and return result", (done) => {

      sandbox.restore();

      let stub = sandbox

        .stub(mongoose.Model, "findById")

        .yields(null, { name: "amir" }); // 1st para = err if any, 2nd = data

      users.get(123, (err, result) => { // my id of object

        expect(err).to.not.exist;

        expect(stub).to.have.been.calledOnce;

        expect(stub).to.have.been.calledWith(123);

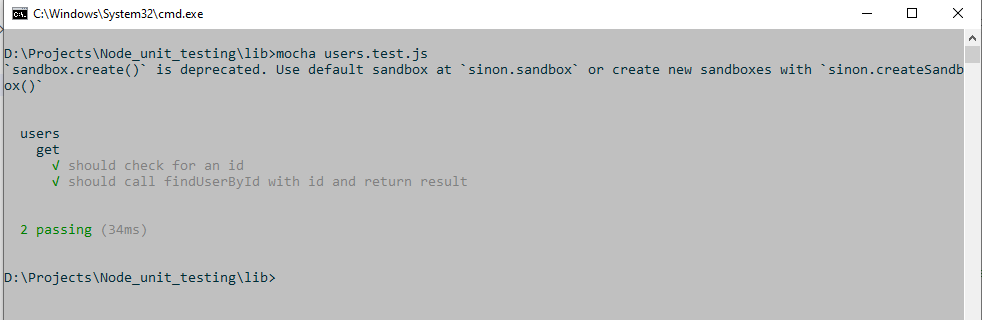
        expect(result).to.be.a("object");

        expect(result).to.have.property("name").to.equal("amir"); // my object name

        done();

      });

    });



Catch error if there is one

it("should catch error if there is one", (done) => {

      sandbox.restore();

      let stub = sandbox

        .stub(mongoose.Model, "findById")

        .yields(new Error("fake")); // think actual error is replaced by this from function

      users.get(123, (err, result) => {

        expect(result).to.not.exist;

        expect(err).to.exist;

        expect(err).to.be.instanceOf(Error);

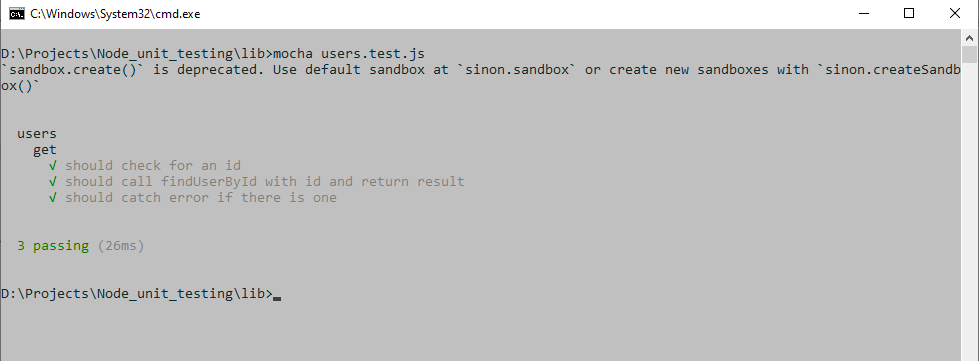
        expect(stub).to.have.been.calledWith(123);

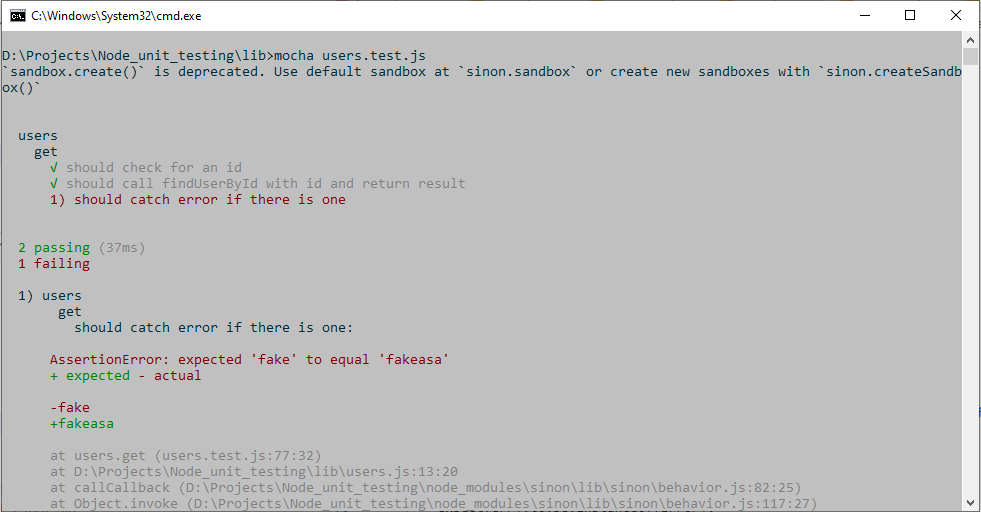
        expect(err.message).to.equal("fake"); // test pass

        // expect(err.message).to.equal("fakeasa");     // will show fail test as err msg above not matched

        done();

      });





1. Users.delete()

users.js

exports.delete = function (id) {

    // return Promise.resolve()

    if (!id) {

        return Promise.reject(new Error('Invalid id'));

    }

    return User.remove({

        \_id: id

    });

}

users.test.js

context("delete user", () => {

// 1st way of writing shorter test

it("should check for an id using return", () => {

      return users

        .delete()

        .then((result) => {

          throw new Error("unexpected success");

        })

        .catch((ex) => {

          expect(ex).to.be.instanceOf(Error);

          expect(ex.message).to.equal("Invalid id");

        });

    });

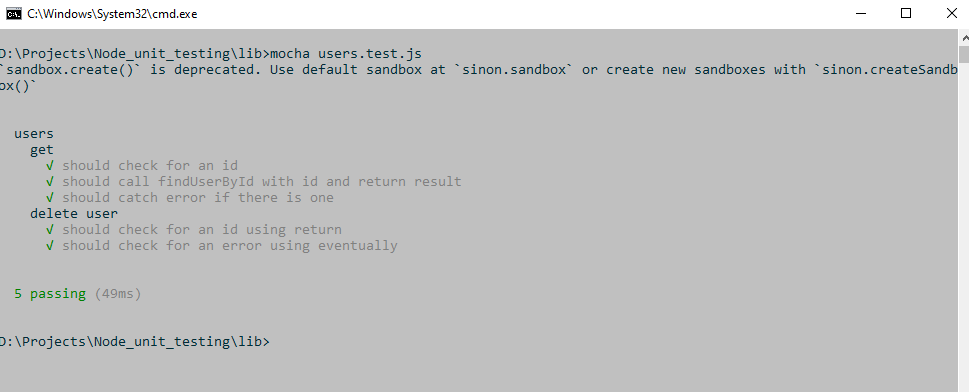
    // 2nd way of writing shorter test - sinonChai plugin

    it("should check for an error using eventually", () => {

      return expect(users.delete()).to.eventually.be.rejectedWith("Invalid id");

    });

  });



3rd way to delete async – await

context("delete user", () => {

//3rd way - using async await

    it("should call User.remove", async () => {

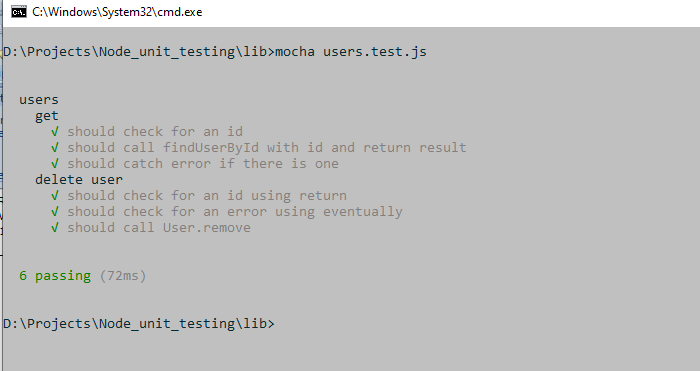
      let result = await users.delete(123);

      expect(result).to.equal("fake\_remove\_result");

      expect(deleteStub).to.have.been.calledWith({ \_id: 123 });

    });

  });



1. Users.create

const rewire = require("rewire");

var mongoose = require("mongoose");

var users = rewire("./users");

var User = require("./models/user");

var mailer = require("./mailer");

describe("users", () => {

  let findStub;

  let deleteStub;

  let sampleArgs;

  let sampleUser;

  beforeEach(() => {

    sampleUser = {

      // this data should be similar to model schema data

      id: 123,

      name: "amir",

      email: "amirengg15@gmail.com",

      age: 27,

    };

    findStub = sandbox.stub(mongoose.Model, "findById").resolves(sampleUser);

    deleteStub = sandbox

      .stub(mongoose.Model, "remove")

      .resolves("fake\_remove\_result");

    mailerStub = sandbox

      .stub(mailer, "sendWelcomeEmail")

      .resolves("fake\_email");

  });

  afterEach(() => {

    sandbox.restore(); // this equal to stub.restore();  but for sandbox version

    users = rewire("./users");

  });

context("create user", () => {

    let FakeUserClass, saveStub, result;

    beforeEach(async () => {

      saveStub = sandbox.stub().resolves(sampleUser); // creating stub

      FakeUserClass = sandbox.stub().returns({ save: saveStub }); // save is the name of function class uses - for this rewire is used above

      users.\_\_set\_\_("User", FakeUserClass);

      result = await users.create(sampleUser); // TEST SAVE PASS CASE

    });

    it("should reject invalid args", async () => { // TEST SAVE ERROR CASES

      await expect(users.create()).to.eventually.be.rejectedWith(

        "Invalid arguments"

      );

      await expect(

        users.create({ name: "amir" })

      ).to.eventually.be.rejectedWith("Invalid arguments");

      await expect(

        users.create({ email: "amirengg15@gmail.com" })

      ).to.eventually.be.rejectedWith("Invalid arguments");

    });

    it("should call user with new", () => { // TEST FOR CORRECT OBJECT CALLED

      expect(FakeUserClass).to.have.been.calledWithNew;

      expect(FakeUserClass).to.have.been.calledWith(sampleUser);

    });

  });

});

Eg2

 ///////////// users.create users //////////////

  context("create user", () => {

    let FakeUserClass, saveStub, result;

    beforeEach(async () => {

      saveStub = sandbox.stub().resolves(sampleUser);

      FakeUserClass = sandbox.stub().returns({ save: saveStub }); // save is the name of function class uses - for this rewire is used above

      users.\_\_set\_\_("User", FakeUserClass);

      result = await users.create(sampleUser);

    });

    // rejection tests

    it("should reject invalid args", async () => {

      await expect(users.create()).to.eventually.be.rejectedWith(

        "Invalid arguments"

      );

      await expect(

        users.create({ name: "amir" })

      ).to.eventually.be.rejectedWith("Invalid arguments");

      await expect(

        users.create({ email: "amirengg15@gmail.com" })

      ).to.eventually.be.rejectedWith("Invalid arguments");

    });

    // new should be called once

    it("should call user with new", () => {

      expect(FakeUserClass).to.have.been.calledWithNew;

      expect(FakeUserClass).to.have.been.calledWith(sampleUser);

    });

    it("should save the user", () => {

      expect(saveStub).to.have.been.called;

    });

    it("should call mailer with email and name", () => {

      expect(mailerStub).to.have.been.calledWith(

        sampleUser.email,

        sampleUser.name

      ); // these parameter is expected in

    });

    it("should reject errors", async () => {

      saveStub.rejects(new Error("fake"));

      await expect(users.create(sampleUser)).to.eventually.be.rejectedWith(

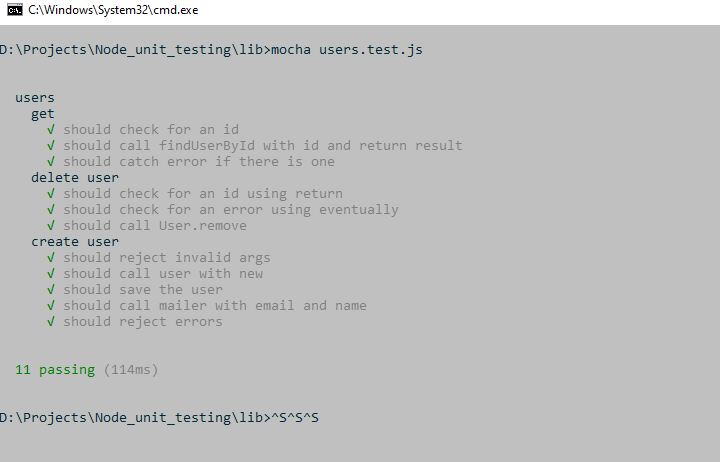
        "fake"

      );

    });

  });

});



1. users.update

users.js

exports.update = async function (id, data) {

  try {

    var user = await User.findById(id); // Write test for this

    for (var prop in data) {

      user[prop] = data[prop];

    }

    var result = await user.save(); // write test for this

    return result;

  } catch (err) {

    // console.warn(err);

    return Promise.reject(err); // one test for rejection

  }

};

Users.test.js

///////////// users.update //////////////

  context("update user", () => {

    it("should find user by id", async () => {

      await users.update(123, { age: 35 });

      expect(findStub).to.have.been.calledWith(123); // find stub called

    });

    it("should call user.save", async () => {

      await users.update(123, { age: 35 });

      expect(sampleUser.save).to.have.been.calledOnce; // sampleUser.save called

    });

    it("should reject if there is an error", async () => {

      findStub.throws(new Error("fake")); // forcefully throws error to check error case

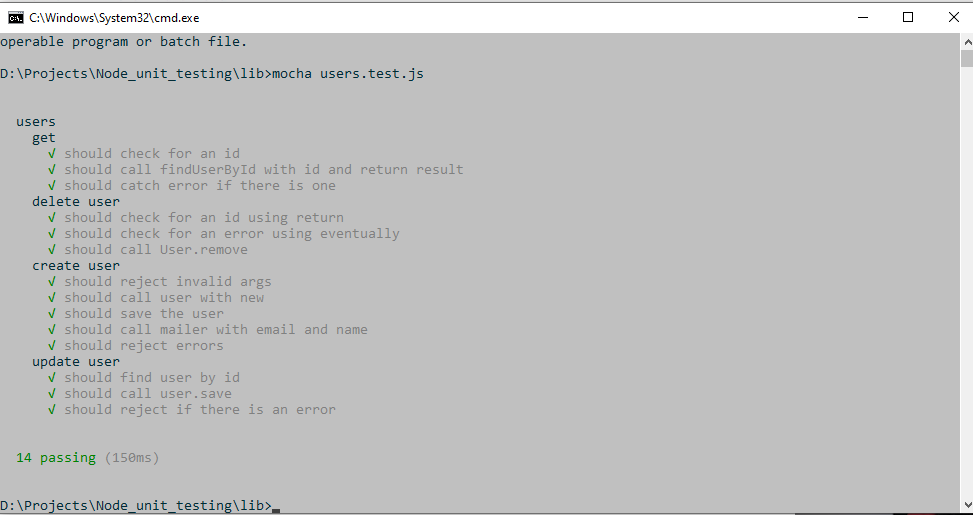
      await expect(

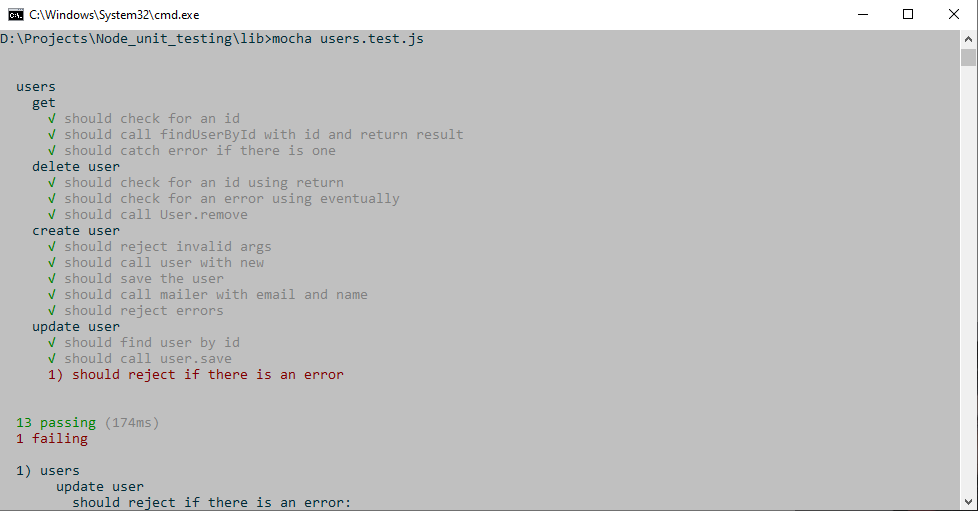
        users.update(123, { age: 35 })

      ).to.eventually.be.rejectedWith("fakeasa");

    });

  });





Reset Password

Users.js

exports.resetPassword = function (email) {

  if (!email) {

    return Promise.reject(new Error("Invalid email")); // write test for this

  }

  //some operations

  return mailer.sendPasswordResetEmail(email); // write test for this

};

Users.test.js

///////////// reset password //////////////

  context("reset password", () => {

    let resetStub;

    beforeEach(() => {

      resetStub = sandbox.stub(mailer, "sendPasswordResetEmail").resolves("email");

    });

    it("should check for email", async () => {

      await expect(users.resetPassword()).to.eventually.be.rejectedWith(

        "Invalid email"

      ); // no arguments passed inside resetPassword() , therefore error

    });

    it("should call sendPasswordResetEmail", async () => {

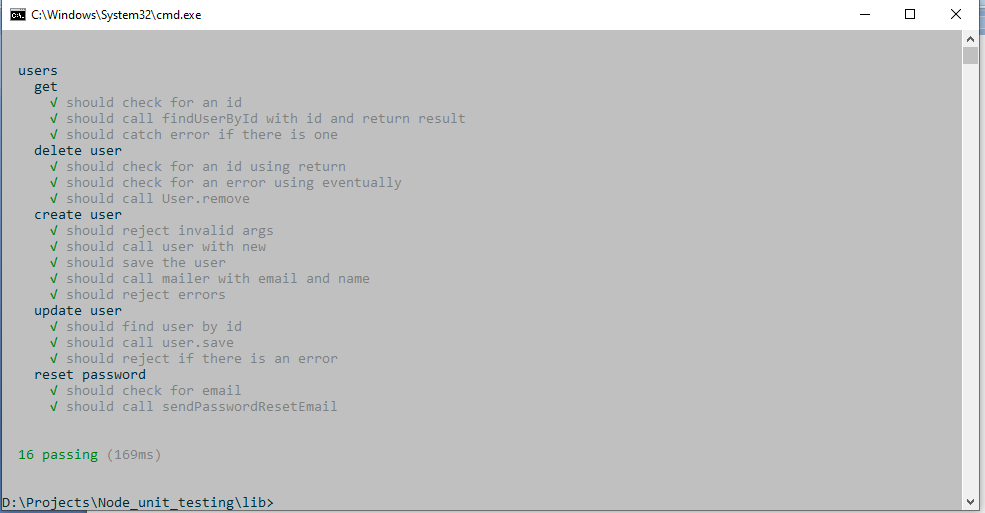
      await users.resetPassword("amirengg15@gmail.com"); // testing email, export.resetPassword called

      expect(resetStub).to.have.been.calledWith("amirengg15@gmail.com"); //expected email = should be same as above

    });

  });

Run – mocha user.tests.js



Mailer

Page – mailer.js

Unit test page – mailer.test.js

Mailer.js

exports.sendWelcomeEmail = function (email, name) {

  // console.log('--- in mailer > sendWelcomeEmail');

  if (!email || !name) {

    return Promise.reject(new Error("Invalid input")); // write test for this

  }

  var body = `Dear ${name}, welcome to our family!`; //write test for this

  return sendEmail(email, body);

};

Mailer.test.js

const chai = require("chai");

const expect = chai.expect;

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

const sinon = require("sinon");

const sinonChai = require("sinon-chai");

chai.use(sinonChai);

const rewire = require("rewire");

var sandbox = sinon.sandbox.create();

var mailer = rewire("./mailer");

describe("mailer", () => {

  let emailStub;

  beforeEach(() => {

    emailStub = sandbox.stub().resolves("done");

    mailer.\_\_set\_\_("sendEmail", emailStub);

  });

  afterEach(() => {

    sandbox.restore();

    mailer = rewire("./mailer"); // mandatory test

  });

  ///////////// Send Welcome Email //////////////

  context("sendWelcomeEmail", () => {

    // rejected mail test

    it("should check for email and name", async () => {

      await expect(mailer.sendWelcomeEmail()).to.eventually.be.rejectedWith(

        "Invalid input"

      );

      await expect(

        mailer.sendWelcomeEmail("amirengg15@gmail.com")

      ).to.eventually.be.rejectedWith("Invalid input");

    });

    // return with email and password test

    it("should call sendEmail with email and password", async () => {

      mailer.sendWelcomeEmail("amirengg15@gmail.com", "Amir");

      expect(emailStub).to.have.been.calledWith(

        "amirengg15@gmail.com",

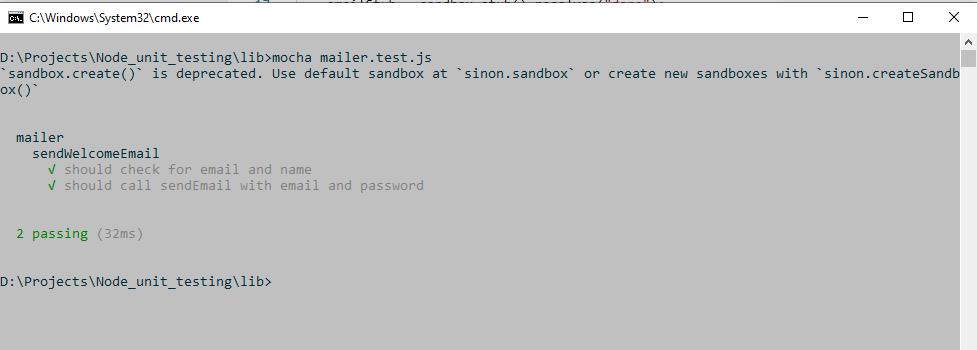
        "Dear Amir, welcome to our family!"

      );

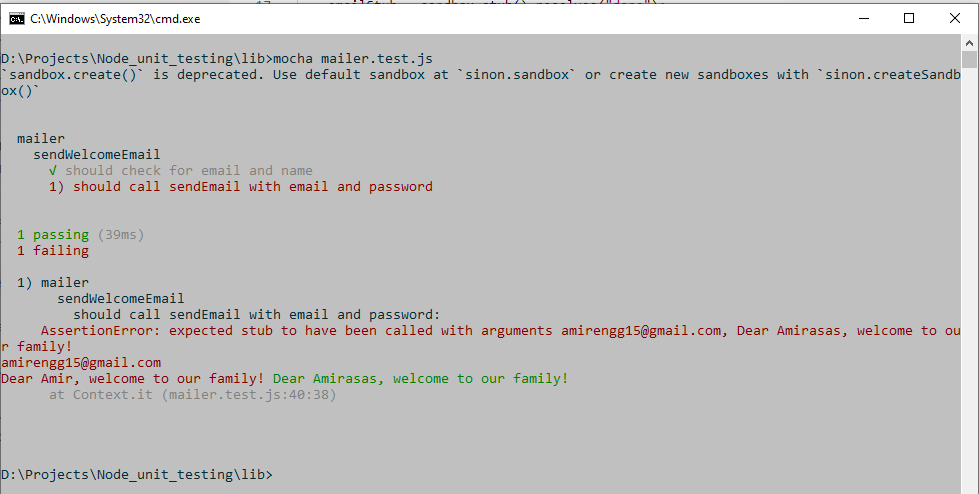
    });

  });

});



If wrong msg passed



1. Test reset email

Mailer.js

exports.sendPasswordResetEmail = function (email) {

  // console.log('--- in mailer > sendPasswordResetEmail');

  if (!email) {

    return Promise.reject(new Error("Invalid input"));

  }

  var body = "Please click http://some\_link to reset your password.";

  return sendEmail(email, body);

};

Mailer.test.js

///////////// Send Password Reset Email //////////////

  context("sendPasswordResetEmail", () => {

    // rejected mail test

    it("should check for email", async () => {

      await expect(

        mailer.sendPasswordResetEmail() // empty email send – throw error

      ).to.eventually.be.rejectedWith("Invalid input");

    });

    // return with email and password test

    it("should call sendEmail with email and password", async () => {

      mailer.sendPasswordResetEmail("amirengg15@gmail.com");

      expect(emailStub).to.have.been.calledWith(

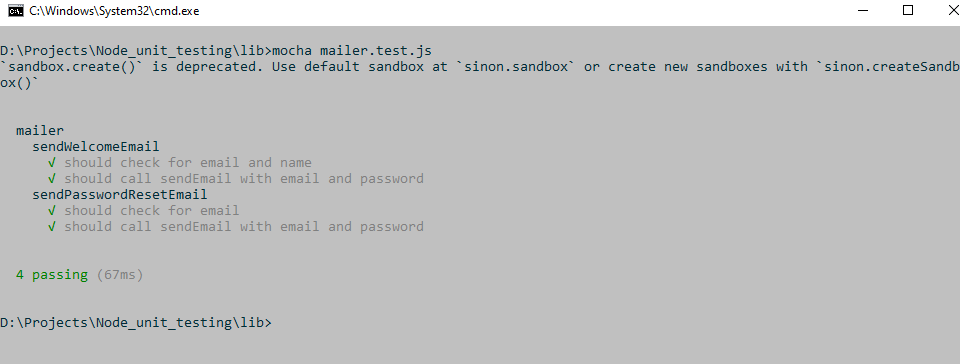
        "amirengg15@gmail.com",

        "Please click http://some\_link to reset your password."

      );

    });

  });



1. Send mail function (this is the private function as no export is written – using reqire \_\_get\_\_)

Mailer.js

function sendEmail(email, body) {

  // console.log('--- in mailer > sendEmail');

  if (!email || !body) {

    return Promise.reject(new Error("Invalid input")); // write test for this

  }

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      console.log("Email Sent!");

      // return reject(new Error('Fake Error'));

      return resolve("Email sent"); // write test for this

    }, 100);

  });

}

Mailer.test.js

///////////// Send Email //////////////

  context("sendEmail", () => {

    let sendEmail;

    beforeEach(() => {

      mailer = rewire("./mailer");

      sendEmail = mailer.\_\_get\_\_("sendEmail"); // we are rewiring it because this is the private function

    });

    // Test 1

    it("should check for email and body", async () => {

      await expect(sendEmail()).to.eventually.be.rejectedWith("Invalid input");

      await expect(

        sendEmail("amirengg15@gmail.com")

      ).to.eventually.be.rejectedWith("Invalid input");

    });

    // Test 2

    it("should should call sendEmail with email and password", async () => {

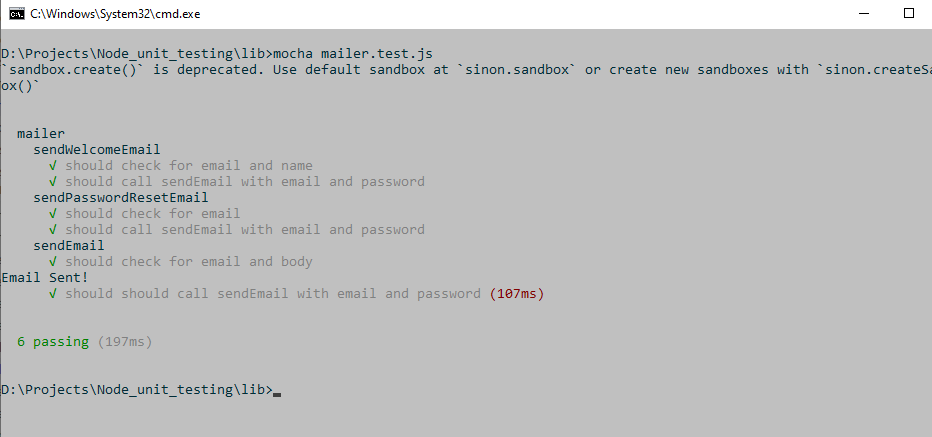
      // stub actual mailer

      let result = await sendEmail("amirengg15@gmail.com", "welcome");

      expect(result).to.equal("Email sent");

    });

  });



Utils.js

const crypto = require("crypto");

const config = require("./config");

//foo = 1f0c01e25707f55ed3014d60bd0d0373

exports.getHash = function (string) {

  if (!string || typeof string !== "string") return null; // write test for this

  string += "\_" + config.secret(); // write test to get secret

  var hash = crypto.createHash("md5").update(string).digest("hex");

  // console.log('Hash: ' , hash);

  return hash; // write test to get hash

};

Utils.test.js

const chai = require("chai");

const expect = chai.expect;

const sinon = require("sinon");

const sinonChai = require("sinon-chai");

chai.use(sinonChai);

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

var crypto = require("crypto");

var config = require("./config");

var utils = require("./utils");

var sandbox = sinon.sandbox.create();

describe("utils", () => {

  let secretStub, digestStub, updateStub, createHashStub, hash;

  beforeEach(() => {

    secretStub = sandbox.stub(config, "secret").returns("fake\_secret");

    digestStub = sandbox.stub().returns("ABC123");

    updateStub = sandbox.stub().returns({

      digest: digestStub,

    });

    createHashStub = sandbox.stub(crypto, "createHash").returns({

      update: updateStub,

    });

    hash = utils.getHash("hashtest"); // can be any name , just below append in test result

  });

  afterEach(() => {

    sandbox.restore();

  });

  it("should return null if invalid string is passed", () => {

    sandbox.reset(); // sometimes sandbox.restore() do not work use reset

    // rejection case test

let hash2 = utils.getHash(null);

    let hash3 = utils.getHash(123);

    let hash4 = utils.getHash({ name: "bar" });

    expect(hash2).to.be.null;

    expect(hash3).to.be.null;

    expect(hash4).to.be.null;

    expect(createHashStub).to.not.have.been.called;

  });

// test to get secret key from config

  it("should get secret from config", () => {

    expect(secretStub).to.have.been.calledOnce;

  });

// test to get correct hash

  it("should call crypto with correct settings abd return hash", () => {

    expect(createHashStub).to.have.been.calledWith("md5");

    expect(updateStub).to.have.been.calledWith("hashtest\_fake\_secret");

    expect(digestStub).to.have.been.calledWith("hex");

    expect(hash).to.equal("ABC123");

  });

});

Routes

* For handeling routes in Node, we use package called supertest

npm i supertest

1. GET / Route

app.js

const express = require('express');

const bodyParser = require('body-parser');

const mongoose = require('mongoose');

const app = express();

const db = require('../config/database');

const users = require('./users');

const auth = require('./auth');

mongoose.connect(db());

app.use(bodyParser.urlencoded({

    extended: true

}));

app.use(bodyParser.json());

//--------------------------------------> routes

app.get('/', (req, res) => {

    res.status(200).json({

        name: 'Foo Fooing Bar'

    });

});

app.test.js

const chai = require("chai");

const expect = chai.expect;

const chaiAsPromised = require("chai-as-promised");

chai.use(chaiAsPromised);

const sinon = require("sinon");

const sinonChai = require("sinon-chai");

chai.use(sinonChai);

const rewire = require("rewire");

const request = require("supertest");

var app = rewire("./app");

var users = require("./users");

var auth = require("./auth");

var sandbox = sinon.sandbox.create();

describe("app", () => {

  afterEach(() => {

    app = rewire("./app");

    sandbox.restore(); // basic for working with sinon/stubs properly

  });

  context("GET /", () =>{

      it("should get /", (done) =>{

        request(app).get("/") // functionality provided by supertest

            .expect(200)

            .end((err, response) =>{

                expect(response.body).to.have.property("name").to.equal("Foo Fooing Bar");

                done(err);

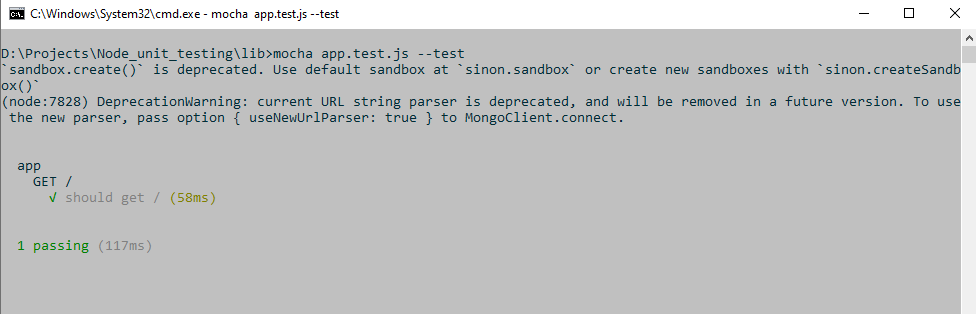
            })

      })

  });

});

Run – mocha app.test.js --test // test flag is must for stopping execution



1. POST Route

app.js

app.post('/user', function (req, res) {

    users.create(req.body).then((result) => {   // write test for this

        res.json(result);

    }).catch((err) => {

        handleError(res, err); // write test for this

    });

});

app.test.js

 ////////// Route 2 Test - POST / ///////////////////

  context("POST /user", () =>{

      let createStub, errorStub;

      it("should call user.create", (done) =>{

// think this to be returned from actual function

        createStub = sandbox.stub(users, "create").resolves({name: "foo"});

        request(app).post("/user")

            .send({name: "fake"})   // way to send data in post route

            .expect(200)

            .end((err, response) => {

                expect(createStub).to.have.been.calledOnce;

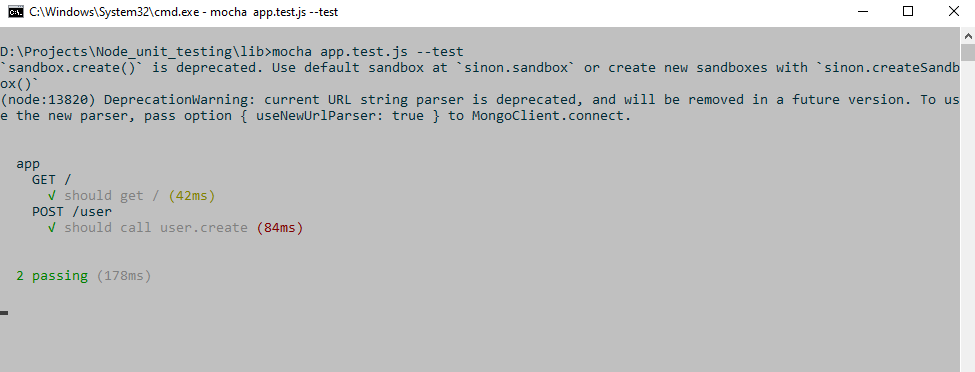
                expect(response.body).to.have.property("name").to.equal("foo"); // should match line 38

                done(err);

            })

      })

  });



Eg2

app.test.js

context("POST /user", () =>{

      let createStub, errorStub;

      it("should call user.create", (done) =>{

        createStub = sandbox.stub(users, "create").resolves({name: "foo"}); // think this to be returned from actual function

        request(app).post("/user")

            .send({name: "fake"})   // way to send data in post route - if route in app.js expected this, will not throw this err

            .expect(200)

            .end((err, response) => {

                expect(createStub).to.have.been.calledOnce;

                expect(response.body).to.have.property("name").to.equal("foo"); // should match line 38

                done(err);

            })

      })

// test for handeling err

      it("should call handleError on error", (done) =>{

        createStub = sandbox.stub(users, "create").rejects(new Error("fake error"));

        errorStub = sandbox.stub().callsFake((res, err) => {

          return res.status(400).json({error: "fake"}); // Stubbed o/p

        });

        app.\_\_set\_\_("handleError", errorStub);

        request(app).post("/user")

          .send({ name: "fake" })

          .expect(400)

          .end((error, response) => {

            expect(createStub).to.have.been.calledOnce;

            expect(errorStub).to.have.been.calledOnce;

            expect(response.body).to.have.property("error").to.equal("fake");   // must be same as line 54

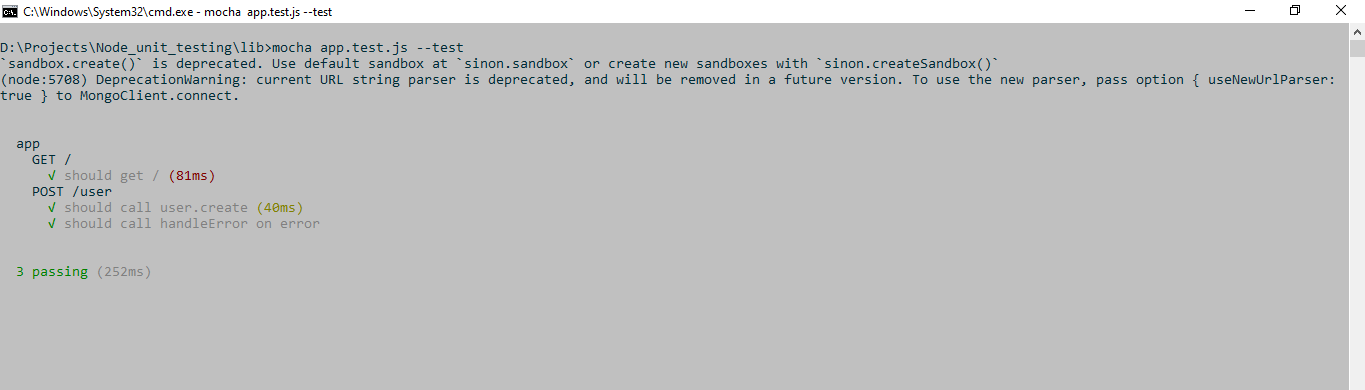
            done(error);

          });

      });

  });

mocha app.test.js --test



Eg 3

///////////// ROUTE3: Delete Route //////////////

Auth.js (middleware)

exports.isAuthorized = function (req, res, next) {

    if (req.headers.authorization === 'foo') {

        return next()

    }

    return res.json({

        error: 'Unauthorized'

    });

}

App.js (route to test)

app.delete('/user/:id', auth.isAuthorized, function (req, res) {

    users.delete({id: req.params.id, name: 'foo'}).then((result) => {

        res.json(result);

    }).catch((err) => {

        handleError(res, err);

    });

});

App.test.js

context("DELETE /user/:id", () =>{

    let authStub, deleteStub;

    beforeEach(() => {

      fakeAuth = (req, res, next) => {

        return next();

      }

      authStub = sandbox.stub(auth, "isAuthorized").callsFake(fakeAuth); // WAY TO STUB MIDDLEWARE

      app = rewire("./app");

    });

    it("should call auth check function and users.delete on success", (done) => {

      deleteStub = sandbox.stub(users, 'delete').resolves("fake\_delete");  // stubbed original and expect fake\_delete

      request(app).delete("/user/123") // Passing the ID

        .expect(200)

        .end((err, response) => {

          expect(authStub).to.have.been.calledOnce;

          expect(deleteStub).to.have.been.calledWithMatch({id: "123"}); // callWithMatch matches property in obj data

          expect(response.body).to.equal("fake\_delete");

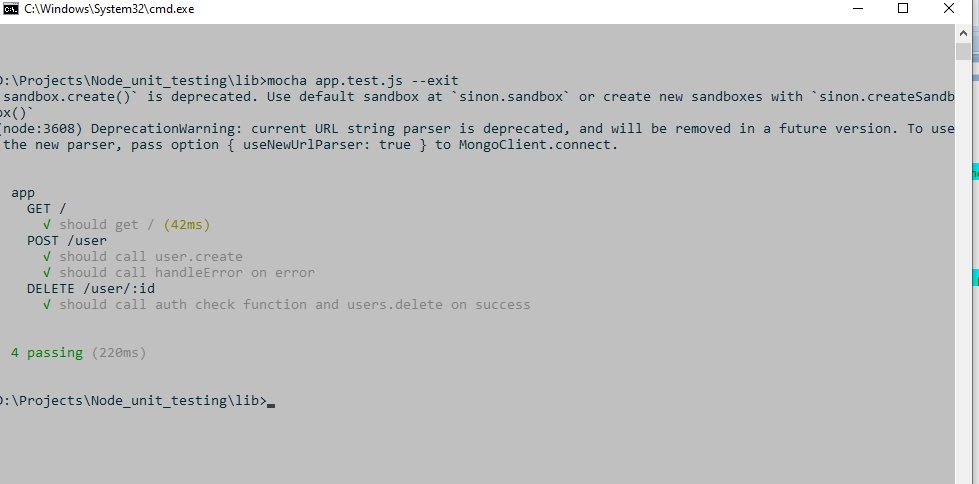
          done(err);

        })

    });

  });

Run mocha app.test.js –exit



Eg 4

app.js

function handleError(res, err) {

    if (err instanceof Error) {

        return res.status(400).json({ // Write test for this

            error: err.message

        });

    }

    return res.status(400).json(err); // Write test for this

}

app.test.js

context("handleError", () => {

    let handleError, res, statusStub, jsonStub;

    beforeEach(() => {

      jsonStub = sandbox.stub().returns("done");

      statusStub = sandbox.stub().returns({

        json: jsonStub

      });

      res = {

        status: statusStub

      }

      handleError = app.\_\_get\_\_("handleError");   // this get functionality is from rewire

    });

    it("should check error instance and format message", (done) => {

      let result = handleError(res, new Error("fake"));

      expect(statusStub).to.have.been.calledWith(400);

      expect(jsonStub).to.have.been.calledWith({

        error: "fake"

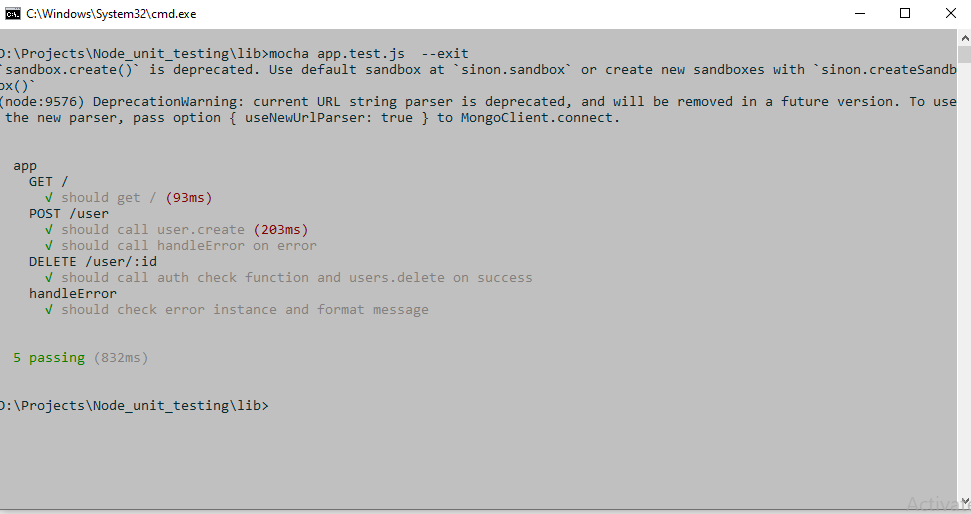
      });

      done();

    })

  })

Run: mocha app.test.js –exit



app.test.js (2nd test)

context("handleError", () => {

    let handleError, res, statusStub, jsonStub;

    beforeEach(() => {

      jsonStub = sandbox.stub().returns("done");

      statusStub = sandbox.stub().returns({

        json: jsonStub

      });

      res = {

        status: statusStub

      }

      handleError = app.\_\_get\_\_("handleError");   // this get functionality is from rewire

    });

    it("should check error instance and format message", (done) => {

      let result = handleError(res, new Error("fake"));

      expect(statusStub).to.have.been.calledWith(400);

      expect(jsonStub).to.have.been.calledWith({

        error: "fake"

      });

      done();

    })

    it("should return object without changing it if not instance of error", (done) => {

      let result = handleError(res, {id: 1, message: "fake error"});

      expect(statusStub).to.have.been.calledWith(400);

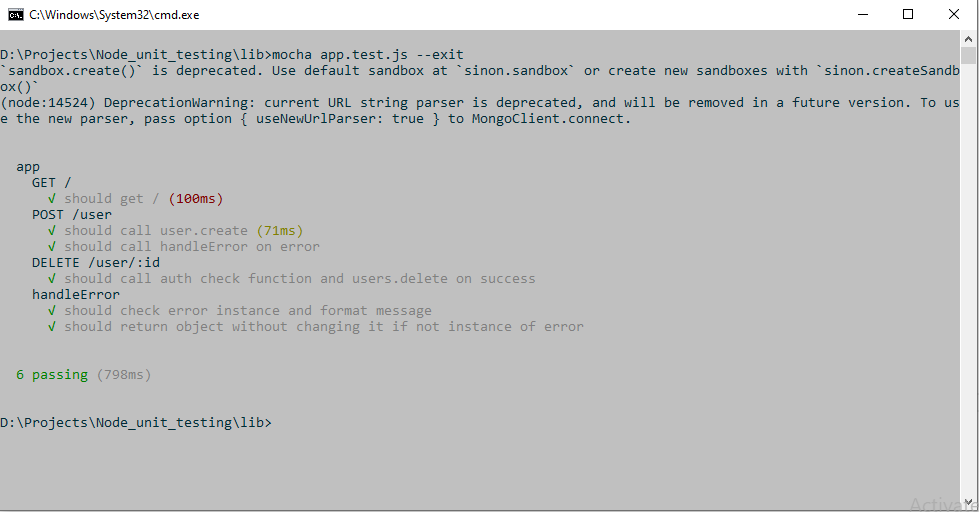
      expect(jsonStub).to.have.been.calledWith({id: 1, message: "fake error"});

      done();

    });

  })

Run: mocha app.test.js



WAY TO TEST MODEL

user.js

Path - Model/user.js

var mongoose = require('mongoose');

var UserSchema = mongoose.Schema({

    name: {type: String, required: true},

    email: {type: String, required: true},

    age: Number

}, {

    collection: 'users'

}); //overrides default collection name auto created

module.exports = mongoose.model('User', UserSchema);

user.test.js

path- model/user.test.js

const chai = require("chai");

const expect = chai.expect;

var User = require("./user"); // model included

describe("User model", () => {

    it("should return error in required areas are missing", (done) => { // when name or email field is missing

        let user = new User(); // err as no data obj passed

        user.validate((err) => {                    // this is mongodb default functions

            expect(err.errors.name).to.exist;

            expect(err.errors.email).to.exist;

            expect(err.errors.age).to.not.exist;    // optional field

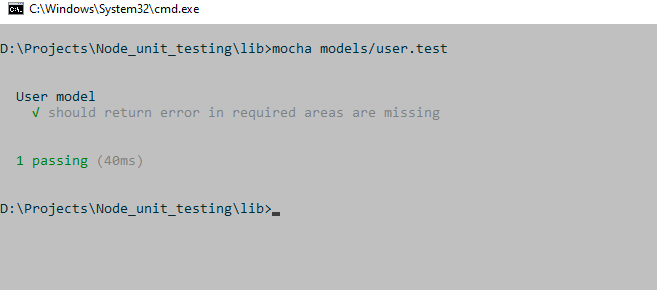
            done();

        });

    })

});

Run: mocha model/user.test.js



Eg 2: user.test.js

it("should have optional age field", (done) => {

        let user = new User({

            name: "foo",

            email: "foo@gmail.com",

            age: 27

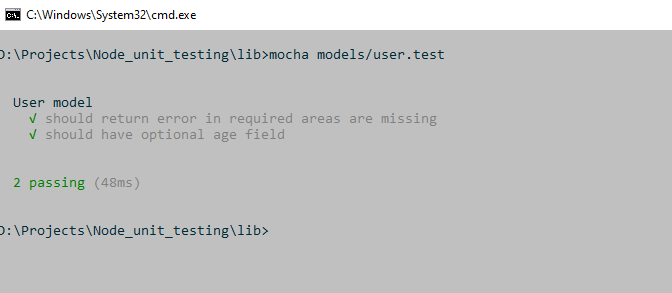
        });

        expect(user).to.have.property("age").to.equal(27);

        done();

});

Run: mocha model/user.test.js



WAY TO TEST CLASSES

order.js

class Order {

    constructor(ref, user, items) {

        this.ref = ref;

        this.user = user;

        this.items = items;

        this.status = 'Pending';

        this.createdAt = Date.now();

        this.updatedAt = Date.now();

        this.subtotal = 0;

        for (let item of items) {

            // console.log(item)

            this.subtotal += item.price

        }

        if (this.subtotal <= 50) {

            this.shipping = 5;

        } else {

            this.shipping = 10;

        }

        this.total = this.subtotal + this.shipping;

    }

    save() {

        //..some logic..

        this.status = 'Active';

        this.updatedAt = Date.now();

        let o = {

            ref: this.ref,

            user: this.user.name,

            updatedAt: this.updatedAt,

            status: this.status,

            items: this.items,

            shipping: this.shipping,

            total: this.total,

        }

        return o;

    }

    cancel() {

        //...some logic...

        this.status = 'Cancelled';

        this.updatedAt = Date.now();

        this.shipping = 0;

        this.total = 0;

        console.warn('Order cancelled');

        return true;

    }

}

Order.prototype.ship = function () {

    this.status = 'Shipped';

    this.updatedAt = Date.now();

}

module.exports = Order;

order.test.js

const chai = require("chai");

const expect = chai.expect;

const sinon = require("sinon");

const sinonChai = require("sinon-chai");

chai.use(sinonChai);

var Order = require("./order");

var sandbox = sinon.sandbox.create();

describe("order", () => {

    let warnStub, dateSpy, user, items, o;

    beforeEach(() => {

        warnStub = sandbox.stub(console, "warn");

        dateSpy = sandbox.spy(Date, 'now');

        user = { id: 1, name: "foo" };

        items = [

            { name: "Book", price: 10 },

            { name: "Dice set", price: 5 }

        ];

        o = new Order(123, user, items);

    });

    afterEach(() => {

        sandbox.restore();

    });

    it("should create instance or Order and calculate total + shipping", () => {

        expect(o).to.be.instanceOf(Order);

        expect(dateSpy).to.have.been.calledTwice;

        expect(o).to.have.property("ref").to.equal(123);// these tests are written as per

        expect(o).to.have.property("user").to.deep.equal(user); // Way to check property

        expect(o).to.have.property("items").to.deep.equal(items);

        expect(o).to.have.property("status").to.equal("Pending");

        expect(o).to.have.property("createdAt").to.be.a("Number");

        expect(o).to.have.property("updatedAt").to.be.a("Number");

        expect(o).to.have.property("subtotal").to.equal(15);

        expect(o).to.have.property("shipping").to.equal(5);

        expect(o).to.have.property("total").to.equal(20);

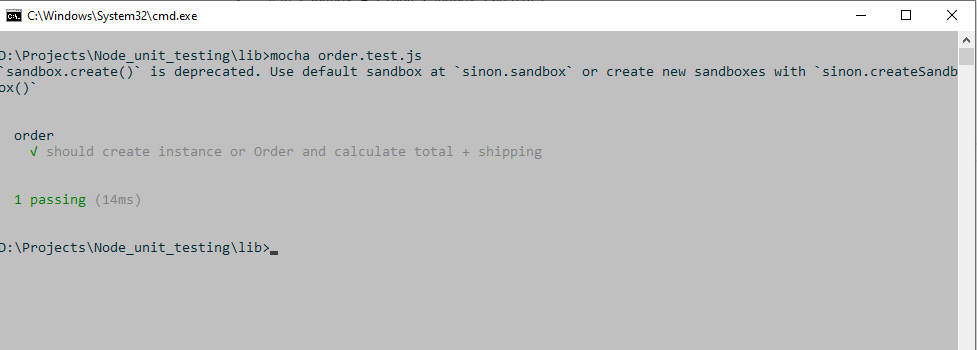
        expect(o.save).to.be.a("function");

        expect(o.cancel).to.be.a("function");

        expect(o.ship).to.be.a("function");

    })

});



Eg2:

Order.js

save() {

        //..some logic..

        this.status = 'Active';

        this.updatedAt = Date.now();

        let o = {

            ref: this.ref,

            user: this.user.name,

            updatedAt: this.updatedAt,

            status: this.status,

            items: this.items,

            shipping: this.shipping,

            total: this.total,

        }

        return o;

    }

order.test.js

it("should update the status to 'Active' and return order details", () =>{

        let result = o.save();

        expect(dateSpy).to.have.been.calledThrice;  // thrice because, it is already called twice in constructor

        expect(o.status).to.equal("Active");

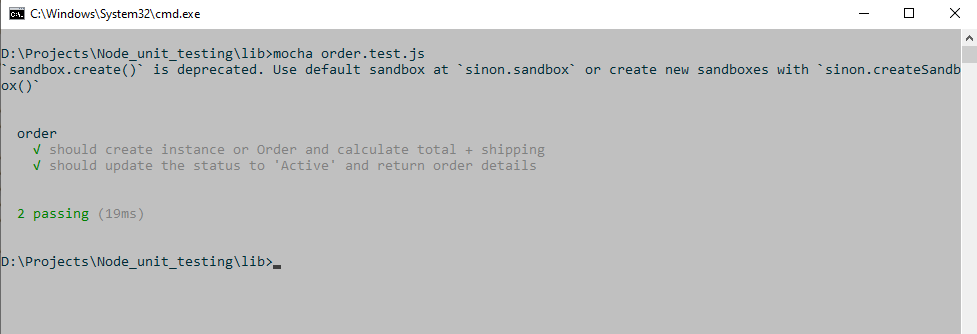
        expect(result).to.be.a("object");

        expect(result).to.have.property("user").to.equal("foo");

        expect(result).to.have.property("updatedAt").to.be.a("Number")

        // check rest of the props

})



Eg3:

Order.js

cancel() {

        //...some logic...

        this.status = 'Cancelled';

        this.updatedAt = Date.now();

        this.shipping = 0;

        this.total = 0;

        console.warn('Order cancelled');

        return true;

    }

order.test.js

it("should cancel an order, updated status and set shipping and total to zero", () => {

        let result = o.cancel();

        expect(warnStub).to.have.been.calledWith("Order cancelled");

        expect(dateSpy).to.have.been.calledThrice;

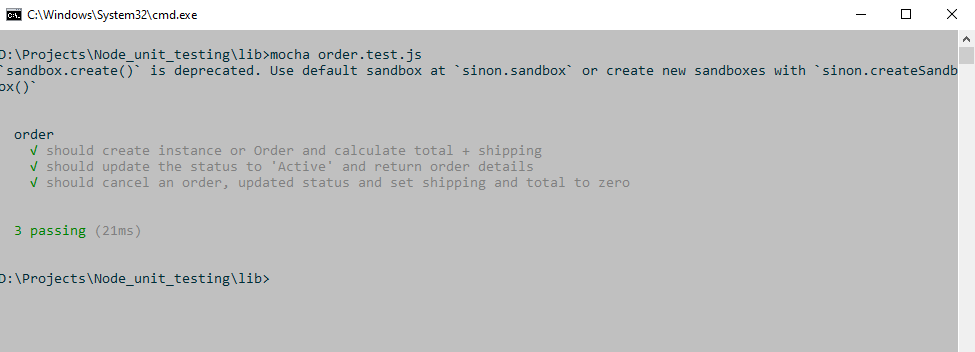
        expect(o.status).to.equal("Cancelled");

        expect(result).to.be.true;

        expect(o.shipping).to.equal(0);

        expect(o.total).to.equal(0);

})



Eg4 – prototype method

Order.js

Order.prototype.ship = function () {

    this.status = 'Shipped';

    this.updatedAt = Date.now();

}

Order.test.js

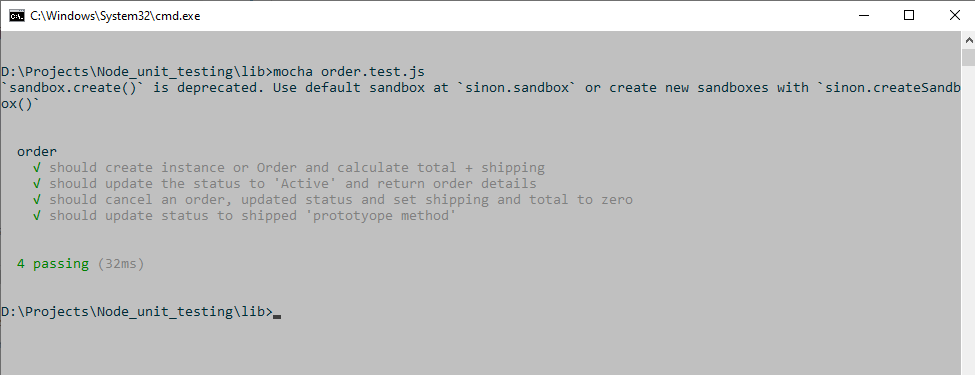
it("should update status to shipped 'prototyope method'", () => {

        o.ship();

        expect(o.status).is.equal("Shipped");

        expect(dateSpy).to.have.been.calledThrice;

    })



* So now we know to how to unit testing for class works, so if anybody changes class, unit testing will fail and we know before hand.