A TASTE OF

CLIENT-SIDE UNIT TESTING

@antoniocapelo

UNIT TESTS

COOL, RIGHT?

ADVANTAGES

- unit tests can serve as a 'code' documentation of your running component (in case of a lack of documentation)
- they give you more confidence on the code you're pushing to master
  - ensure that we're warned if new features/refactors break the expected behavior

ADVANTAGES

- they enforce you to write more organized code (and testable) code, or else the unit testing proccess is painful
- they make your source code clearer - sometimes you find out that a certain logic is invalid/useless because you just can't test it

AND..

- JavaScript is a dynamically typed language so it comes with almost no help from the compiler - more easier to break if something changes (or if we code it wrong)

```
// Cannot read property 'length' of undefined

// undefined is not a function
```

anyone?

OBSTACLES

- it takes time, and we got features to deliver - we have to make space (i.e. give adequate points to US's) on the sprint grooming

- it can involve some refactoring when begining to test an already existing codebase - start as soon as possible

JS UNIT TESTING

TEST INDIVIDUALLY EACH APP COMPONENT

- This means that when testing the component X, each interaction with components Y and Z will be mocked, because:
  - we're not testing the other components behavior, that belongs on Y and Z unit tests
  - we have control on our tests scenarios (component X state)

RULE OF THUMB

- test the returned value of *'public methods'* and current value of *'public properties'* of the component

WRITING TESTABLE JS CODE

SAMPLE APP

Items to choose (.available)

- Cras justo odio
- Dapibus ac
- Morbi leo risus
- Porta ac consectetur ac
- Consectetur ac
- Facilisis in
- Porta Vestibulum
- Porta ac consectetur ac
- Vestibulum at eros

Prev 1 2 3 4 5 Next

My List (.my-list)

3 (.items-count)

- Cras justo odio
- Porta ac consectetur ac
- Vestibulum at eros

TYPICAL JQUERY-ISH CODE

```
$(document).ready(function(){
    var listDiv = $('.available'),
    myList      = $('.my-list'),
    itemCount   = getItemCount( '.item' );

    $(.items-count).html(itemCount);

    listDiv.on('click',function(){
        var $clickedEL = $(this);
        $.ajax(...);
        function onSucces() {
            $clickedEL.remove();
            var $newEl;
            itemCount+=1;
            $(.items-count).html(itemCount);
            $newEl = $('<li class="item">').text($clickedEL.text());
```

(NOT TESTABLE)

WHY?

- All logic is hidden inside a ready() function
- CSS classes and jquery selectors sprayed accross the snippet
- anonymous functions harder to test
- (not related with tests) - code harder to reuse and customize

TESTABILITY++

```
function MyListComponent($context) {
    var compClasses = {
        item: 'item'
    };
    var compSelectors = {
        item : '.item',
        component : '.my-list',
        itemCount : 'item-count'
    };

    // Private
    var $component = $context.find(compSelectors.component);

    function initComponent() {
        updateItemCount();
    }
```

WHAT HAPPENED?

By creating the `MyListComponent` we can:

- **focus** on what the component should do
- test it **individually**
- quickly **pinpoint** the broken logic spot on future regressions or refactors
- **divide work** when creating the tests (one dev can test just this component while another tests its sibling)

- besides that, we:
  - increased **reusability**
  - separated concerns
  - are happier :)

We can break it down even more (on NAP Extranet project, we started creating a file for 'business' logic and a file for View logic (which includes more visual Components) for each app module)

APPLYING THE SAME LOGIC:

```
// Available Component Definition
function AvailableComponent($context, handleAvailableListClick) {
    var compClasses = {
            ...
    };
    var compSelectors = {
            ...
    };

    // Private
    var $component = $context.find(compSelectors.component);

    // Public
    this.removeItem = function($item) {
            ...
    };
```

```
// ListChooser Page definition
function ListChooserPage() {
    var $page = $('#container');

    var myListComponent= MyListComponent.create($page);
    var availableListComponent = AvailableListComponent.create($page

 function handleAvailableListClick() {
    var $clickedEL = $(this);
    $.ajax(...);
    function onSuccess() {
        availableComponent.removItem($clickedEL);
        myListComponent.addItem($clickedEL.text());
    }
};
```

WHAT TO DO WITH THIS?

TEST *AVAILABLELIST* COMPONENT MORE EASILY

AT FIRST GLANCE:

- check if it's being correctly initiated:
  - if it finds the items throught the selectors
  - if the `removeItem` public method has the expected behavior
- check if the click event delegation is correctly set up

TESTING THE *LISTCHOOSERPAGE* COMPONENT:

- check if it initiates correctly
  - if searches (and finds) its selectors
  - if instantiates its dependencies
- if its click handler function is called when clicking its bound element and if it does what's expected:
  - ajax call with certain parameters and on success:
  - call availableComponent `removeItem`
  - call myListComponent `addItem`

IN THE END WE REALIZE THAT

- *List* component became responsible for their internal representational logic + internal jQuery stuff and for wiring up their event handlers - that's what we're going to test
- **Page** component is in charge of the page logic, making ajax calls, setting up the event handler functions, etc - that's what we're going to test

KEY CONCEPTS

TEST FILES

Normally all the test cases related to the same part of the app are grouped into the same test file. In our previous example, we'd have 3 test files (one forEach() component).

DESCRIBE

A `describe` is a function that defines a test suite.

SPEC === TEST

A spec contains one or more expectations that test the state of the code. A spec with all true expectations is a passing spec. A spec with one or more false expectations is a failing spec.

WHAT IT LOOKS LIKE

```
describe("the component/behavior we're testing", function() {
    var myComp;

    beforeEach() {
        myComp = new myComponent();
    }

    it("should return true when getBoolValue is called", function()
        var fnReturnValue = myComp.getBoolValue();

        expect(fnReturnValue).toBe(true);
    });
});
```

MATCHERS

HELPER FUNCTIONS USED IN EXPECTATIONS

```
.toEqual({yo: true}) // 'strict equal' and more general equals
.toBeUndefined()
.toHaveBeenCalled()
.toBeTruthy()
.toContain()
.toThrow(e)
.not.to....();
```

SPIES

Spies are utilities for stubbing any function and tracking calls to it and all arguments.

A spy only exists in the describe or it block in which it is defined, and will be removed after each spec.

HOW THEY LOOK LIKE

```
it("should call the myComp method", function() {
    spyOn(myComp, 'getBoolValue');
    myComp.methodThatAlsoCallsGetBoolValue();

    expect(myComp.getBoolValue).toHaveBeenCalled();
});
```

FIXTURES

On **AngularJS**, testing is given straight out-of-the-box, the framework itself can detect templates used in directives, **ngMock** module can inject and mock dependencies.

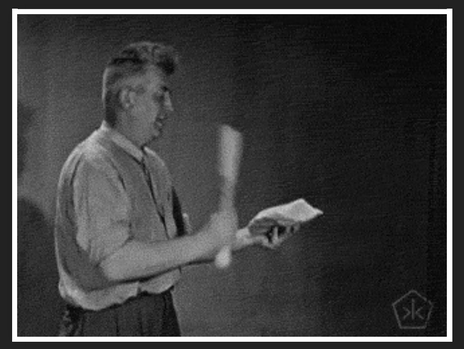
An initial approach is well documented here.

FIXTURES

On **jQuery** apps, the dev is more responsible for organizing the code so that it's testable.

Also, for testing components with DOM logic, it's necessary to inject HTML content into the tests so that jQuery has something to run against - **fixtures**.

TOOLS OF THE TRADE



- Karma - 'Spectacular' Test Runner for Javascript
- Test runners
  - mocha - test framework for JS, normally used with:
    - chai - assertion library (stuff lik `assert`, `should` and `expect`)
    - sinon - test spies, stubs and mocks
  - Jasmine - simpler solution (although less powerfull) gives you a behavior-driven development testing framework + `expect`, spies, etc in one package

- Jasmine-jQuery - set of matchers and fixture loaders for jquery
- Reporters
  - Karma Coverage - gives statement, line, function and branch coverage
  - and more...

LET'S DO SOME TESTING, THEN!

TESTING *LISTCHOOSERPAGE* COMPONENT

```
describe('ListChooserPage', function () {
    var ListChooserPage = require('../../components/ListChooserPageC
    var listChooser;
    (...)

  beforeEach(function () {
    loadFixtures('listChooserPage.html');
    myList = jasmine.createSpyObj('myList', ['addItem']);
    availableList = jasmine.createSpyObj('availableList', ['removeIt
  });

  it('should initiate correctly', function () {
    // should check if the ListChooserPage instantiates its dependen
    // set up spies
    spyOn(AvailableComponent, 'create');
    spyOn(MyListComponent, 'create');
```

TESTING *AVAILABLELIST* COMPONENT

```
describe('AvailableComponent', function () {
    var availableList;

    beforeEach(function () {
        loadFixtures('availableListFixture.html');
    });

    it('should initiate correctly', function () {
        // set up spies
        var $context = $('#jasmine-fixtures');
        spyOn($context, 'find').and.callThrough();
        spyOn(AvailableListComponent, 'create').and.callThrough();

        availableList = AvailableListComponent.create($context);
        expect($context.find).toHaveBeenCalledWith('.available');
    });
```

TYPICAL SETUP

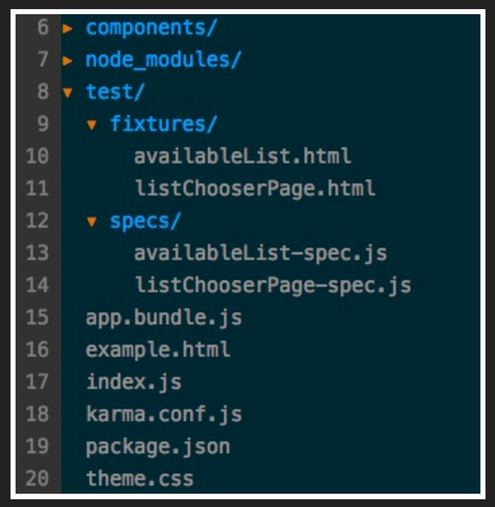
JQUERY PROJECT

```
11 ▾ js-unit-tests/
12   ▸ fixtures/
13   ▸ mocks/
14   ▾ specs/
15     ▸ behaviors/
16     ▸ modules/
17     ▸ services/
18   ▸ target/
19     karma.conf.js
```

ANGULAR PROJECT

```
12 ▾ test/
13   ▾ spec/
14     ▸ controllers/
15     ▸ directives/
16     ▸ services/
17     karma.conf.js
```

THIS DEMO PROJECT



ENTRY POINT: KARMA.CONF.JS

```
    // list of files / patterns to load in the browser
    files: [
        'node_modules/jasmine-jquery/lib/jasmine-jquery.js',
        'test/fixtures/*.html*',
        'test/**/*-spec.js'
    ],

    // frameworks to use
    frameworks: ['jasmine-jquery','jasmine', 'browserify'],

    // preprocess matching files before serving them to the browser
    preprocessors: {
        'test/**/*.js': [ 'browserify' ]
    },

    plugins: [
```

RUNNING TESTS

```
// on package.json
...
"scripts": {
     "test": "./node_modules/.bin/karma start karma.conf.js"
},

// on terminal
npm test
```

IT'S FLEXIBLE

REPORTING

Code coverage report for **All files**

Statements: **98.77%** (1124 / 1138) Branches: **100%** (651 / 651) Functions: **97.92%** (423 / 432) Lines: **98.6%** (988 / 1002) Ignored: none

File	Statements		Branches		Functions		Lines	
app/	100%	(63 / 63)	100%	(23 / 23)	100%	(25 / 25)	100%	(62 / 62)
app/components/alert-danger/	100%	(2 / 2)	100%	(4 / 4)	100%	(1 / 1)	100%	(1 / 1)
app/components/alert-success/	100%	(2 / 2)	100%	(4 / 4)	100%	(1 / 1)	100%	(1 / 1)
app/components/employee-account-details/	100%	(18 / 18)	100%	(16 / 16)	100%	(6 / 6)	100%	(16 / 16)
app/components/employee-authorizations/	100%	(21 / 21)	100%	(18 / 18)	100%	(8 / 8)	100%	(19 / 19)
app/components/employee-bank-details/	100%	(5 / 5)	100%	(6 / 6)	100%	(2 / 2)	100%	(3 / 3)
app/components/employee-contact-details/	100%	(5 / 5)	100%	(6 / 6)	100%	(2 / 2)	100%	(3 / 3)
app/components/employee-hourly-rate/	100%	(7 / 7)	100%	(10 / 10)	100%	(2 / 2)	100%	(5 / 5)
app/components/footer/	100%	(4 / 4)	100%	(4 / 4)	100%	(2 / 2)	100%	(3 / 3)
app/components/header/	100%	(8 / 8)	100%	(6 / 6)	100%	(4 / 4)	100%	(6 / 6)
app/components/modal-document/	100%	(34 / 34)	100%	(16 / 16)	100%	(11 / 11)	100%	(29 / 29)
app/components/modal-error/	100%	(6 / 6)	100%	(6 / 6)	100%	(2 / 2)	100%	(5 / 5)
app/components/modal-location/	100%	(29 / 29)	100%	(18 / 18)	100%	(6 / 6)	100%	(26 / 26)
app/components/modal-partner/	100%	(18 / 18)	100%	(10 / 10)	100%	(4 / 4)	100%	(15 / 15)
app/components/modal-position/	100%	(16 / 16)	100%	(8 / 8)	100%	(4 / 4)	100%	(13 / 13)
app/components/profile-completeness-bar/	100%	(5 / 5)	100%	(6 / 6)	100%	(2 / 2)	100%	(3 / 3)
app/core/config/	96.97%	(32 / 33)	100%	(14 / 14)	87.5%	(7 / 8)	96.97%	(32 / 33)

QUESTIONS?

`git checkout` this