**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: function() {   
 sinon.spy(jQuery, ‘ajax’);**

**},**

**teardown: function() {  
 jQuery.ajax.restore(); //** unrwraps the spy

**},**

“test should inspect jQuery.getJSON’s usage of jQuery.ajax”: function() {   
 **jQuery.getJSON(‘/some/resource’);**

**Assert(jQuery.ajax.calledOnce);**

**AsseryEquals(‘some/resource’, jQuery.ajax.getCall(0).args(0).url);**

**} }**

**Var spy = sinon.spy();**

Create an anonymous function that records arguments **this** value, expceptions and return values for all calls

**Var spy = sinon.spy(myFunc);**

Spies on the provided function.

**Var spy = sinon.spy(object, “method”);**

Create a spy for **object.method** and replaces the original method with the spy. An exception is thrown if the property is not already a function. The spy axts exactly like the original method in all cases. The original method can be restored by calling **object.method.restore();** The returned spy is the function object which replaced the original method **spy === object.method**

**STUBS**

Test stubs are functions (spies) with pre-programmed behavior;

They support the full test spy API in addition to methods which can be used to alter the stub’s behavior.

As spies, stubs can be either anonymous, or wrap existing functions. When wrapping an existing function with a stub , the original function is not called.

**When to use stubs?**

Use a stub when you wan to:

1. Control a method’s behavior from a test to force the code down a specific path. Examples Including 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 **XMLHttpRequest** or similar)

**MOCKS**

Mocks (and mock expectations) are fake methods (like spies) with pre-programmed behavior

(like stubs) as well as **pre-programmed expecations.**

A mock will fail you test If is not used as expected.

**When to use mocks?**

Mocks should only be used the *method under test* in every unit test, there should be one unit under test.

If you want to control how your unit is being used and like stating expecations upfront (as opposed to asserting after the fact),use a mock.

**When not use the mocks?**

Mocks come with built-in expectations that may fail test.

Thus, they enforce implementation details. The rule of thumb is: if you would not add a an assertion for some specific call, don’t mock it. Use a stub instead.

In general you should have  **no more than one mock** ( possibly with several expecations) in a single test.

**Expectations** implement both the spies and the stubs APIs.

// Telerik

**Unit testing –** Нещо, което проверява, дали нещо работи коректно. Тества се изолирана част от кода. Ако тества повече от един файл, това не е **unit test.**

**Frameworks**

* Jsunit –
* qUnit – jQuery testing. Front-end testing
* **Jasmine, Mocha (pluggable framework) –** примерно USB( където може да сложим мишка, клавиатура, и т.н.)
* **Mocka (дава ни framework), но иска и някакъв синатикси, с който да пишем unit tests**
* **Assert, should, expect ( CHAI ни дава синтаксиса)**
* **Should (extend-ва Prototype на object test, съответно всички обекти, получават един метод should –** не се използва
* **Assert.areEqual(expect, actual)**
* **Expect(actual).to.be.eql(expected); !!!**

**Yarn add mocha chai –dev (В** dev слагаме всичко, което го нямаме в production)

**Всичко, което не искаме да бъде качено в някакъв сайт, го слагаме в devDependencies**

**Describe – Задават се допълните методи, като**

* **Before() –** пуска се един път преди всички тестове за съответния describe
* **After() –** изпълнява се един път след всички тестове
* **beforeEach –** преди всеки тест
* **afterEach –** след всеки тест

**TOOLING**

* **IDE –> VSCode, WebStorm , Atom, Vim**
* **Debugging -> Node**
* **Nodemon -> automatic app rerun**
* **Package Managers - NPM , YARN, Bower**
* **App scaffolding – Yeoman (**
* **Task runners – Gulp, Grunt, WebPack –** Дават ни възможност да атоматизираме някакъв процес… Пример: Пишем си код, Някаква част, която се отваря в браузъра, ( js, ajax) …Конфигурираме си task runners, ккато пример да го компилира с Babel.
* (install gulp-cli, gulp – g)