BDD and TDD for AngularJS

Acceptance testing with CucumberJS and Protractor

Acceptance testing for AngularJS is done via the Protractor tool, which is a framework developed by the team behind AngularJS. It is worth noting that Protractor uses by default Jasmine as the testing framework and it was not until recently that CucumberJS was integrated to Protractor.

You should also be aware of the fact that CucumberJS does not cover all the features provided by the standard Cucumber (particularly those that have been deprecated after the controversy behind BDD and Cucumber). Nevertheless, CucumberJS is already good enough for our purposes.

Let's start by copying the following Gherkin user story in the file `test/features/filt er_movies.feature`

```
Feature: display list of movies filtered by MPAA rating

  As a concerned parent
  So that I can quickly browse movies appropriate for my family
  I want to see movies matching only certain MPAA ratings

  Background: movies have been added to database

    Given the following movies exist:
      | title                   | rating | release_date |
      | Aladdin                 | G      | 25-Nov-1992  |
      | The Terminator          | R      | 26-Oct-1984  |
      | When Harry Met Sally    | R      | 21-Jul-1989  |
      | The Help                | PG-13  | 10-Aug-2011  |
      | Chocolat                | PG-13  | 5-Jan-2001   |
      | Amelie                  | R      | 25-Apr-2001  |
      | 2001: A Space Odyssey   | G      | 6-Apr-1968   |
      | The Incredibles         | PG     | 5-Nov-2004   |
      | Raiders of the Lost Ark | PG     | 12-Jun-1981  |
      | Chicken Run             | G      | 21-Jun-2000  |

    And  I am on the RottenPotatoes home page

  Scenario: restrict to movies with 'PG' or 'R' ratings
```

```
    When I check the checkbox for rating 'PG'
    And I check the checkbox for rating 'R'
    And I uncheck the checkbox for rating 'G'
    And I uncheck the checkbox for rating 'PG-13'
    Then I should see movies with ratings 'PG', 'R'
    And I should not see movies with ratings 'PG-13', 'G'

  Scenario: all ratings selected
    When I check the checkbox for rating 'PG'
    And I check the checkbox for rating 'R'
    And I check the checkbox for rating 'G'
    And I check the checkbox for rating 'PG-13'
    Then I should see 10 movies
```

Now, let's configure protractor. Copy the following snippet to the file `ProCukeConf.js` (you can use whatever name you like).

```
require('coffee-script');
exports.config = {
    seleniumAddress: 'http://localhost:4444/wd/hub',
    framework: 'cucumber',
    specs: [
        'test/features/*.feature'
    ],
    capabilities: {
        'browserName': 'chrome'
    },
    baseUrl: 'http://localhost:9000',
    cucumberOpts: {
        require: 'test/features/steps/*_steps.coffee',
        format: 'pretty'
    }
};
```

Before going on, we need to setup Protractor and friends. Let's start by installing Protractor and CucumberJS:

```
npm install -g protractor

npm install -g cucumber
```

> Just keep in mind that if you are using a UNIX based operating system, you have to run the command above using `sudo`.

By default, Protractor expects javascript. To add support to coffescript, we have to install a package to our project (note that this is local to our project). By the same token, let's install the packages `chai` and `chai-as-promised` that will provide a set of matchers that are easy to read (intuitive).

```
npm install -D coffee-script
npm install -D chai-as-promised
```

Protractor runs its tests directly on a browser. To that end, Protractor interfaces with a middleware (i.e., selenium), which needs some additional steps for configuration. First, we have to install a driver for interacting with an actual browser. I will assume that you have chrome installed in your computer. If so you can execute the following command:

```
webdriver-manager update --chrome
```

We are now ready to start the BDD cycle. In three different terminals you have to execute the following commands (one per terminal):

```
grunt serve

webdriver-manager start

protractor ProCukeConf.js
```

As expected, protractor on behalf cucumber reports that no step is defined and provides some snippets to start with (in Javascript, though). However, we can use cucumber to get coffeescript snippets:

```
cucumber-js test/features/filter_movies.feature --coffee
```

Based on cucumber snippets, we can write our initial set of cucumber steps. The file, called `test/cucumber/steps/filter_movies_steps.coffee`, should look like:

```
chai = require 'chai'
chaiAsPromised = require 'chai-as-promised'
chai.use chaiAsPromised
expect = chai.expect
By = `by`

module.exports = ->
  @Given /^the following movies exist:$/, (table, callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @Given /^I am on the RottenPotatoes home page$/, (callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @When /^I check the checkbox for rating 'PG'$/, (callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()
```

```
  @When /^I check the checkbox for rating 'R'$/, (callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @When /^I uncheck the checkbox for rating 'G'$/, (callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @When /^I uncheck the checkbox for rating 'PG\-(\d+)'$/, (arg1, callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @Then /^I should see movies with ratings 'PG', 'R'$/, (callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @Then /^I should not see movies with ratings 'PG\-(\d+)', 'G'$/, (arg1, call
back) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @When /^I check the checkbox for rating 'G'$/, (callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @When /^I check the checkbox for rating 'PG\-(\d+)'$/, (arg1, callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()

  @Then /^I should see (\d+) movies$/, (arg1, callback) ->
    # express the regexp above with the code you wish you had
    callback.pending()
```

In contrast to Rails/Cucumber, in our setting AngularJS/CucumberJS we don't usually have access to the backend's database. Therefore, we cannot store the sample movies into the database. Instead, we have to inject a mock AngularJS service, itself in javascript alas. Copy the following snippet to implement the first step:

```
  generateMockServiceScript = (movies) ->
    script = '''
    var app = angular.module('coffee1AppMock', []);
    app.service('MoviesService', function() {
        var movies = [
'''
    for movie in movies
      script += "{title: '#{movie.title}', rating: '#{movie.rating}', release_
date: '#{movie.release_date}'},"

    script += '''
        ];
```

```
        this.all = function() { return movies; };
        this.add = function(movie) { movie.id = movies.count; movies.push(movi
e); };
      });
'''

  @Given /^the following movies exist:$/, (table, next) ->
    script = generateMockServiceScript(table.hashes())
    browser.addMockModule 'coffee1AppMock', script
    next()
```

You can easily verify that the mock service has the same structure than the `MoviesService` that we used in the initial implementation of our rottenpotatoes. You can also see that we use a for loop to generate object literals for each movie, all of them stored in the array `movies`. It is worth noting that we defined the service as a string that will be injected by protractor in the browser. By the way, `browser` is an object provided by Protractor. This implementation would override the existing implementation of `MoviesService`.

It is also important to note that the second parameter received by the function in `@Given`, that I have renamed as next, is called at the end of the function, which will mark to cucumber that it is time to continue with the next step.

```
  @Given /^I am on the RottenPotatoes home page$/, (next) ->
    browser.get '#/movies'
    next()
```

The step is rather simple, it consists only in “opening” the page `#/movies`. Once again, we use `browser` (provided by Protractor) and the method `get` can be mapped to a HTTP GET.

There are several steps in the user story referring to checkboxes in the user interface that can be used for selecting the rating to be filtered/kept. Copy the snippet below and remove all the steps that are subsumed.

```
  @When /^I (.*)check the checkbox for rating '(.*)'$/, (uncheck, rating, next
) ->
    checkbox = element(By.id('rating_' + rating))
    checkbox.isSelected().then (selected) ->
      if (selected and uncheck == 'un') or (not selected and uncheck != 'un')
        checkbox.click()
      next()
```

What is very important is to note the construction `element(By.id('rating_' + rating))` that let us access to the DOM element identified by a string of the form `rating_G`. Note that in this way we are fixing a requirement to be met on the HTML by the web page designers: there must be checkboxes using identifiers of that form.

Promises

One of the challenges to face when testing a javascript web front-end application stems from the fact interactions happen asynchronously. As a way of example, consider Google maps, the response to a simple query would usually take a little while. To cope with asynchrony, we will adopt the notion of promises: any call to an asynchronous function returns immediately not with the expected value but with a promise that such a value (or an error) will eventually be returned. In the code above, when we query whether the checkbox is selected or not we doing so via a promise. The function `then` registers two functions for the positive outcome and the other one (which is optional) for error handling. Therefore, once the status of the checkbox has been determined in the browser the function handling the positive outcome is called with the parameter `selected`. Afterwards, depending on the value of `selected` we will click the checkbox or not (e.g., if the checkbox was already checked).

Completing scenario “all ratings selected”

When we follow the BDD-TDD cycle, this would be the time to start writing the unit test (TDD) to guide the implementation. This time, we will short-circuit the cycle and we will add some code to complete the second scenario “all ratings selected”.

First, we have to modify the view so as to include one checkbox per movie rating. Copy the following snippet in the file `views/movies/index.html`:

```
<div>
    <label ng-repeat="rating in ['G', 'PG', 'PG-13', 'R']">
        <input id="rating_{{rating}}" type="checkbox"/>{{rating}}
    </label>
</div>
```

You should already understand the directive `ng-repeat`: it iterates over a collection (an array in this example) and creates a label and input element. Note that we use `rating` to specify the text to be rendered as label and also to specify the identifier of the input element. This is done via `rating_{{rating}}`. Note that this is in line with what we specified in the cucumber steps.

With this changes, you will have several steps in green. Please remember that our goal is just to pass one scenario of the cucumber specification, even if at this moment no movie filtering is implemented.

To complete the scenario “all ratings selected” we only need to count the number

of movies listed in the index page. To this end, we will adopt the following convention: we will annotate every row corresponding to a movie in the table with the CSS class ".movie-info". In this case we cannot use the element identifier because it must be unique. With this idea in mind we can complete the cucumber step as follows:

```
    @Then /^I should see (\d+) movies$/, (number_of_movies, next) ->
      allMovies = element.all(By.css('.movie-info'))
      expect(allMovies.count()).to.eventually.equal(parseInt(number_of_movies))
        .and.notify(next)
```

Please note that the idea is to query all DOM elements annotated with class ".movie-info". The we can specify our expectation as a promise: we expect the middleware will eventually determine the number of matched elements and that this number is equal to the integer value of `number_of_movies`. As the expectation will be evaluated in the future we delegate to this promise the responsibility of notifying the completion (via `.and.notify(next)`).

Of course, if you run protractor the test would fail because we haven't annotated the table rows with the class `movie-info`. Modify the line that opens up the table row displaying the information about a movie as follows:

```
      <tr class="movie-info" ng-repeat="movie in movies">
```

With this last addition, we completed the second scenario.

TDD with Karma and Jasmine

We will now write a unit test for guiding the implementation of filter, that will restrict the list of movies according to their rating. To this end, we will use Jasmine a testing framework for javascript that is very close in syntax to RSpec.

We have to install some other tools and modify the configuration files. IN a terminal window, run the following commands:

```
npm install -D karma-jasmine
npm install -D karma-phantomjs-launcher
npm install -D karma-coffee-preprocessor
```

For convinience, let us also install the Karma command line:

```
npm install -g karma-cli
```

Karma is a node.js package that works as test runner. Therefore, we need to

modify the configuration file for karma. Open the file `karama.conf.js` and add the following lines just before the section `files:`

```
    preprocessors: {
      '**/*.coffee': ['coffee']
    },
    coffeePreprocessor: {
      options: {
        bare: true,
        sourceMap: false
      },
      transformPath: function(path) {
        return path.replace(/\.coffee$/, '.js');
      }
    },
```

The above instructs Karma to precompile all the coffeescript files into javascript before performing the test.

Scroll down a bit the same file and change the following parameters as shown below:

```
    autoWatch: true,
    browsers: ['PhantomJS'],
```

The first parameter, namely `autowatch`, instructs Karma to monitor the coffeescript files. When Karma detects a change it will automatically run the test once more (think about this feature as if you were using `autotest` in the Rails context). The second parameter, that is `browsers`, specify the list of browsers to be considered when running the test. You can specify several browsers, but for our purposes we will only use PhantomJS which is a simulated browser that does not require to open anything in the screen, and therefore very convenient for development.

We are ready to start working. Copy the following snippet into the file `test/spec/controllers/movies_controllers.coffee`.

```
'use strict'

describe 'Controller: MoviesIndexController', ->
  it 'should fail because it is a contradiction', ->
    expect(true).toBe false
```

You can easily verify that the above spec should fail. The purpose of such a spec is to verify if our configuration is correct. Let's now launch karma:

```
karma start karma.conf.js
```

If everything is correct, you should get 1 failed test: "Expect true to be false".

Please note that Karma keeps running. Change the specification to something correct.

Adding a filter test

Angular controllers provide a kind of placeholder for data to be render and also some functions to interact with such data. Think about the “MoviesIndexController” in our running example. This controller already exports “movies” and is also the place where we should implement the filtering logic.

Let’s start with a simple test. Copy the following snippet to your Jasmine specification (replace entirely the previous specification).

```
'use strict'

describe 'Controller: MoviesIndexController', ->
  MoviesIndexController = {}
  scope = {}

  beforeEach module 'rottenpotatoesApp'

  beforeEach inject ($controller, $rootScope) ->
    scope = $rootScope.$new()
    MoviesIndexController = $controller 'MoviesIndexController', {
      $scope: scope
    }

  it 'should export a list of movies', ->
    expect(scope.movies).toBeDefined()
```

The first `beforeEach` block connects the test with the AngularJS module representing our application. The second `beforeEach` block instantiates the `MoviesIndexController` and passes a mock `scope` that we are going to use to read/write data handled by the controllerand to verify the results of any computation performed on such data.

If you turn now your attention to the only test case in the specification you will notice that we are checking if the controller exports a variable `movies`, the one that holds the list of movies. Of course, this test should pass without any problem.

Let’s first add some fixtures for the test. I will use the same list of movies evoked in the user story as fixtures for our text. Replace the only `it` block that we have by the following snippet:

```
  sampleMovies = [
    {title: 'Aladdin',                rating: 'G'}
    {title: 'The Terminator',         rating: 'R'}
    {title: 'When Harry Met Sally',   rating: 'R'}
```

```
      {title: 'The Help',                rating: 'PG-13'}
      {title: 'Chocolat',                rating: 'PG-13'}
      {title: 'Amelie',                  rating: 'R'}
      {title: '2001: A Space Odyssey',   rating: 'G'}
      {title: 'The Incredibles',         rating: 'PG'}
      {title: 'Raiders of the Lost Ark', rating: 'PG'}
      {title: 'Chicken Run',             rating: 'G'}
    ]

    beforeEach ->
      scope.movies.push sampleMovies...

    it 'should export a list of movies', ->
      expect(scope.movies).toBeDefined()
      expect(scope.movies.length).toBe(10)
```

As you infer from the code above, we are going to set `scope.movies` first as an empty array and then we are going to copy all the movies in `sampleMovies`. This will be executed before every `it` block. You can also notice that we have added a new expectation: we should have now a list of 10 movies.

This is just the setup part. We can now start with real stuff :P At this moment, we have modified the view to include a group of checkboxes one per movie rating. However, this is disconnected from the controller. Therefore, we have to wire them up. We will assume that the controller exports an object (a hash?) with the rating and the status of the corresponding checkbox. Let's add a new `it` block

```
    it 'should export an object with the status of rating checkboxes', ->
      expect(scope.selectedRatings).toBeDefined()
```

As we go red with our test, we have to take a time to fix the controller `MoviesIndexController`. Add the following line to the corresponding file.

```
    $scope.selectedRatings = {'G': true, 'PG': true, 'PG-13': true, 'R': true}
```

This should be enough to get the test accepted.

Note that we initialized the object such that all the ratings are displayed. Afterwards, the user can discard/select some of the ratings by means of the checkboxes in the view. In the context of this controller test we are bypassing the view. However, we can programatically change the object `selectedRatings` in the controller without requiring any manipulation of elements in the view.

To implement the expected behavior, filter out movies depending on their rating we will introduce a new concept: AngularJS filter. AngularJS provides a set predefined filters. For instance, we can use a filter to format the release date by changing the corresponding line in `views/movies/index.html` as follows:

```
          <td>{{movie.release_date | date:'mediumDate'}}</td>
```

Behind scenes, AngularJS calls the filter `date` with two parameters: a date (i.e., `movie.release_date`) and a format (i.e., `'mediumDate'`).

Another type of filter can be implemented via a function declared in the controller. Roughly, we want to change the element that iterates over the list of movies and creates a new row in the table to look something like: `<tr class="movie-info" ng-repeat="movie in movies | filter:byRating">`, where `byRating` is a function implemented in the controller and that takes a movie as input and returns `true` if the movie should be rendered and `false` otherwise. The movie corresponds to the one that we are selecting during the iteration and the filtering depends on the status of the checkboxes associated to each rating.

With all the above elements, we can now formulate our test.

```
    it 'should filter out "G" rated movies', ->
      scope.selectedRatings['G'] = false
      for movie in sampleMovies
        expect(scope.byRating(movie)).toBe(scope.selectedRatings[movie.rating])
```

Initially, all the checkboxes associated to ratings are checked, meaning that `selectedRatings` is set to `{'G': true, 'PG': true, 'PG-13': true, 'R': true}`. In the test, we simulate the case were the user unchecks the checkbox for movies rated `'G'`. Finally, we iterated over the list of movies and we call the filter `scope.byRating(movie)` and set our expectation such that the function returns true or false according to the movie rating.

Sure enough our test will fail. However, the code to be added to the controller is clearly hinted withing the test:

```
    $scope.byRating = (movie) ->
      $scope.selectedRatings[movie.rating]
```

Believe it or not, we are done with the controller.

Connecting all the pieces together

Now we can go back to the acceptance test to complete the missing steps. The good news is that we can simplify the steps so as to have only one cucumber step. Therefore, replace the two pending steps with the code below.

```
    @Then /^I should (.*)see movies with ratings (.*)$/, (not_see, ratings, next
  ) ->
      ratings = ratings.replace(/[\s\']/g, '').split(',')
```

```
    displayed_ratings = element.all(By.css('.movie-rating'))
      .map((elm) ->  elm.getText())

    if not_see is 'not '
      expect(displayed_ratings).to.eventually.not.include.members(ratings)
        .and.notify(next)
    else
      expect(displayed_ratings).to.eventually.include.members(ratings)
        .and.notify(next)
```

As for the one of our previous examples, we will use a CSS selector to gain access to the information about the movie rating, i.e., `element.all(By.css('.movie-rating'))`. But this time we do further, we iterate over the list of DOM elements and create an array containing only their text (this is done by means of the function `map()`).

Of course, our test will fail because there is no element on our HTML with the class `movie-rating`. Change the corresponding line in the file `views/movies/index.html` as follows:

```
        <td class="movie-rating">{{movie.rating}}</td>
```

Finally, in our cucumber step we use the nice matchers `.to.eventually.include.members()` and `.to.not.eventually.include.members()` provided by ChaiJS to compare the contents of the arrays `ratings` and `displayed_ratings`. Clearly, our test will fail because we haven't filtered out any movie.

We first need to connect the rating checkboxes in the view with the variable `selectedRatings` exported by the controller. Replace the corresponding lines of code in the file `views/movies/index.html` as follows:

```
    <label ng-repeat="(rating, status) in selectedRatings">
        <input id="rating_{{rating}}" type="checkbox"
               ng-model="selectedRatings[rating]"/>{{rating}}
    </label>
```

Note that the directive `ng-repeat` iterates over the variable `selectedRatings` and not longer over a constant array. Additionally, we have connected the checkbox with `selectedRatings[rating]` via the `ng-model` directive. This gives us the opportunity to keep synchronized both the view and the controller.

The only missing piece is the filter. Surprisingly, integrating the filter requires only a small change:

```
    <tr class="movie-info" ng-repeat="movie in movies | filter:byRating">
```

Just to double check, run protractor. The test should be all green!!!

> Written with StackEdit.