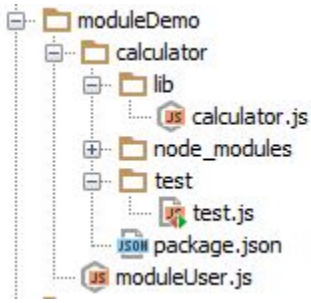
Exercises – Mocha Testing with Mocha, Chai and Nock

Use http://mochajs.org/ as reference for Mocha

And http://chaijs.com/api/bdd/ as reference for the chai assertion library

1) Basic Mocha/chai testing

Create a simple calculator module that should implement the four standard arithmetic operations.



Provide the module with a file structure as given here.

Let the `divide(n1,n2)` function throw an Error (`throw new Error("Attempt to divide by zero")`).

Test the functionality using Mocha/chai

2) Testing Asynchronous Code

If you have not already completed the "Make It Modular" exercise in the learnyounode exercises do it now.

This exercise requires you to write a module with a function that; given a path and a filter, returns all files that matches the filter.

This is an asynchronous function and it should be tested ;-)

a)

Initially you can test the method with a known existing folder (we know the content of the folder). Use http://mochajs.org/#asynchronous-code as a reference

b) The problem with the solution above is that it relies on the content of an existing folder. If anyone deletes the folder, delete files in it or add files, the tests will fail.

Use the before() and after() or beforeEach() and afterEach() methods to set up a temporary test folder with dummy files and delete the files and the folder after the test.

Hint: Even though asynchronous is the Node way of doing things, I suggest you use the synchronous versions of the methods relevant for this exercise:

- `fs.mkdirSync(..)` Create a Directory
- `fs.writeFileSync(..)` Creates a file
- `fs.unlinkSync(..)` Deletes a file
- `fs.rmdirSync(..)` Removes (an empty) directory

3) Test a Rest API

Test the REST-API implemented in the exercise: "Problem-4 Getting RESTless". If you haven't completed this part, use this simple implementation of (only) the REST-API:

https://github.com/Lars-m/simpleJokeAPI.git (remember `npm install`)

For this test I recommend that you actually test the real server, using real HTTP-requests. There are many packages which can help (see for example Supertest) but I suggest the *request* package (remember `npm install request --save`) since its one the most used packages and definitely worth learning.

The example below shows, how you can *start*, and *stop* the server from within your test, how to perform a *POST-request* using the *request-package*, and a simple asynchronous test

```
var expect = require("chai").expect;
var request = require("request");
var expect = require("chai").expect;
var http = require("http");
var app = require('../app');
var server;
var TEST_PORT = 3456;

before(function(done){
  var app = require('../app');
  server = http.createServer(app);
  server.listen(TEST_PORT,function(){
    done();
  });
})
after(function(done){
  server.close();
  done();
})

describe("POST: /api/joke ", function () {
  var options = {
    url: "http://localhost:" + TEST_PORT + "/api/joke",
    method: "POST",
    json: true,
    body: {joke: "Its better to be late than to arrive ugly"}
  }

  it("should get a random joke", function (done) {
    request(options, function (error, res, body) {
      var addedJoke = body.joke;
      expect(addedJoke.joke).to.be.equal("Its better to be late than to arrive ugly");
      //You should also check whether the joke actually was added to the Data-store
      done();
    });
  })
});
```

Complete the test to test the full JOKE-API.

The test above has a problem in that it leaves the "database" changed. Find a way to reset the jokes array to an initial state before each of these tests.

5) Mocking HTTP-requests

Implement a (reusable) module as sketched here:

http://js-plaul.rhcloud.com/test1/unitTestingBackend.html#20

Use the nock package to mock out the actual network request and test the module as outlined in part-2 of the slide.

6) Mocking HTTP-requests, continued

Imagine a module that provides information about available flights from an airline as sketched below:

```
var request = require("request");
//This is how you call the API
//http://angulararirline-plaul.rhcloud.com/api/flightinfo/SXF/2016-03-09T00:00:00.000Z/4
function getAvailableTickets(airport,date,numbOfTickets,callback){
  var isoDate = date.toISOString();
  var URL =
    "http://angulararirline-plaul.rhcloud.com/api/flightinfo/"+airport+"/"+isoDate+"/"+numbOfTickets;
  request(URL, function (error, response, body) {
    if (!error && response.statusCode == 200) {
      return callback(null,JSON.parse(body));
    }
    else{
      return callback(error, JSON.parse(body))
    }
  })
}

module.exports = getAvailTickets = getAvailableTickets;
```

You can use the module using a require statement and call the function like:

```
var angularAirline = require(..);

var date = new Date("2016-03-09");
getAvailableTickets("SXF",date,4,function(err,response){
  console.log(response);
});
```

This would give you a response like this:

```
{ airline: 'AngularJS Airline',
  flights:
   [ { flightID: 'COL2215x100x2016-03-09T08:00:00.000Z',
       numberOfSeats: 4,
       date: '2016-03-09T08:00:00.000Z',
       totalPrice: 340,
       traveltime: 60,
       origin: 'SXF',
       destination: 'CPH' } ] }
```

Testing a module like this will be hard, since it depends on an external resource (what if net is unavailable) and return a reply we can't predict (at least for a real airline).

Use nock, and your experience from ex-5 to write a test that mocks out the real http request.

Ideally you should also test the reaction the erroneous scenarios. Call the API, from a browser to see the

expected result for these scenarios, an implement a (mocked) test to test the module.