**Unit Testing**

**Mocha –**

**Chai for assets –** more readable,

**Sinon for mocking**

**Then we do It all together**

**Not everthing is easy**

**Istanbul for code coverage**

**Unit vs Integration vs Functional Testing**

**Unit Testing**

Find the smallest available piece. Just a single a function. Test just that. Mock everything else.

**Testing with Mocha**

Just a Test runner ( Jasmine, Mocha, Justine)

Lets dig into mechanics. Managing what test execute.

**Assert**

The **assert** moduled provides a simple set of assertion(**твърдение**) tests that can be used to test invariants.

**Assert(value[,message])**

**Assert.deepEqual(actual, expected[,message])**

Test for deep equality between the **actual** and **expected** parametes. Primitive values are compared with ( **==)**

**Default timeout for mocha test is 2000 miliseconds!!!**

**Hooks**

Before(), after(), beforeEach(), afterEach((). These should used to set up preconditions and clean up after your test.

**Before() –** runs before all tests in this blocks

**After() –** runs after all tests In this block

**BeforeEach() –** runs before each test in this block

**afterEach() –** runs after each test in this block

**PENDING TESTS**

**“**Pending**” ( в очакване на) –if “**someone should write these test case eventually“ – test-cases are simply those *without a callback*

***It(‘Should return -1 when the value is not present’)***

**Exclusive tests**

Exclusive feauture allows you to run **only** the specified suite or test-case by appending **.only()** to the function. Here’s example

**Describe(‘Array’, function() {**

**Describe.only(‘#indexOf’, function() {   
}**

**}**

**Note:** All nested suites will be executed

**Individual test case**

It.only(‘should return only -1 unless present’, function() {  
 }

INCLUSIVE TESTS

This feature is the inverse of **.only().** By appendinding **skip() you** may to tell Mocha to simply ingnore these suite(s) and test case(s)/> Anything skipped will be marked as pending and report as such

**Describe(‘Array’, function() {   
describe.skip(‘#indexOf’, function() {   
});**

**});**

You may also skip at *runtime* using this.skip(). *I*f a test needs an environment of configuration which cannot be detected beforehand, a runtime skip is appropriate.

it('should only test in the correct environment', **function**() {

**if** (*/\* check test environment \*/*) {

*// make assertions*

} **else** {

**this**.skip();

}

});

**BDD-Style Assertions**

Provides describe(), contect(), it(), specify(), before(), after(), beforeEach(), afterEach()

**Assert is clunky**

**Chai for better assertions**

**Two options –** expect and should

**Verifying objects**

**Expect = it(‘should …’)**

**Expect(something).to.be**

**Expect(something).to.equal**

**Expect(something).to.have**

**Expect(auth).to.be(true)**

Should – Clear, Natural Language (Something.should…)

* Should add itself to Object.prototype
* Should is a function that’s to need to be executed **chai.should()**
* **Shoud is available everywhere -**

Expect is a Step in the Right Direction

**Async with Promises (install chai-as-promised0)**

**Testing is not always straight forward**

**Mocking objects is required to unit test**

**SINON to the rescue!**

**Install (npm install sanon)**

Let func = sinon.spy();

// Gives Us a Fake Function

A test spy is a function that records arguments, return value, the value of **this and** exception thrown (if any) for it all calls. There are two spies: Some are anonymous functions, while othew wrap methods that already exist in the system under test.

**Spy an anonymourse function**

When THE behavior of the spied-on function is not under test, you can use an anonymous function spy. The spy won’t expect record information about its calls. A common use case for this type of spy is testing how a function handles a callback, as in the fallowing simplified example.

**“test should call subscribers on publish”: function () {   
 var callback = sinon.spy();**

**PubSub.subcribe(‘message’, callback);**

**PubSub.publishSync(‘message’);**

**AssertTrue(callback.called);**

**}**

**Spy to wrap an existing method**

**Sinon.spy(object, “method”)** creates a spy that wraps the existing function **object.method.** The spy will behave exactly like the original method (including when used as a constructor), but you will have access to data about all calls. The fallowing is a slightly contrieved example

**setUp**