Lab- Mocha Framework with Chai and Sinon

November, 2015

Document Control

Version	1
Created by	Karthikeyan.J
Updated by	Karthikeyan.J
Reviewed by	Chand Ali
Date	30.10.2015
Earlier versions	

Table of Contents

1. Overview

This lab book is a guided tour for learning Mocha framework with Chai and Sinon framework. It will show you, how to unit-test simple JavaScript & AngularJS applications with Unit Testing frameworks like Mocha.

- **Mocha:-** Mocha is a JavaScript **test framework** running on node.js, featuring browser support, asynchronous testing, test coverage reports, and use of any assertion library.
- **Chai:-** It is assertion library that can be paired with any Javascript testing framework.
- Mocha does not come with inbuilt assertion library like Jasmine framework so, we will be using Chai along with Mocha.
- **SinonJS:-** It comes with standalone test spies,stubs and mocks for JavaScript. It has no dependencies and can work with any unit-testing framework.

2. Pre-requisites

- JavaScript
- Unit Testing Concepts

3. Software Requirement

- Javascript editor like Webstorm,Aptana Studion,Visual Studio,Eclipse
- NodeJS (appropriate version)

  We need Nodejs (JavaScript Runtime) to load and run karma Test runner (written in JS) as well as to load mocha,chai,sinon libraries

4. Installation

- Install nodejs (if not already installed) from **https://nodejs.org/en/**
- Install **Mocha & Chai** using **npm**
  - Open a command prompt using **Start→Run→cmd**
  - Type the following command at the command prompt:

    npm install –g mocha chai

5. Labs

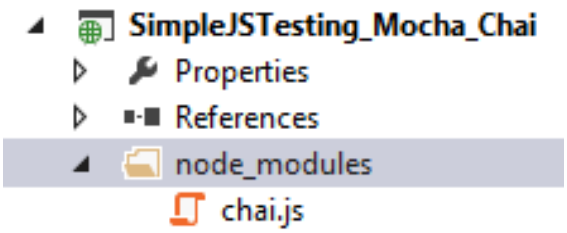
Lab 1: Testing simple JavaScript code using Mocha & Chai

In this lab, we will use **Mocha & Chai** to test simple JavaScript functions. We will also learn how to use different assertion styles provided by **Chai.**

It is recommened that you should use **Visual Studio 2012/2013 Community Edition** for implementing these labs. However, you may use an IDE of your choice.

- Create a web application named **SimpleJSTesting_Mocha_Chai** on the local drive of your system.
- Create a folder named **node_modules** in the root of the web application & copy the file **chai.js** from the **node_modules** folder into this folder.
    - Note that the **node_modules** folder **(npm)** got installed when you made a global installation of **Mocha & Chai** earlier. You will find this installation at the following location:
    - **C:\Users\<<logged in username>>\AppData\Roaming**

- Your project structure must appear as shown below (this snapshot has been taken from Visual Studio 2013)



- Create a file named **test.js** in the root of the web application.
- Add the following code inside it:

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

function isEven(num)
{
    return (num % 2 == 0);
}

describe('Testing Simple Java Script', function ()
{
    it('should return true if the number is even', function ()
    {
        //use a should style assertion
        isEven(4).should.to.be.true;
    });

    it('should return false if the number is odd', function ()
    {
        //use an expect style assertion
        expect(isEven(5)).to.be.false;
    });
});
```

- Now let us run this simple **test suite** using command-line.
  - Switch to the command prompt.
  - Navigate to the web application folder which contains the **test.js** file.
  - Run the following command:

    mocha test.js

    This command will run all the tests inside the .js file & report whether the tests passed or failed. Mocha uses the name of the "describe" and "test" as a useful documentation for our code.
  - Notice that Mocha displays a simple readable English sentence-like output as shown below:

```
Testing Simple Java Script
  √ should return true if the number is even
  √ should return false if the number is odd


2 passing (15ms)
```

    Also notice that in the first test, we have used a "should" style assertion, whereas in the second test, we have used an "expect" style assertion.

Lab 2: Using setup & teardown

In this lab, we will implement **setup & teardown** for our tests using **beforeEach()** & **afterEach()** methods.

- Add another function named **Add()** inside **test.js** as follows:

```
function Add(num1, num2)
{
    return parseInt(num1) + parseInt(num2);
}
```

- Add the following test suite which will test this function on a hard-coded value. We will add two tests, each will add some number to the hard-coded value.

```
describe("Testing with setup and teardown", function ()
{
    var num = 5;

    it("should be 10 when adding 5 to 5", function ()
    {
        num = Add(num, 5);
        num.should.equal(10);
    });

    it("should be 12 when adding 7 to 5", function ()
    {
        num = Add(num, 7);
        num.should.equal(12);
    });
});
```

Notice that we have added 5 & 7 to the hard-coded value 5. Also, notice that the hard-coded value has been initialized inside the "describe" block. This is to make sure that the initial value of "num" is available to each test.

- Run the tests from the command line.

Notice that the second test fails as shown below:

```
Testing with setup and teardown
  √ should be 10 when adding 5 to 5
  1) should be 12 when adding 7 to 5


1 passing (10ms)
1 failing

1) Testing with setup and teardown should be 12 when adding 7 to 5:

    AssertionError: expected 17 to equal 12
    + expected - actual

    -17
    +12
```

This is because, as expected the value of "num" is initialized only once before the first test is run & not when the second test is run. So, for the second test, the value of "num" is 10 and not 5 as expected.

- We can use the **beforeEach() & afterEach()** functions to execute code before & after each & every test runs. Add the following functions inside the **describe** block:

  Notice that the "num" variable is declared inside the "describe" block & initialized inside "beforeEach()" function.

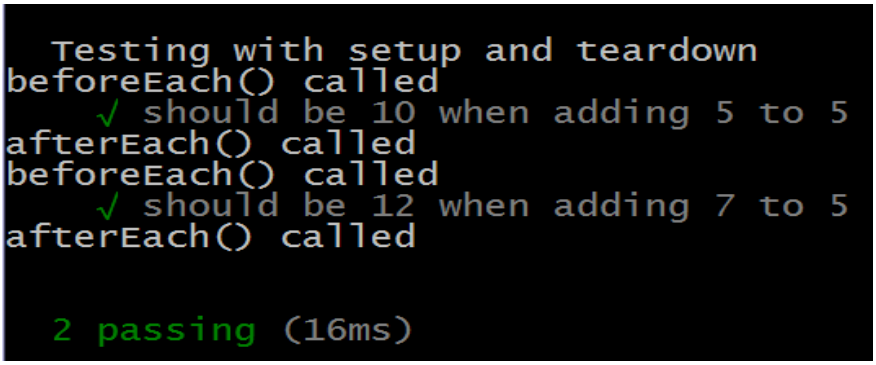
```
var num;

beforeEach(function ()
{
        num = 5;
        console.log("beforeEach() called");
});

afterEach(function ()
{
        console.log("afterEach() called");
});
```

- Run the test again. Both the tests will now pass as shown below:

  In the output, notice how the "beforeEach()" & "afterEach()" functions are executed before & after each test is executed.



Lab 3: Nested describes

In this lab, we will use **nested describes** to group our test suites.

- Add the following functions inside **test.js:**

```
//numeric functions

//check if a number is divisible by 3
function isDivisibleByThree(number)
{
    return (number % 3 == 0);
}

//check if a number is a prime number
function isPrime(number)
{
    var prime = false;
    for(var i=2 ; i<number-1 ; i++)
    {
        if(i % number==0)
        {
            prime = true;
            break;
        }
    }
    if (prime == true)
        return true;
    else
        return false;
}


//string functions

//check whether a string contains atleast 5 characters
function minLength(str)
{
    return str.length >= 5;
}

//reverse a string
function reverseString(str)
{
    return str.split('').reverse().join('');
}
```

- Add the following nested describes:

```
//nested describes
describe("MY TESTS", function()
{
    describe("NUMERIC TESTS", function()
    {
        describe("isDivisibleByThree", function ()
        {
            it("should return true if 9 is divisible by 3", function()
            {
                isDivisibleByThree(9).should.to.be.true;
            });
            it("should return false if 4 is not divisible by 3",
                    function()
            {
                isDivisibleByThree(4).should.to.be.false;
            });
        });
    });

    describe("STRING TESTS", function()
    {
        describe("minLength", function()
        {
            it("should return true if 'hello' contains atleast 5
                    characters", function ()
            {
                expect(minLength("hello")).to.equal(true);
            });

            it("should not return false if 'abc' does not contain atleast
                    5 characters", function()
            {
                expect(minLength("abc")).to.equal(false);
            });
        });
        describe("reverseString", function()
        {
            it("should return 'cba' for 'abc'", function()
            {
                expect(reverseString("abc")).to.equal("cba");
            });
            it("should not return 'cba' for 'cba'", function()
            {
                expect(reverseString("cba")).to.equal("abc");
            });
        });
    });
});
```

- Run the tests to get the following output:

  Notice how the test suites have been grouped now.

```
MY TESTS
  NUMERIC TESTS
    isDivisibleByThree
      √ should return true if 9 is divisible by 3
      √ should return false if 4 is not divisible by 3
  STRING TESTS
    minLength
      √ should return true if 'hello' contains atleast 5 characters
      √ should not return false if 'abc' does not contain atleast 5 characters

    reverseString
      √ should return 'cba' for 'abc'
      √ should not return 'cba' for 'cba'
```

Lab 4: Skipping & running specific tests

In this lab, we will see how you can skip/run certain tests. Mocha allows us to do this without commenting any test.

Let us first skip some tests from running.

You can do this in two ways:

- By using the skip() method on the it() or describe() methods.
- By using xit() or xdescribe() instead of it() & describe().

Let us skip the *isDivisibleByThree* test suite.

- Modify the **describe** for the **isDivisibleByThree** test suite as follows:

```
describe.skip("isDivisibleByThree", function()
```

- Run the tests again.

  Observe that the test suite has been skipped & mocha explicitly informs us by showing the status as "pending".

```
MY TESTS
  NUMERIC TESTS
    isDivisibleByThree
      - should return true if 9 is divisible by 3
      - should return false if 4 is not divisible by 3
  STRING TESTS
    minLength
      √ should return true if 'hello' contains atleast 5 characters
      √ should not return false if 'abc' does not contain atleast 5 characters

    reverseString
      √ should return 'cba' for 'abc'
      √ should not return 'cba' for 'cba'


4 passing (15ms)
2 pending
```

- Try skipping the **isDivisibleByThree** test suite by modifying the **describe** as follows:

```
xdescribe("isDivisibleByThree", function()
```

- Try using **it.skip()** or **xit()** to skip specific tests within a test suite.

Now, let us see how to run only specific tests.
You can use the only() function on a describe() or it() when you want to run a specific test or tests within a test suite. Please note that when using only(), all the other tests are automatically excluded from running.
Let us run only all numeric tests.

- Modify the **describe** of the numeric test suite as follows:

```
describe.only("NUMERIC TESTS", function()
```

- Run the tests.
  Observe that in this case, only the *NUMERIC TESTS* test suite is run & all other test suites are excluded.

- Try using the **only()** function with an **it()** to run only a specific test within a test suite

Lab 5: Using Mocha's watch feature

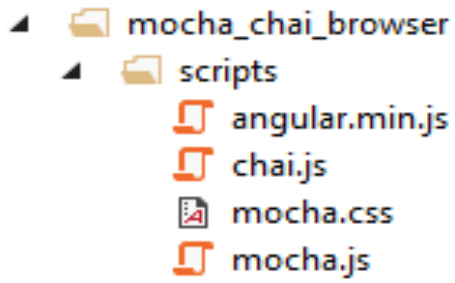
In this lab, we will use Mocha's **watch** feature. This feature watches test files as they change & Mocha re-runs all the tests again when the test file changes.

- Run the **mocha** command from the command line with the **–w** switch:
  - ***mocha –w <<filename>>.js***
- In this case, the command prompt does not return.
- Make some changes in your test file & save the changes.
- The moment the changes are saved, Mocha picks up those changes & immediately re-runs all the tests again, without you having to run the **mocha** command again.

Lab 6: Running Mocha tests inside a browser

In this lab, we will use a browser to run our tests. This lab uses **Angular JS** to create a test suite on the fly based on the inputs supplied through a view. This example has ample scope for modification/enhancement, but to begin with, it gives you a very good idea on how to run Mocha using a browser instead of a command line.

- To begin with, add a folder named **mocha_chai_browser** at the root of your web application.
- Add a sub-folder named **scripts** inside the folder created above.
- Copy/Add the files shown in the snapshot below inside the **scripts** folder:



- Add a JavaScript file named **src.js** inside the **mocha_chai_browser** folder.
- Add the following JavaScript functions inside it:

```
function Add(num1,num2)
{
    return (num1 + num2);
}

function Square(num)
{
    return (num * num);
}
```

- Add an HTML file named **Index.html** inside the **mocha_chai_browser** folder.
- Delete the existing code inside it (if any).
- Add the following <head> element:

```
<head>
    <title></title>
    <script src="scripts/mocha.js"></script>
    <script src="scripts/chai.js"></script>
    <script src="src.js"></script>
    <link href="scripts/mocha.css" rel="stylesheet" />
    <script src="scripts/angular.min.js"></script>
</head>
```

- Add a **<body>** element.
- Append the following inside the **<body>** element:

```
<div id="mocha" style="border-style:solid;border-color:orange;border-
          width:5px;margin:5px;padding:10px;height:200px;overflow:scroll;">

</div>

<br />

<div style="border-style:solid;border-color:blue;border-
          width:2px;margin:5px;padding:10px;">
        <h2 align="center">Testing function Add()...</h2>

        <p>
            Enter value for operand1:
            <input type="text" ng-model="Operand1" />
        </p>


        <p>
            Enter value for operand2:
            <input type="text" ng-model="Operand2" />
        </p>

        <input type="button"  value="Run Test" ng-click="RunTests('add')"/>
</div>

<div style="border-style:solid;border-color:blue;border-
          width:2px;margin:5px;padding:10px;">
        <h2 align="center">Testing function Square()...</h2>

        <p>
            Enter number:
            <input type="text" ng-model="Number" />
        </p>
        <input type="button" value="Run Test" ng-click="RunTests('square')" />
</div>
```

- Append the following inside the **<body>** element:

```
<script>mocha.setup('bdd');</script>
```

  This tells Mocha to use the BDD-style interface

- Append the following **<script>** tag inside the **<body>** tag:
  This code sets up a "describe" on the fly based on the function being tested. This example just demonstrates how to run Mocha inside a browser. The integration with Angular JS in this example is just to supply the inputs for testing. The example has ample scope to be enhanced/modified.

```
<script>
                var module = angular.module("MainApp", []);
                module.controller("TestController", function ($scope) {
                    $scope.Operand1 = 0.00;
                    $scope.Operand2 = 0.00;
                    $scope.Number = 0;

                    $scope.RunTests = function (name)
                    {
                        chai.should();
                        if (name == 'add')
                        {
                            var addresult = parseFloat($scope.Operand1) +
                                 parseFloat($scope.Operand2);
                            describe("MOCHA ON BROWSER", function ()
                            {
                                it.only("should return " + addresult + " when adding:
                                        " + $scope.Operand1 + " and " +
                                        $scope.Operand2, function ()
                                {
                                    Add(parseFloat($scope.Operand1),
                                        parseFloat($scope.Operand2)).
                                        should.equal(addresult);
                                });
                            });
                        }
                        else if (name == 'square')
                        {
                            var square = Square(parseFloat($scope.Number));
                            describe("MOCHA ON BROWSER", function ()
                            {
                                it.only("should return " + square + " when
                                 calculating square of " + $scope.Number, function ()
                                {
                                    Square(parseFloat($scope.Number)).
                                        should.equal(square);
                                });
                            });
                        }
                        mocha.run();
                    };
                });
</script>
```

- Run the **Index.html** page. Enter inputs for each function & click on the **Run Tests** button. Notice that the test is run & the output is displayed inside the ** ** element whose id is **mocha.**

Lab 7: Testing Angular JS code using Mocha & Chai

In this lab, we will test an Angular JS Controller using Mocha & Chai.

- Write the following Angular JS code inside a file named **src.js:**

```
function Patient(patientid, patientname, age)
{
    this.patientid = patientid;
    this.patientname = patientname;
    this.age = age;
}

var module = angular.module("TestApp", []);
module.controller("PatientController", function ($scope)
{
    $scope.Patients =
    [
        new Patient(1, "Patient1", 34),
        new Patient(2, "Patient2", 36),
        new Patient(3, "Patient3", 31),
        new Patient(4, "Patient4", 32),
        new Patient(5, "Patient5", 30),
    ];

    $scope.GetPatient = function (patientid)
    {
        //console.log("id " + patientid);
        if (patientid == undefined)
        {
            return null;
        }
        else
        {
            var found = false;
            var patient;
            for (var i = 0; i < $scope.Patients.length; i++)
            {
                if ($scope.Patients[i].patientid == patientid)
                {
                    found = true;
                    patient = $scope.Patients[i];
                    break;
                }
            }
            if (found)
                return patient;
            else
                throw "Patient with id: " + patientid + " not found";
        }
    };
});
```

- Include the following script after the call to **mocha.setup()** function:

```
<script src="scripts/angular-mocks.js"></script>
```

- Following is a test suite to test the Angular JS code:

```
<script>
        chai.should();
        var expect = chai.expect;

        describe("GetPatient() of PatientController", function ()
        {
            beforeEach(angular.mock.module('TestApp'));
            var thescope;


            beforeEach(angular.mock.inject(function ($controller,
                    $rootScope)
            {
                thescope = $rootScope.$new();
                $controller("PatientController",
                    {
                        $scope: thescope
                    });
            }));

            it("should return a 'Patient' object when correct patient id
                    is entered", function ()
            {
                thescope.GetPatient(2).should.be.an.instanceof(Patient);
            });

            it("should throw an exception when an incorrect patient id is
                    entered", function ()
            {
                //Using expect to test whether a method throws an
                //exception or not
                //expect(function () { thescope.GetPatient(999);
                          }).to.throw();

                //using should to test whether a method throws an
                //exception or not
                (function () { thescope.GetPatient(999);
                    }).should.throw();
            });
            it("should return 'null' if patientid is not passed",
             function ()
            {
                //using expect to test whether a method returns null or
                //not
                expect(thescope.GetPatient()).to.be.null;
            });

            it("should return the correct patient object when passed a
                    patient id", function ()
            {
                //randomly extract a patient object from the array
                var thispatient = thescope.Patients[4];
                thescope.GetPatient(5).should.eql(thispatient); //check
                    //deep equality of objects
            });
        });
mocha.run();</script>
```

Lab 8: Using Sinon Spies

In this lab, we will use Sinon, a very popular Mocking Library. For this lab, we will use the following JavaScript code:

```
var patient, hospital;

patient =
        {
            giveTreatment: function (doctorid, callback) {
          //check of callback is an object & has onGiveTreatment() method
                if (!!callback.onGiveTreatment) //if function exists
                {
                    console.log("object passed");
                    callback.onGiveTreatment();

                }
                else
                {
                    console.log("func passed");
                    callback();
                }
            },
        };

        //create another object
        hospital = {
            onGiveTreatment: function () {
               console.log("hospital object's onGiveTreatment() called");
            }
        };
```

In the above example, the **giveTreatment()** checks if the callback has a function named **onGiveTreatment().** If yes, then it is called on the callback object. In case, the callback is not an object, then the callback is called directly.

Now, you can use a Sinon spy to check if the callback is actually called or not,irrespective of whether the callback is a standalone function or it belongs to an object.

Following is a test case for the same:

```
it("should call the callback and log to the console", function ()
{
            function onGiveTreatment()
            {
                console.log("onGiveTreatment() was called");
            }

            var spy = sinon.spy(onGiveTreatment);
            patient.giveTreatment(1, spy);
            expect(spy.called).to.be.true;
});
```

Notice that in this case, the **onGiveTreatment()** function is directly passed to the **spy()** function. In other words, the **onGiveTreatment()** function does not belong to an object.

Following is a test case which can be used when the callback function belongs to an object:

```
it("should call the callback even if it's a method of an object", function ()
{
  //pass the hospital object which has the callback
  //var spy = sinon.spy(hospital.onGiveTreatment);
  //patient.giveTreatment(1, spy);    //WORKS

  //ASSERTION ERROR because onGiveTreatment() is called on object and not on
spy
  //patient.giveTreatment(1, hospital.onGiveTreatment);

  //expect(spy.called).to.be.true;

  //SOLUTION
  //var spy = sinon.spy(hospital, "onGiveTreatment");
  //patient.giveTreatment(1, spy);   //WORKS
  //patient.giveTreatment(1, hospital);   //WORKS
  //patient.giveTreatment(1, hospital.onGiveTreatment);   //WORKS
  //expect(spy.called).to.be.true;
});
```

Lab 9: Using Sinon Stubs

A Sinon stub is almost like a spy. The main difference is that a spy invokes the actual implementation of the function on which it is spying, whereas a stub does not. A stub is typically useful to monitor all functions within an object.

Following is a test case which uses Sinon stubs:

```
it("should call a stubbed method", function ()
{
   //pass the object to the stub
   //sinon will go through each function in this object & replace it with
   //a stub function
   var stub = sinon.stub(hospital);

   //CAN PASS A FUNCTION ON THE STUB OR EVEN THE ENTIRE STUB
   //patient.giveTreatment(1, stub.onGiveTreatment);
   patient.giveTreatment(1, stub);
   expect(stub.onGiveTreatment.called).to.be.true;
});
```

6. Summary:

Mocha is a test runner which can be executed from Node as well as from a browser. It provides a BDD and TDD-style of testing interface.

Chai is an assertion library which allows us to choose from a variety of assertion syntaxes. It can be used with any testing framework as such.

Sinon is a mocking library. Sinon uses spies and stubs as test doubles. The main difference between both is that a spy invokes the actual method implementation during testing whereas a stub does not. In addition, a stub automatically monitors all the functions of an object.