**TESTING USING MOCHA**

1. **MOCHA**

Mocha is a JavaScript testing framework, running on Node JS and is used for testing asynchronous code. It has a number of features like accurate reporting, detecting code leaks, support for promises and callbacks, highlighting test execution time, etc.

Mocha framework can be used for Unit, Functional and Integration Testing.

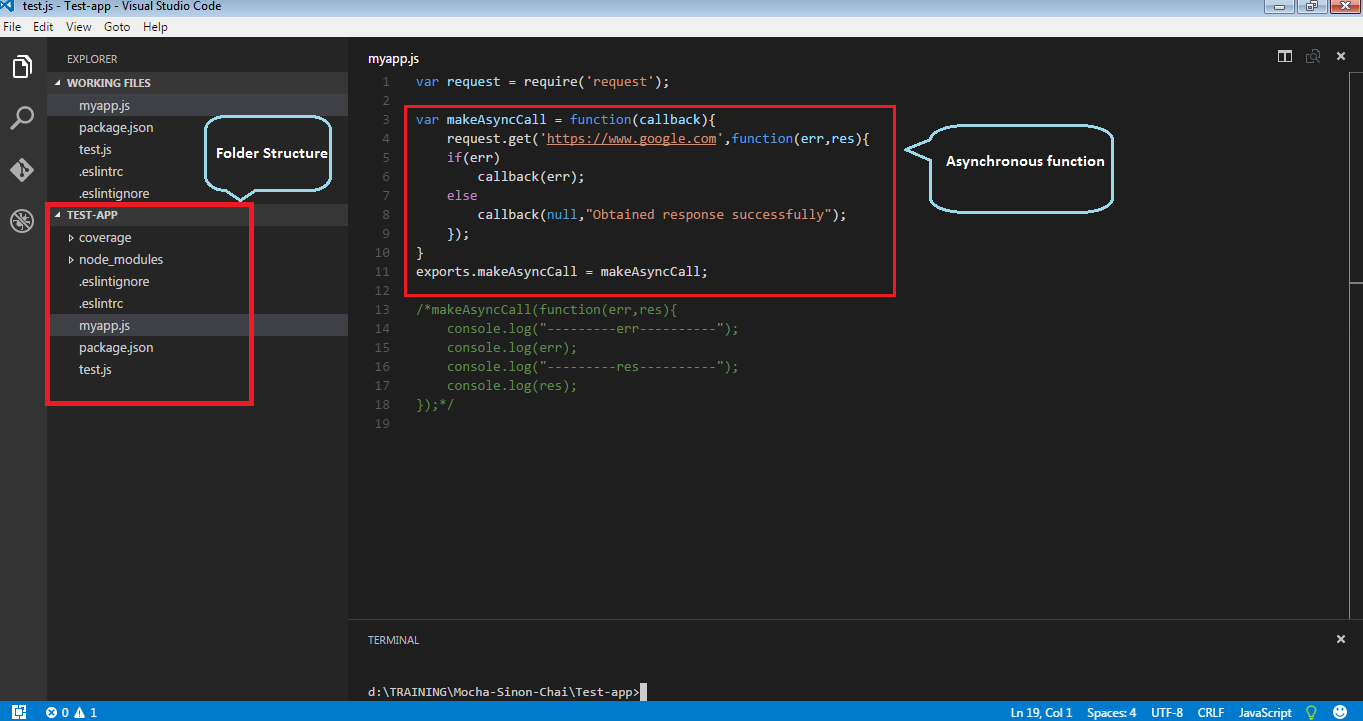
**Installation:**

npm install mocha

To save the dependency in package.json

npm install –save-dev mocha

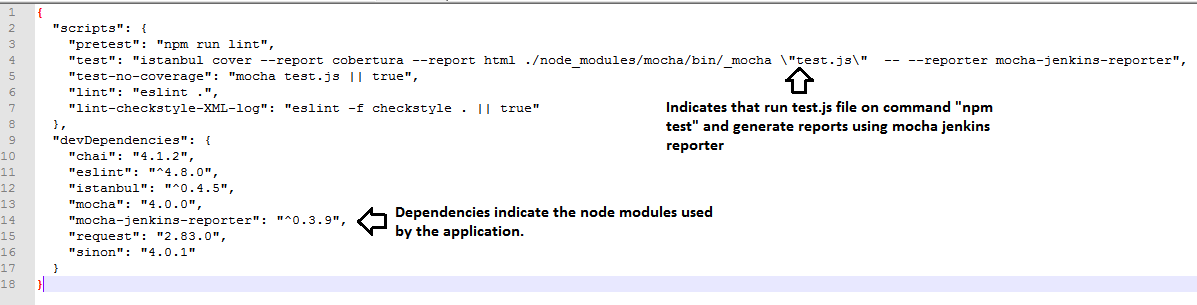
Consider the following example code for which we would be writing a test case:



The folder contains package.json, .eslintrc, .eslintignore files.

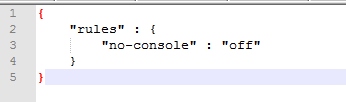
**Package.json**:

The package.json file contains information about the node modules that the application uses and also the scripts that would run on certain commands.



**.eslintrc:**

This file specifies the linter rules that help in writing syntactically correct code. An empty “rules”:{} object indicates apply all the linter rules on the files. To stop a particular rule, use “off”.



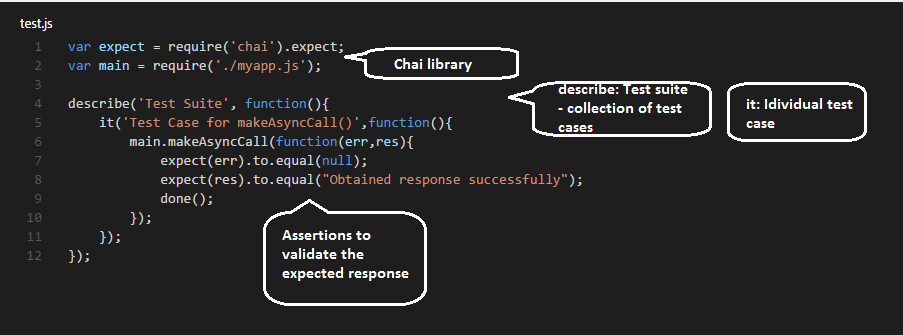
**.eslintignore:**

The folders mentioned in this file are skipped by eslint.

**Istanbul:**

Having tests is all well and good, however knowing how much of your codebase is covered by your tests is even better. Istanbul does a great job of providing this information, it reports on line coverage, statement coverage, function coverage and the most important metric, branch coverage.

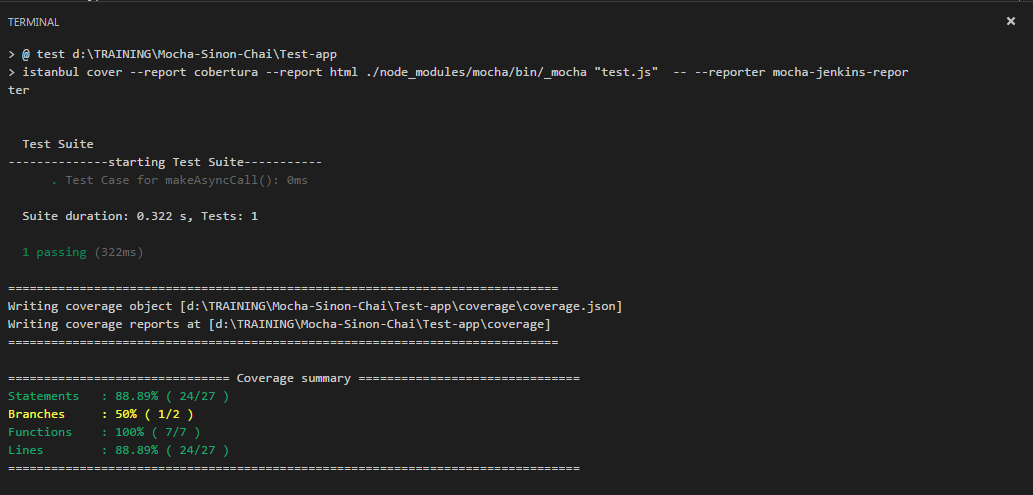
**Test Cases using Mocha:**



As seen in the above example, we need “Assertion libraries” to validate the output of the code under testing.

The above test file will be executed with the command – **“npm test”** or **“mocha test.js”**

**Result:**



**Note:** If the test files are in the same folder as the files under testing, .test.js files will also be included in test coverage. So, it is a good practice to keep test files in a separate folder.

1. **CHAI Assertion Library:**

Assertion libraries are used to validate the output of the code under testing. They complement mocha and give us the ability to string together very natural sounding function chains.

e.g: foo.should.be.a(‘string’) , expect(err).to.not.equal(null)

Chai provides different assertion libraries like:

1. Should
2. Expect
3. Assert

More details: <http://chaijs.com/>

1. **Using SINON Stub & Mock**

There might be cases when you want to test the code behavior for a specific output, like error conditions. In this case, you might want to force/mock the behavior of a certain call. This is typically used in Unit Testing where we only want to test “what we have written” and not any other dependent module. In this case, we can mock the dependent module.

**Installing sinon:**

npm install –save-dev sinon

**Stubs:**

Stubs are functions with pre-programmed behavior.

Use stubs when:

1. Control a method’s behavior from a test to force the code down a specific path. Examples include forcing a method to throw an error in order to test error handling.
2. When you want to prevent a specific method from being called directly (possibly because it triggers undesired behavior, such as a XMLHttpRequest or similar).

Detailed explanation here: <http://sinonjs.org/releases/v4.0.1/stubs/>

**Mockery:**

Mocks are fake methods along with pre-programmed expectations. The rule of thumb is: if you wouldn’t add an assertion for some specific call, don’t mock it. Use a stub instead.

More details: <http://sinonjs.org/releases/v4.0.1/mocks/>

Mockery is used along with stub to force the behavior of a module.

<http://bulkan-evcimen.com/using_mockery_to_mock_modules_nodejs.html>

**Example:**

var mockery = require('mockery');

mockery.enable({

warnOnReplace: false,

warnOnUnregistered: false,

useCleanCache: true

});

var stub = sinon.stub(); //create an anonymous stub

mockery.registerMock('request', stub); //bind it to request module

it('Test Case for makeAsyncCall()', function () {

stub.onCall(count++).yields({ //provide fake response

"statusCode": 404,

"body": {

"message": "Resource not found"

}

}, null);

main.makeAsyncCall(function (err, res) {

expect(err).to.not.equal(null);

expect(err.statusCode).to.equal(404);

done();

});

});

Here, by mocking request module we mean, use my implementation of “request” and not the original one.

**Difference between stub and mock:**

Stub will always provide static response no matter what input is given. Mock will have pre-programmed expectations for every input.

<https://stackoverflow.com/questions/3459287/whats-the-difference-between-a-mock-stub>

1. **Hooks:**

Mocha also provides hooks for performing some operations, like initializations before the test suite or test case begins. Similarly, clean up tasks can be performed after test case execution.

Hooks provided by mocha are: before(), after(), beforeEach(), afterEach()

Example:

describe('Test Suite', function () {

before(function(){ //executes before Test Suite begins

count = 0;

console.log("--------------starting Test Suite-----------");

});

after(function(){ //executes after Test Suite is over

mockery.disable();

});

**Selectively running test cases:**

1. .only – will run only that single test case or test suite

Example: it.only('Test Case for makeAsyncCall()', function () {

1. .skip – will skip this test case or test suite

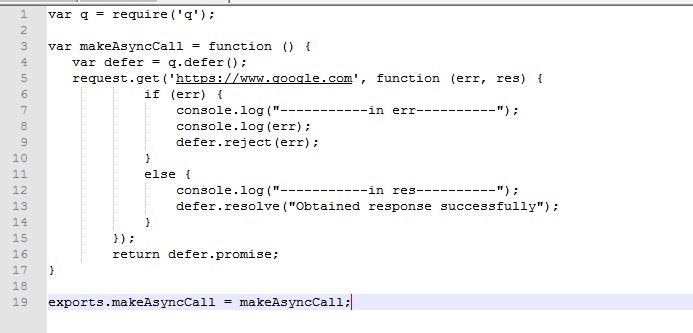
Example: it.skip('Test Case for makeAsyncCall()', function () {

1. **Working with Promise**

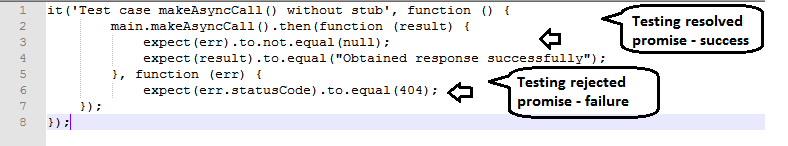
Promises are used when asynchornous tasks are to be performed in series. If there are dependent tasks t1, t2, then t2 will begin only after t1 has returned a promise. If t1 returns a promise with error, t2 is not triggered and the error is propagated. Instead of having multiple nested callbacks for such tasks, promises is a good way to avoid the complexity.

**More details:** <http://2ality.com/2014/10/es6-promises-api.html>

Consider the code below:



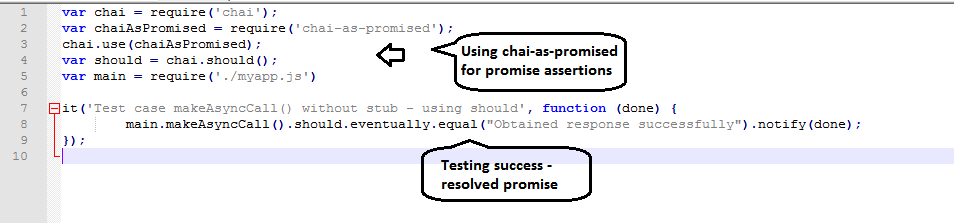
To test the above method we need to write a test case using the promise syntax.



Another decent way of writing the above test case is by using the **“chai-as-promised”** node module.

Chai as Promised extends Chai with a fluent language for asserting facts about promises.chai promised.

Hence our test case will be:



**Refer:** <https://www.sitepoint.com/promises-in-javascript-unit-tests-the-definitive-guide/>

The above test cases are functional test cases and call all the dependencies from the method. To isolate our method, we need to mock the dependencies by providing fake output. Here, the HTTP request using the “request” module is a dependency. Hence we must mock it while Unit Testing.

For mocking this request, **“nock”** can be used.

**Refer:** <https://www.npmjs.com/package/nock>

