CITY UNIVERSITY
LONDON
EST 1894

Unit Testing your JavaScript

Aris Markogiannakis
City University - Short Courses
CS3606 : JavaScript 2: Advanced Javascript for Websites and Web

Today

- Unit Tests?
- Examples
- Create our own Unit Tests using libraries such as Mocha, Chai, Sinon

Have you been there?

Spend a day trying to find
what was changed on my
code

<u>How could I have avoided this?</u>
If you had written some unit
tests you should have been able
to find what was changed, as unit
test suppose to check was what the
expected

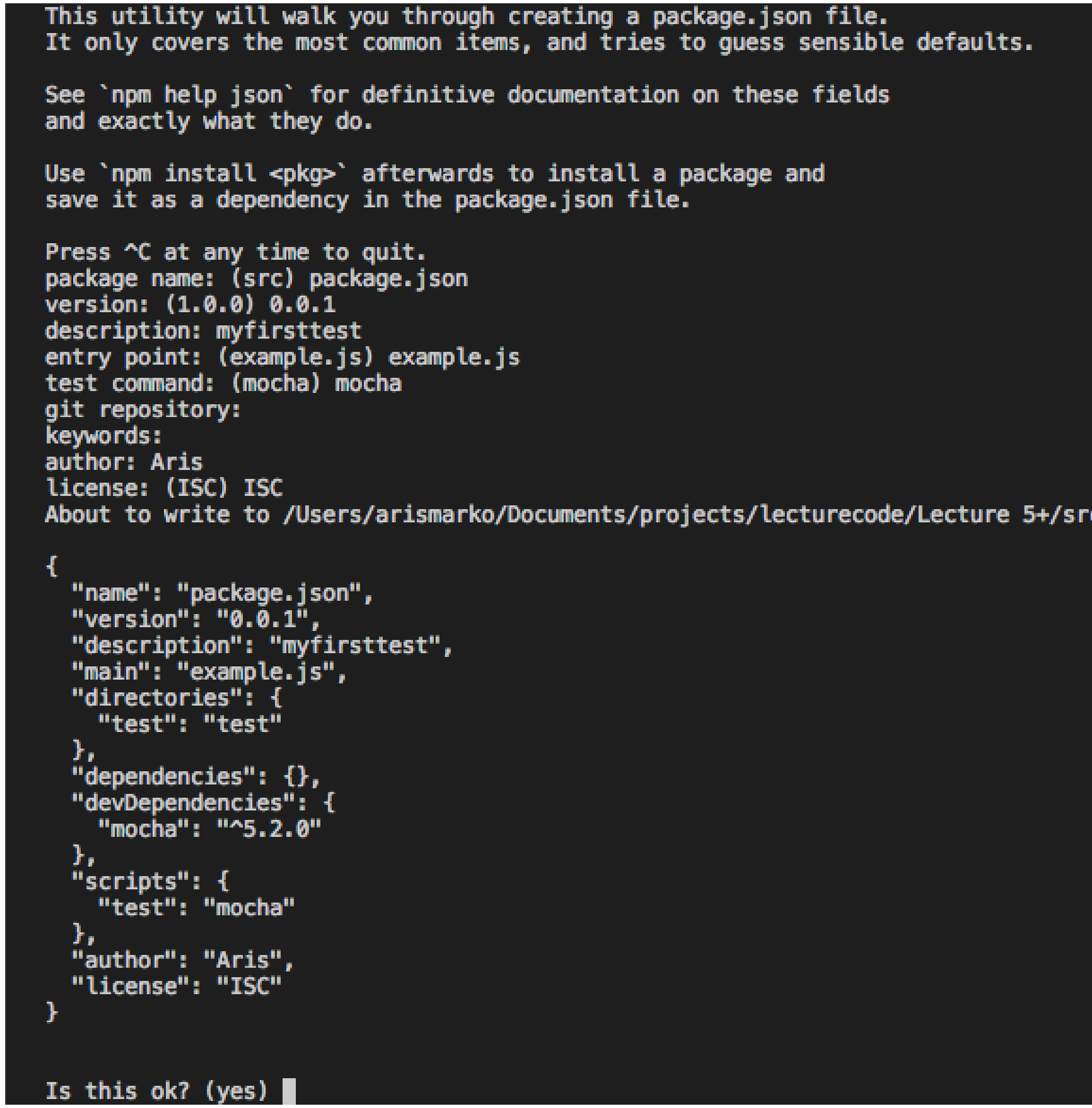
Testing frameworks, libraries and utilities

- Mocha (mochajs.org) OS framework runs on top of node.js
  - It uses Assertion libraries

  We first need to make sure we have node on our machine
  - node –v to get the current version of node
  - npm install mocha --save-dev
  - Create a test folder and then inside a test.js file

Get Started

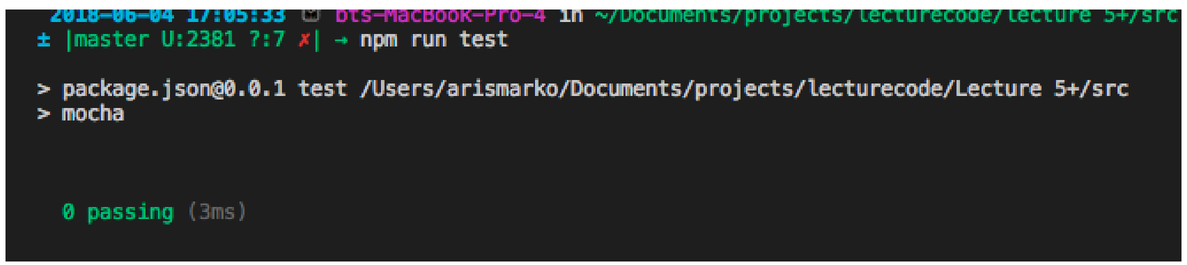
- To get started we need to setup our environment
  - Create a folder called myfirst-js-tdd
  - Inside the folder lets first setup our environment
    - **run npm init** - you should be able to see my example on the right.
  - Then lets install mocha
    - npm install mocha - - save - dev



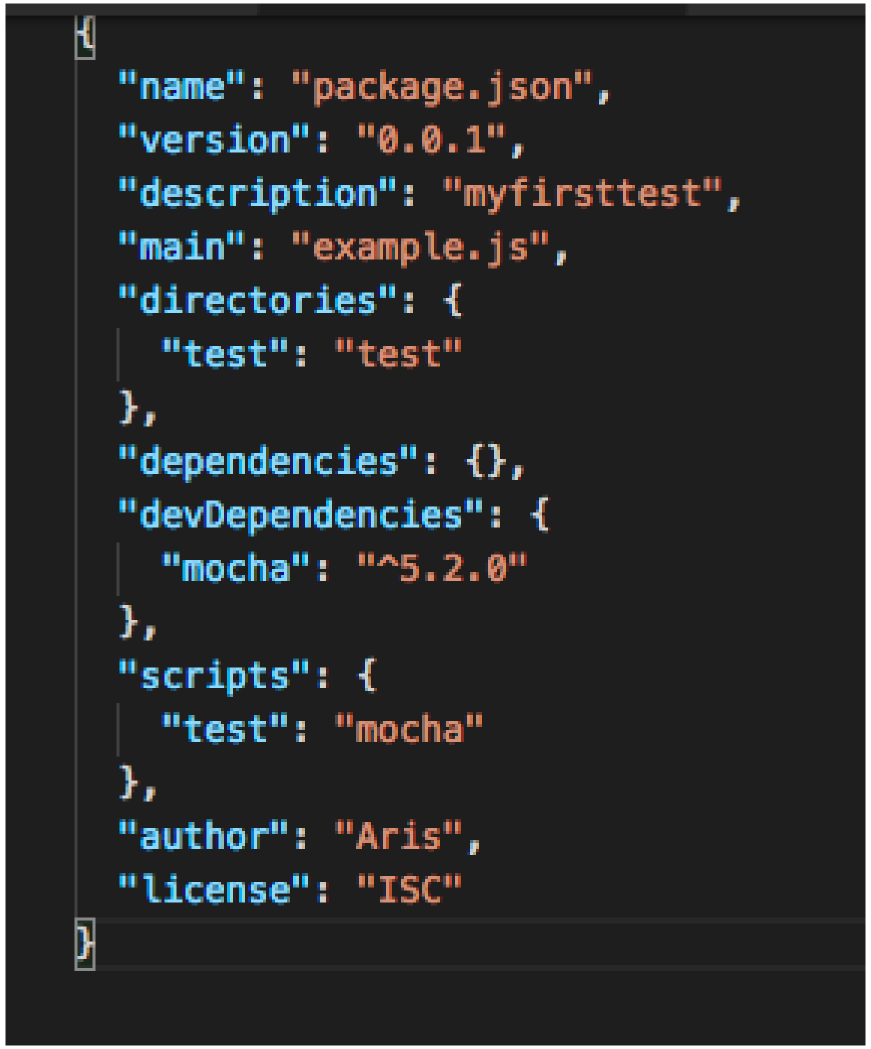
```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (src) package.json
version: (1.0.0) 0.0.1
description: myfirsttest
entry point: (example.js) example.js
test command: (mocha) mocha
git repository:
keywords:
author: Aris
license: (ISC) ISC
About to write to /Users/arismarko/Documents/projects/lecturecode/Lecture 5+/sr

{
  "name": "package.json",
  "version": "0.0.1",
  "description": "myfirsttest",
  "main": "example.js",
  "directories": {
    "test": "test"
  },
  "dependencies": {},
  "devDependencies": {
    "mocha": "^5.2.0"
  },
  "scripts": {
    "test": "mocha"
  },
  "author": "Aris",
  "license": "ISC"
}


Is this ok? (yes)
```

Get Started

- Run our test for the first time
  - Create a folder called test and inside add a file called tests.js
  - You can run your tests by
    - **npm run test**
- You will see the following:



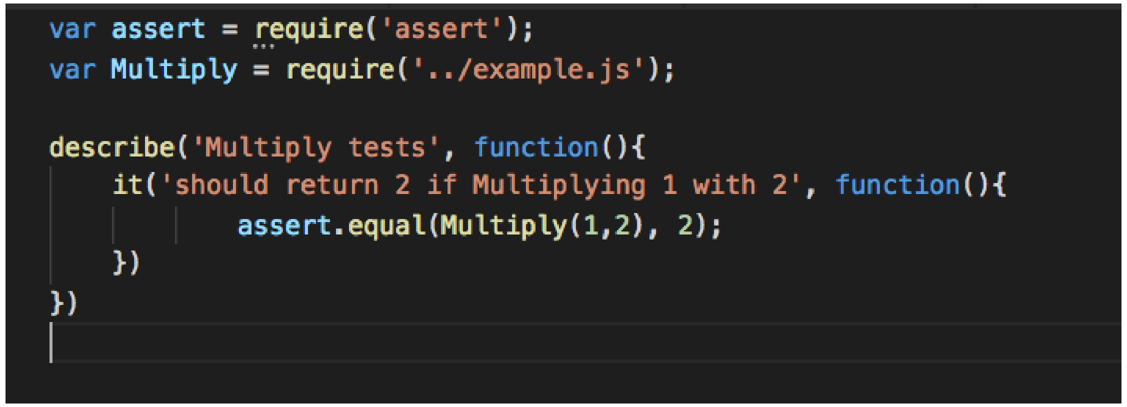
```
 2018-06-04 17:05:33 ☐ bts-MacBook-Pro-4 in ~/Documents/projects/lecturecode/lecture 5+/src
± |master U:2381 ?:7 ✗| → npm run test

> package.json@0.0.1 test /Users/arismarko/Documents/projects/lecturecode/Lecture 5+/src
> mocha



  0 passing (3ms)
```



```
{
  "name": "package.json",
  "version": "0.0.1",
  "description": "myfirsttest",
  "main": "example.js",
  "directories": {
    "test": "test"
  },
  "dependencies": {},
  "devDependencies": {
    "mocha": "^5.2.0"
  },
  "scripts": {
    "test": "mocha"
  },
  "author": "Aris",
  "license": "ISC"
}
```

Describe

- Describe is a testing suite
  - It takes two arguments:
    - a string "name of the suite",
    - and a function,
      - takes a function called it which also takes two arguments,
        - what the code should do,
        - and a function where **we write our tests**
- A simple test can be seen on the right
  - We use assert to check that a condition is right.

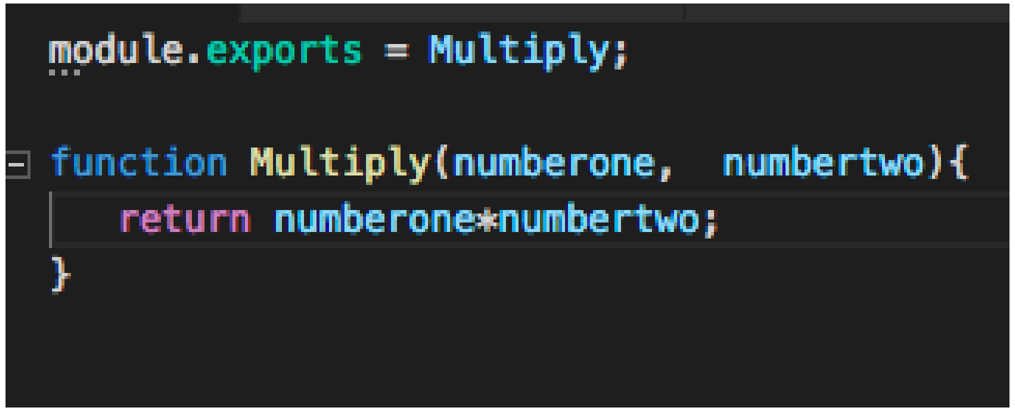


```
var assert = require('assert');
var Multiply = require('../example.js');

describe('Multiply tests', function(){
    it('should return 2 if Multiplying 1 with 2', function(){
            assert.equal(Multiply(1,2), 2);
    })
})
```

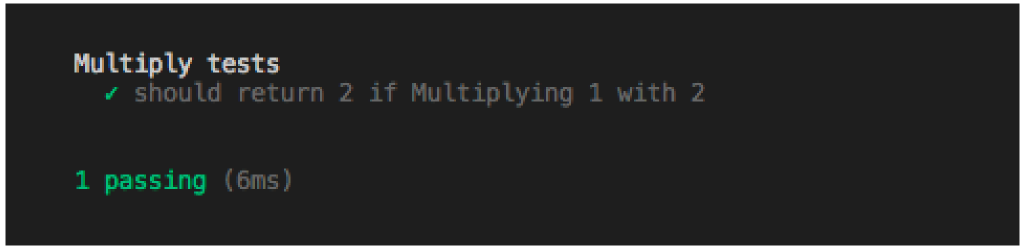
Checking against our code

- So if we write some code

- So now if we try to change our code we should always be able to see if something went wrong.



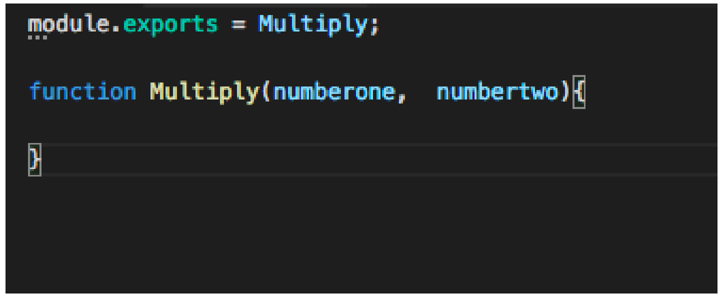
```
module.exports = Multiply;

function Multiply(numberone,  numbertwo){
    return numberone*numbertwo;
}
```



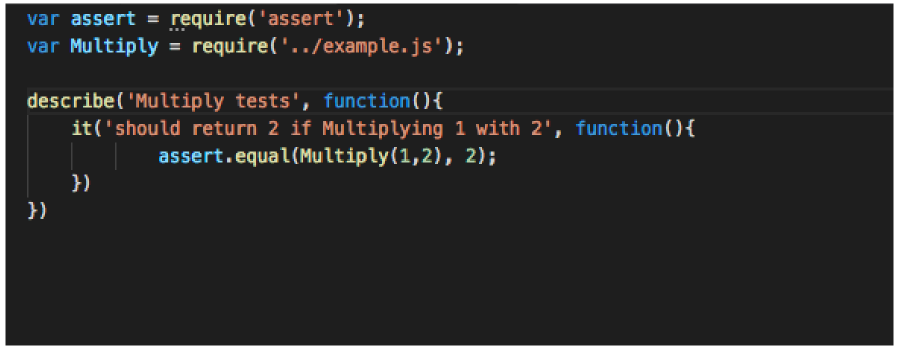
```
  Multiply tests
    ✓ should return 2 if Multiplying 1 with 2


  1 passing (6ms)
```

Checking against our code

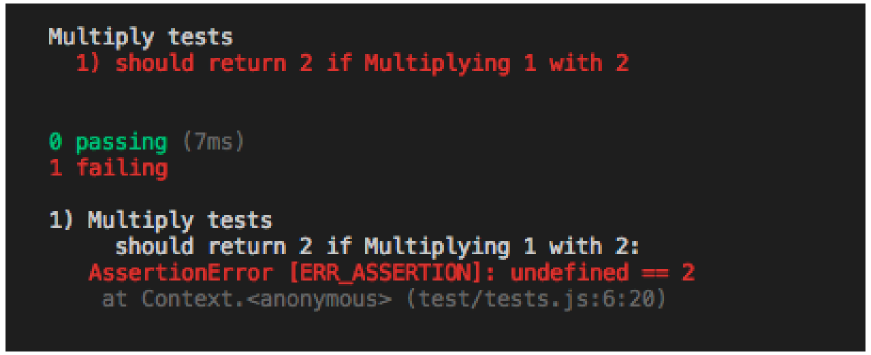
- Now is time to check against our code
  - Imagine we will need to write a code that will implement a multiplication
  - We will write first our test
    - As you can see we are importing our Multiply function
    - Then inside the code we are using the Multiply function together with the assert method to check if the result is correct.
  - If we try to run it we will get the following output
    - We get 0 passing
    - And 1 failing - **you know why?**



```
module.exports = Multiply;

function Multiply(numberone,  numbertwo){

}
```



```
var assert = require('assert');
var Multiply = require('../example.js');

describe('Multiply tests', function(){
    it('should return 2 if Multiplying 1 with 2', function(){
        assert.equal(Multiply(1,2), 2);
    })
})
```



```
  Multiply tests
    1) should return 2 if Multiplying 1 with 2


  0 passing (7ms)
  1 failing

  1) Multiply tests
       should return 2 if Multiplying 1 with 2:
     AssertionError [ERR_ASSERTION]: undefined == 2
      at Context.<anonymous> (test/tests.js:6:20)
```

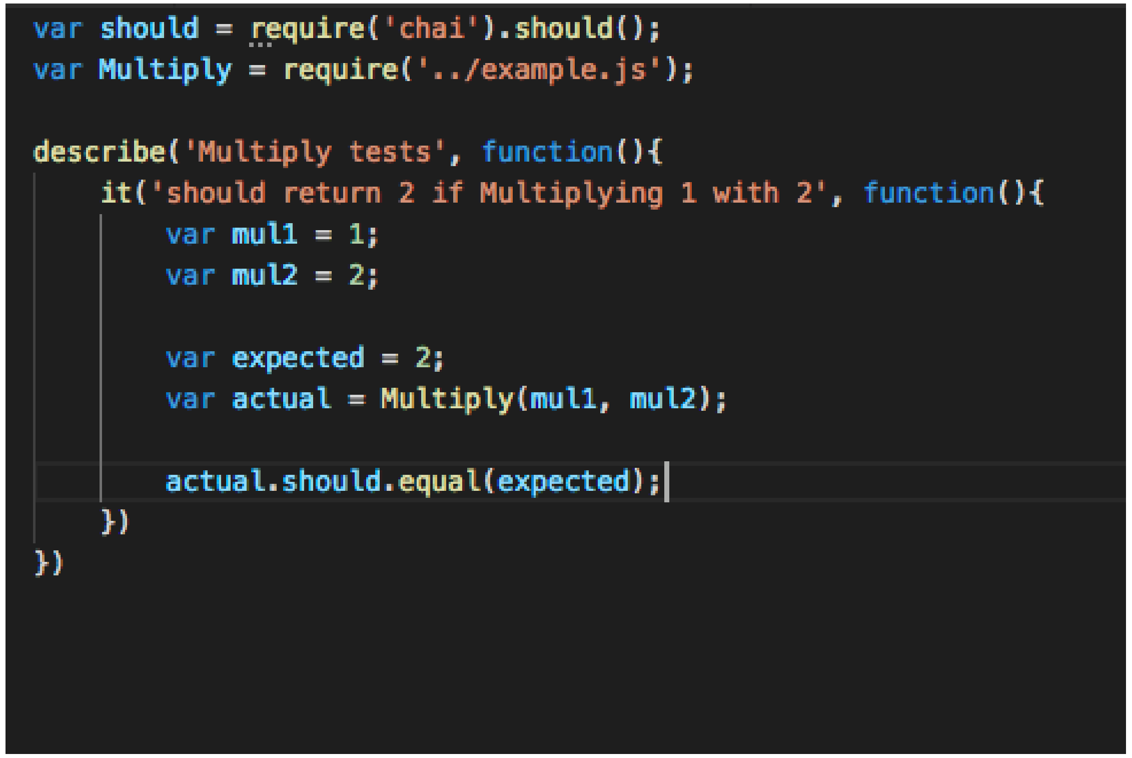
Chai

- You can read more about Chai on the website Chai.js
- Mocha is a testing framework, and Chai is an assertion library.
- We can use Chai with Mocha

- Prepare our environment
  - For this we need to install
    - npm install chai - - save - dev

Should

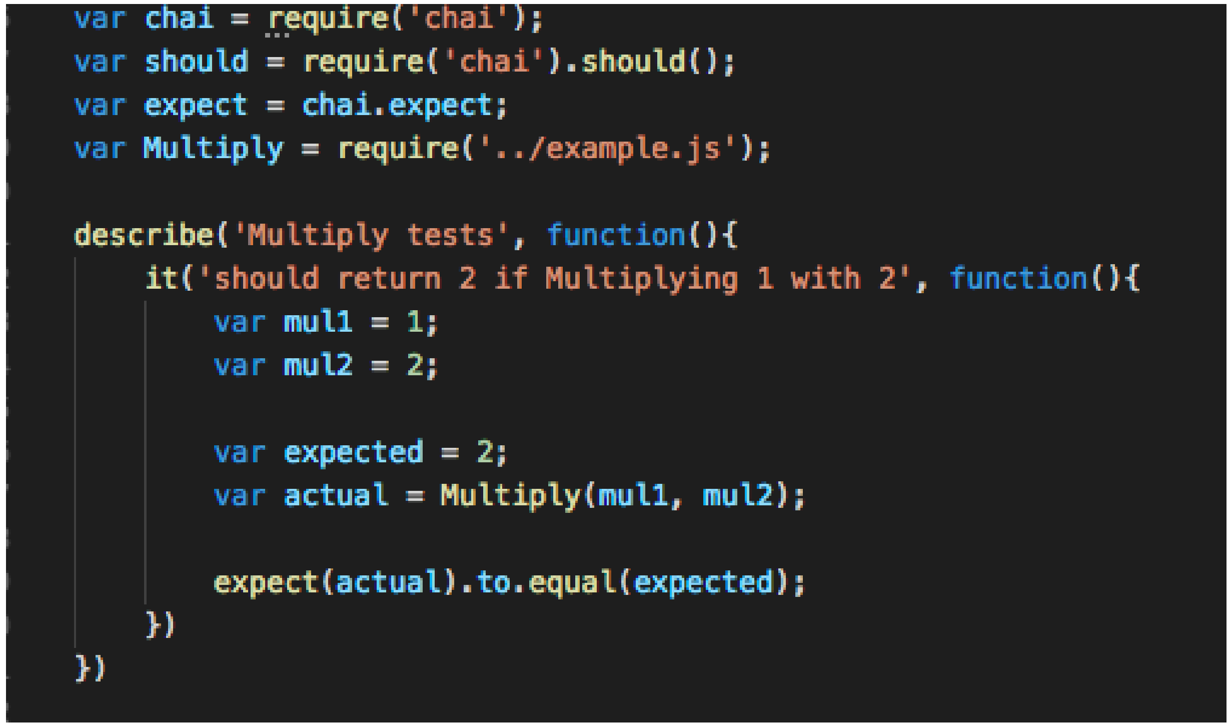
- So our first try we will import chai and the method should
- Then in our code we have two variables mul1, and mul2, and then what is expected and what is the actual from our method.
- Then we use our should
- actual.should.equal(expected)
- We can also use it as
  - actual.should.not.equal



```
var should = require('chai').should();
var Multiply = require('../example.js');

describe('Multiply tests', function(){
    it('should return 2 if Multiplying 1 with 2', function(){
        var mul1 = 1;
        var mul2 = 2;

        var expected = 2;
        var actual = Multiply(mul1, mul2);

        actual.should.equal(expected);
    })
})
```

Expect

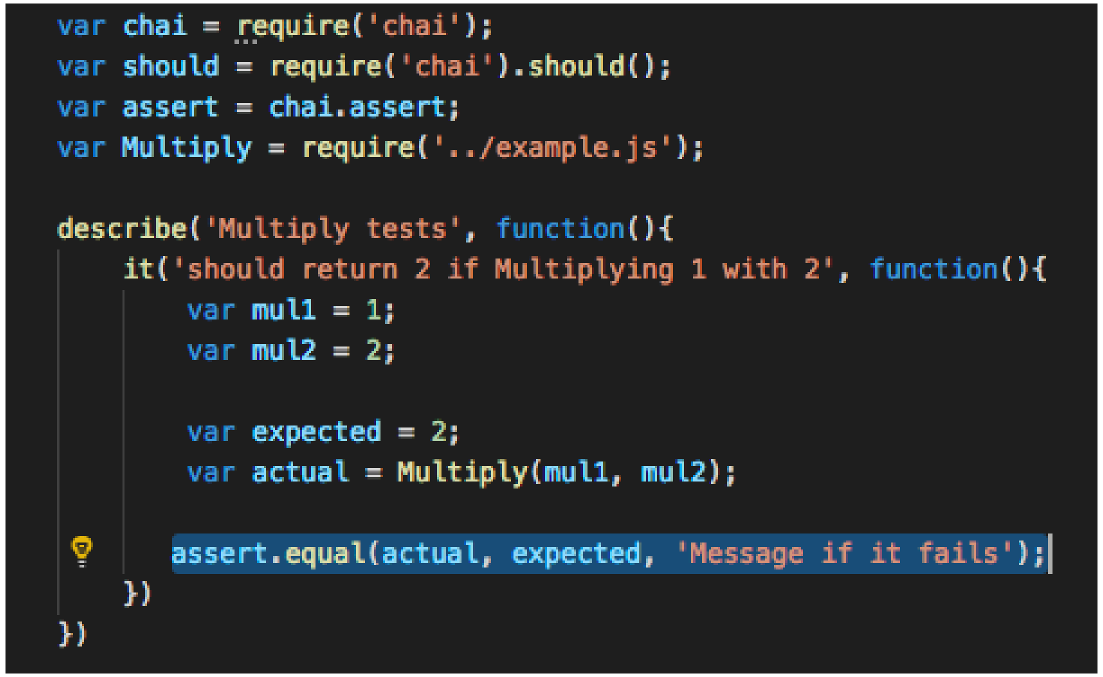
- The expect is a bit different



```
var chai = require('chai');
var should = require('chai').should();
var expect = chai.expect;
var Multiply = require('../example.js');

describe('Multiply tests', function(){
    it('should return 2 if Multiplying 1 with 2', function(){
        var mul1 = 1;
        var mul2 = 2;

        var expected = 2;
        var actual = Multiply(mul1, mul2);

        expect(actual).to.equal(expected);
    })
})
```

Assert

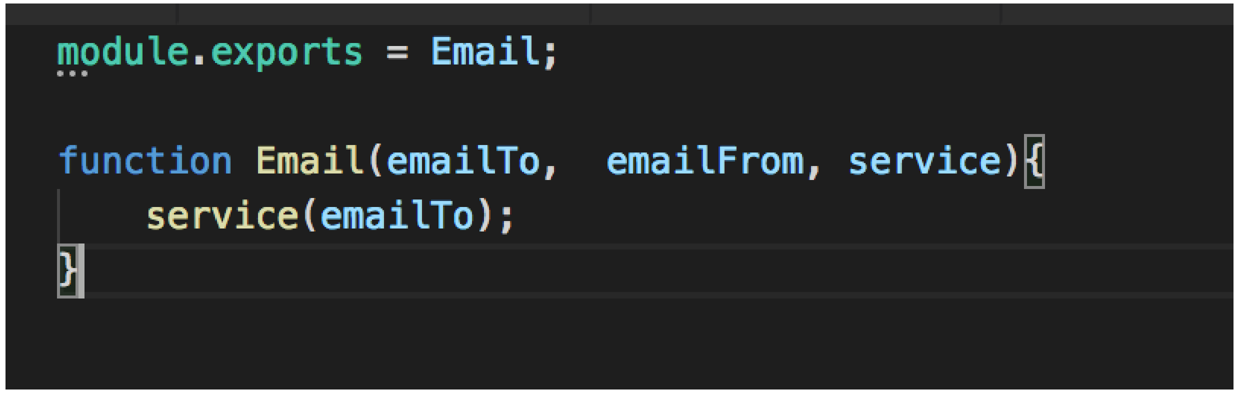
- You can also use
  - assert.equal(actual, expected, 'Message if it fails');
  - assert.not.equal(actual, expected, 'Message if it fails');



```
var chai = require('chai');
var should = require('chai').should();
var assert = chai.assert;
var Multiply = require('../example.js');

describe('Multiply tests', function(){
    it('should return 2 if Multiplying 1 with 2', function(){
        var mul1 = 1;
        var mul2 = 2;

        var expected = 2;
        var actual = Multiply(mul1, mul2);

        assert.equal(actual, expected, 'Message if it fails');
    })
})
```

Sinon

- Another JavaScript testing library
- Its purpose is to help us to write tests if you are trying to access for some more advanced code like API's
- e.g. You can mock a server for example when you enter the details for
- Imaging the function on the right an email service where there is a function that will send an email to and email from

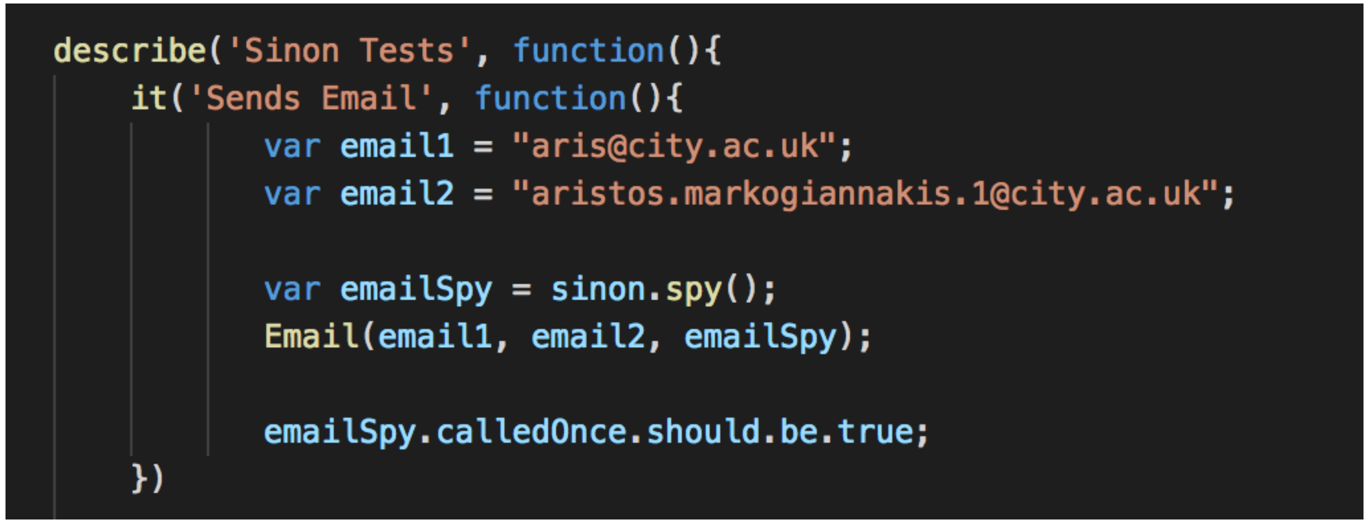


```
module.exports = Email;

function Email(emailTo,  emailFrom, service){
    service(emailTo);
}
```

Spies

- With spies we can test the function that we call, and our test will try to see at the end of the code the service function was called twice

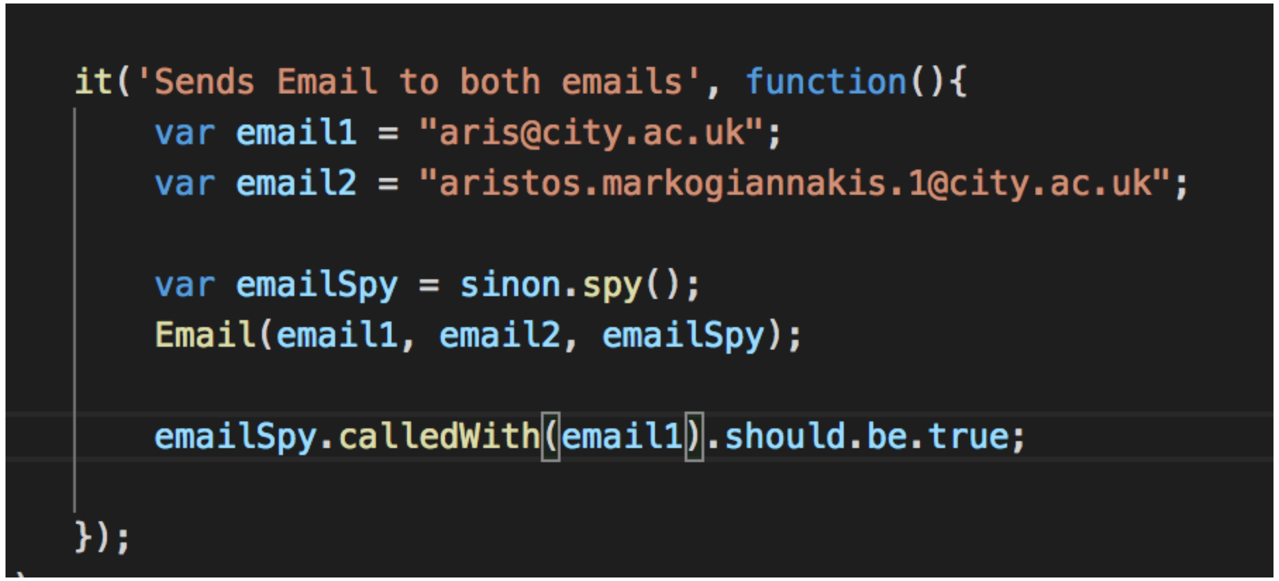
- Try to comment out the line where we execute the service method.



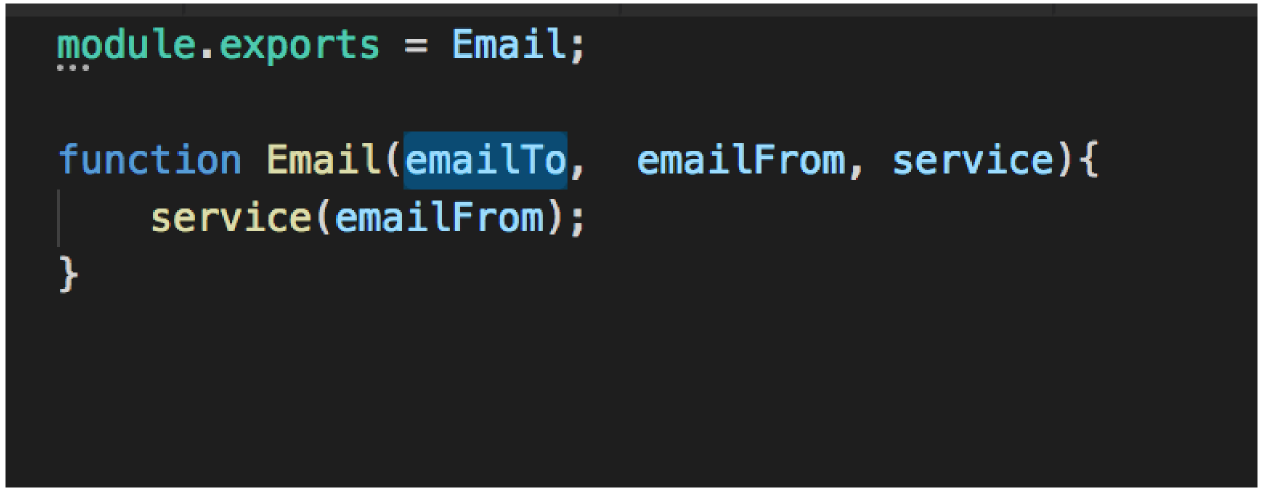
```
describe('Sinon Tests', function(){
    it('Sends Email', function(){
            var email1 = "aris@city.ac.uk";
            var email2 = "aristos.markogiannakis.1@city.ac.uk";

            var emailSpy = sinon.spy();
            Email(email1, email2, emailSpy);

            emailSpy.calledOnce.should.be.true;
    })
```

Spies

- We can also test that the spy method has been called with a certain argument
- What if we were going to change and this time send the email to email2?



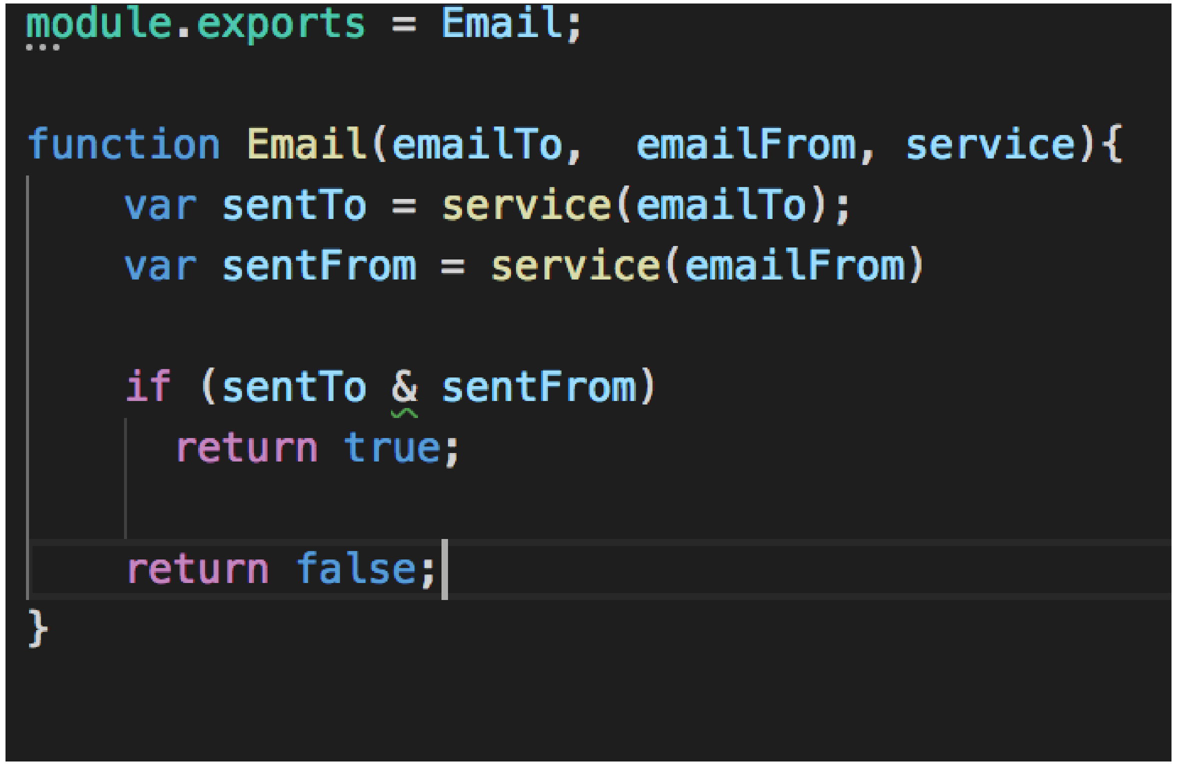
```
it('Sends Email to both emails', function(){
    var email1 = "aris@city.ac.uk";
    var email2 = "aristos.markogiannakis.1@city.ac.uk";

    var emailSpy = sinon.spy();
    Email(email1, email2, emailSpy);

    emailSpy.calledWith(email1).should.be.true;

});
```



```
module.exports = Email;

function Email(emailTo,  emailFrom, service){
    service(emailFrom);
}
```

Stubs

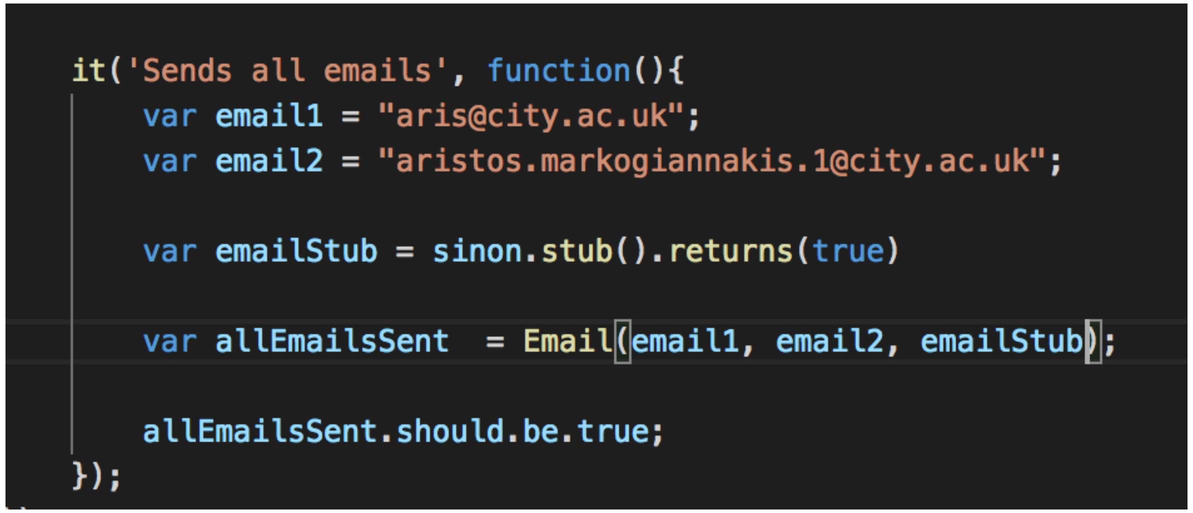
- Sinon Stubs can be used to make sure that a certain value has returned,
- For example in our example imagine we had to send the email twice if all emails have been sent then the function would return true



```
module.exports = Email;

function Email(emailTo,  emailFrom, service){
    var sentTo = service(emailTo);
    var sentFrom = service(emailFrom)

    if (sentTo & sentFrom)
      return true;

    return false;
}
```

Stub

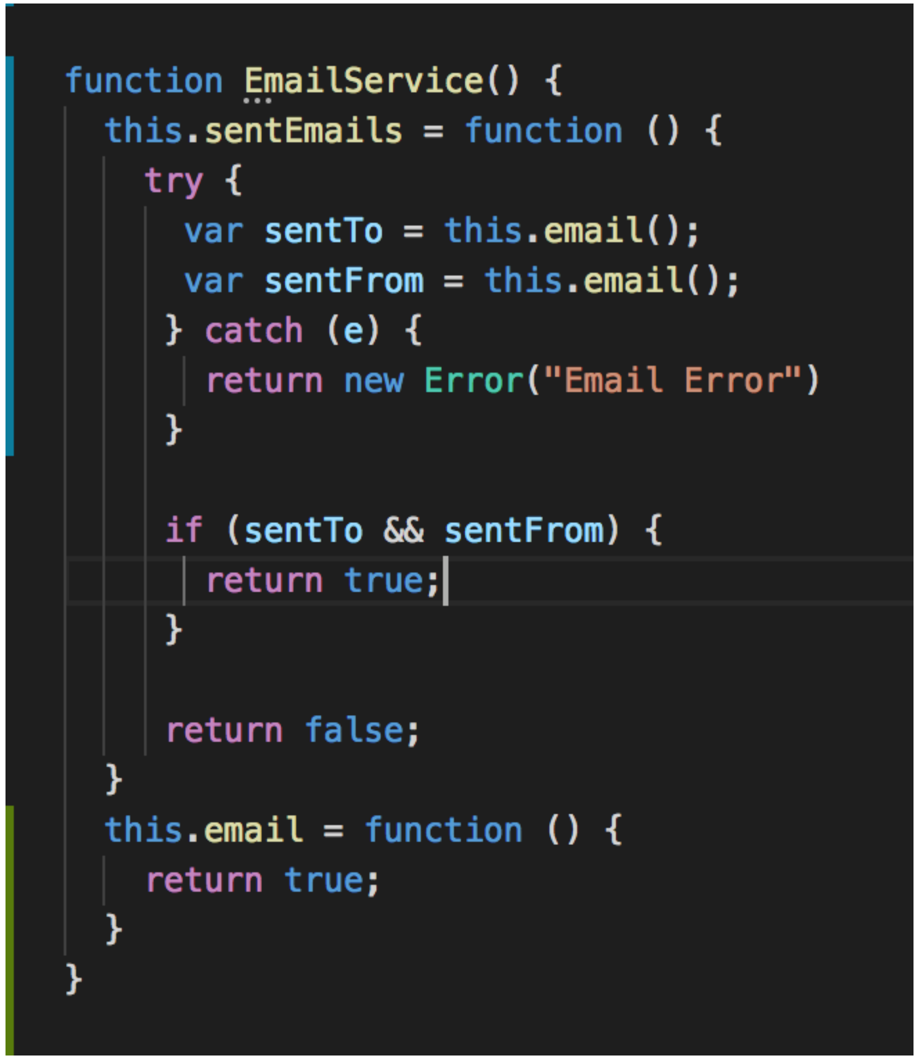
- So now we will implement our stub,
  - We have created a stub, that we expect to return true
  - On our code we have to send where we expect when those email are sent to receive a true.
  - If all those return true then the **allEmailsSent** will return true.



```
it('Sends all emails', function(){
    var email1 = "aris@city.ac.uk";
    var email2 = "aristos.markogiannakis.1@city.ac.uk";

    var emailStub = sinon.stub().returns(true)

    var allEmailsSent  = Email(email1, email2, emailStub);

    allEmailsSent.should.be.true;
});
```

Mock

- We can use Mock to mock an object, in our case we can change the code to create a Mock object
- Imaging the class on the right where we have an EmailService
- What we want to test is when we run our sentEmails the email method was executed twice.

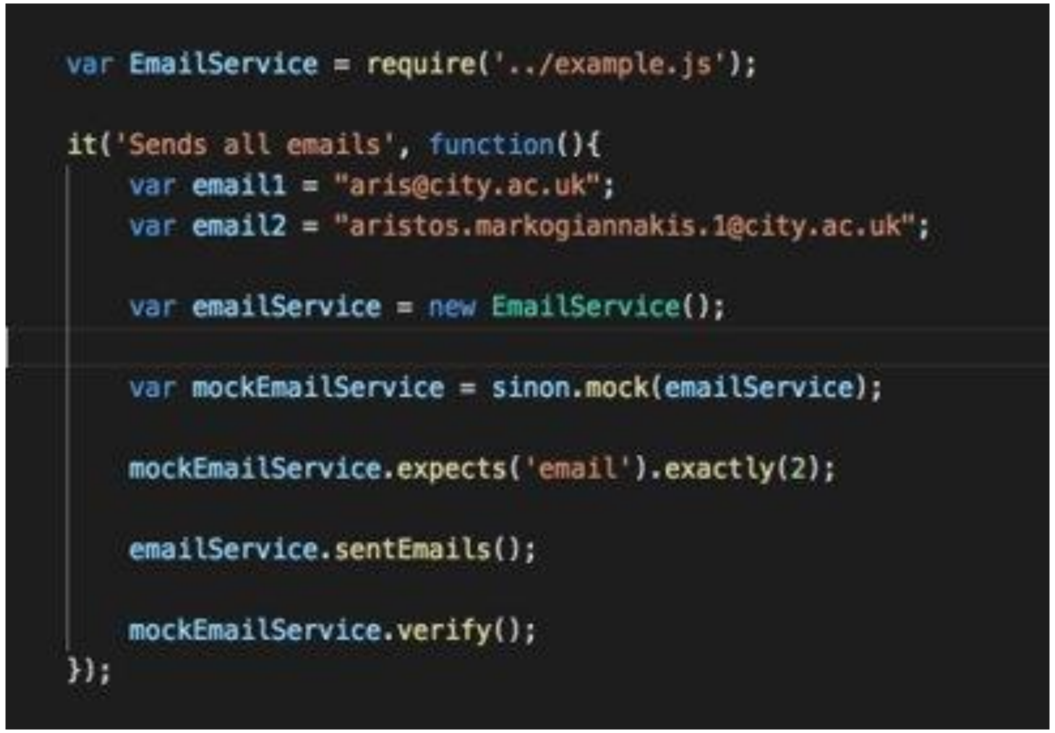


```
function EmailService() {
  this.sentEmails = function () {
    try {
      var sentTo = this.email();
      var sentFrom = this.email();
     } catch (e) {
      return new Error("Email Error")
     }

     if (sentTo && sentFrom) {
      return true;
     }

     return false;
  }
  this.email = function () {
    return true;
  }
}
```

Mock that object!

- We will import the EmailService and then using sinon.mock we will add the object we want to mock.

- We will then take the mock and declare the function that we expect to run twice!
- We will run the method from the original object
- And then verify.



```
var EmailService = require('../example.js');

it('Sends all emails', function(){
    var email1 = "aris@city.ac.uk";
    var email2 = "aristos.markogiannakis.1@city.ac.uk";

    var emailService = new EmailService();

    var mockEmailService = sinon.mock(emailService);

    mockEmailService.expects('email').exactly(2);

    emailService.sentEmails();

    mockEmailService.verify();
});
```

Further resources

- Check mochajs.org for more information on Mocha
- Check chaijs.com for more information on Chai
- Check sinon.js website for more information on Sinon

- When we start doing React will also visit Enzyme