jamk.fi

Unit testing of AngularJS

A look into writing tests for web application

Joel Karttunen

Bachelor's thesis
May 2016
Technology, Communication and Transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Karttunen, Joel	Bachelor's thesis	05 2016
		Language of publication: English
	Number of pages: 30	Permission for web publication: (X)

Title of publication

Unit testing of AngularJS

A look into writing tests for web application

Degree programme

Software Engineering

Supervisor(s)

Rantala Ari

Assigned by

Protacon Solutions Oy

Abstract

This task was assigned by Protacon Solutions Oy with the objective set to study and implement unit testing in AngularJS JavaScript framework as part of the development process of the customer's application.

The thesis focuses on studying the AngularJS framework and if and how its design philosophy complements the unit testing principles. The focus was on studying unit testing via developing unit tests iteratively onto the application created beforehand.

The tests were created with Jasmine testing framework, a tool recommended by the AngularJS development team. Other testing frameworks were studied before the thesis, however Jasmine was chosen to be examined more closely.

The thesis results in several unit tests created for the customer's application, and knowledge was provided for the company about the unit testing of a modern JavaScript web application to be distributed among the co-workers in the company.

In addition, the backbone for developing and expanding the unit tests for the application and other future applications developed with AngularJS was laid down. The tests were created with trial-and-error method, and best practices were recorded to be expanded upon, determined by the nature of the application and what was best suited for it.

Keywords/tags

JavaScript, AngularJS, Unit testing

Miscellaneous

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Karttunen Joel	Opinnäytetyö, AMK	05 2016
	Sivumäärä 30	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: (X)

Työn nimi
AngularJS yksikkötestaus
Yksikkötestien teko web-sovellukseen

Tutkinto-ohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Ari Rantala

Toimeksiantaja(t)
Protacon Solutions Oy

Tiivistelmä

Opinnäytetyön toimeksiantajana toimi Protacon Solutions Oy. Tehtävänä oli tutkia ja kehittää yksikkötestejä AngularJS- sovelluskehyksellä kehitettyyn web-pohjaiseen asiakkaalle toteutettuun sovellukseen.

Työ keskittyi AngularJS-sovelluskehyksen ominaisuuksien tutkimiseen ja miten niitä pystyi hyödyntämään sovelluksen yksikkötestejä kehitettäessä, ja kuinka AngularJS sovelluskehyksen rakenne tuki yksikkötestien suunnitelun teesejä.

Testit kehitettiin käyttäen Jasmine-testaussovelluskehystä, mikä on AngularJS-kehittäjien suosittelema työkalu testien kehitykseen. Muita testaussovelluskehyksiä tutkittiin pikaisesti ennen työn aloittamista, mutta niistä päätettiin luopua ennen työtä.

Tuloksena syntyi useita yksikkötestejä asiakkaan AngularJS-sovellukseen, ja osaamista ja tietotaitoa yritykselle yksikkötestien tekemisestä AngularJS-sovelluskehyksellä kehitettyyn sovellukseen. Tarkoituksena on jakaa tätä tietotaitoa eteenpäin.

Lisäksi tuloksena oli pohjustus uusien testien tekemiseen ja vanhojen lisäkehitykselle asiakkaan sovelluksessa. Testit kehitettiin yrityksen ja erehdyksen kautta, josta voi jatkaa parhaaksi todetuin menetelmin testien kehitystä. Testit suunniteltiin ja toteutettiin juuri kyseiseen sovellukseen sopiviksi.

Avainsanat (asiasanat)
JavaScript, AngularJS, yksikkötestaus

Muut tiedot

Contents

Acronyms

TDD	Test Driven Development
BDD	Behaviour Driven Development
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheets
PHP	Hypertext Preprocessor

1 Introduction

Testing is a key part of application development and should be a standard in every company wanting to succeed in today. However, are there applications, where some elements require more attention for testing than others?

In the past, web applications were mostly structured by processing every request on the server side, and afterwards views were generated for the web browser to display. The client side interactivity was provided by some small JavaScript snippets, yet the core logic of the whole application rested within the server side code (Graetz, A brief history of single-page applications.)

What has changed? Today, web application design has moved increasingly into depending more on client side execution for logic, with popular JavaScript libraries beginning from such as JQuery, Knockout, and moving to frameworks like AngularJS and EmberJS. The server side architecture can be as simple as some lightweight PHP framework with REST architecture, while the client side web page handles not only display logic with views, but also the application's business logic (Graetz, A brief history of single-page applications.)

What about testing? Writing unit tests can be viewed as a norm in standard server-side programming languages such as C# and PHP. They have years of development with them and standardized tools and methods to write efficient unit tests for the application. But when the main business logic moves to the browser side of the application, what then? How, and what do the tests need to accomplish the task they are set to do? What tools to use? Do they differ, and if so, how?

Of course, web applications have been tested before, and probably also successfully. But do these new shiny frameworks offer easy and simple implementation of unit and other type of tests into the software?

AngularJS framework is described to be created with testability in mind (AngularJS: Developer Guide, Unit testing.) As a new developer looking into the framework and the promises it is offering, in this document methods used to test a fresh AngularJS application are described, part of the development of which the author of the thesis was.

The work was done in assigning company, Protacon Solutions Oy. The tests were created using the tools and environments provided by the company.

2 Specifying the problem

The process described in this document started while the author worked as developer in a newly founded AngularJS project. The whole AngularJS framework had to be learned it mostly on-the-go. As the framework is created with testability in mind – or so the development team behind it says – the idea of studying these claims emerged to see if they were true. As the company had not really done any unit testing on client side of a web application project, the theme for this study was set.

As a rather new developer in the field of programming, unit testing concepts and methods were much to none existent, so at start I the unit testing of other applications was studied, mostly some backend unit tests created for .Net project. As the knowledge of unit testing principles began to grow, it was time to set those principles to the previously created AngularJS application, and do the testing in Angular way.

The AngularJS part of the testing was rather straight forward. Usually the tests were written component by component, improve and iterate over and over. As the REST architecture used in the application makes the service calls rather similar, some utility components were needed to be used only for the tests.

While testing the application, some rather good points about the structure of the application, and what was wrong with it, emerged. As the AngularJS framework was new and the understanding of it in the start was rather limited, some controllers and other components were bloated and became difficult to test afterwards. In this project, these too-big-to-fail components were a good example of why unit testing should be done before, or at the same time as the development of the application, and TDD, or BDD can be a good thing.

Most of the unit testing carried out in the application was mocking and isolating the components from each other, as they should be. Of course, there are other points of

view for what unit tests should do, as should they interact more broadly with other components of the application; however, this case study focuses only on the isolated, smallest unit testing cases of the application.

This study consists of three larger parts listed as follows.

First, there is some background information about the AngularJS framework and its structure, components and usage. As the framework can be rather odd-looking for a developer with background in for example jQuery library, some examples are made and key components are described. Afterwards the philosophy behind unit testing is explained.

Second, the tools used in the unit testing of the AngularJS application are explained.

Finally, the results of how testable the AngularJS application is are given a thought, along with any notices of how one should structure their components to be more testable.

3 AngularJS framework

3.1 Introduction to AngularJS

AngularJS is a JavaScript framework created and managed by Google. Its main purpose is to bring useful architectural features from server side programming languages to client side programming. AngularJS frameworks several key points that make it good can be described as follows (Freeman 2014, 3):

- Extendable: AngularJS is made to be extendable. It is easy to develop new features to your existing AngularJS application.
- Maintainable: AngularJS applications are easy to debug and fix, which makes them easier to maintenance.
- Testable: AngularJS supports unit and end-to-end tests, which helps in finding bugs and design flaws.
- Standardized: AngularJS extends HTML5 technologies and uses the capability of modern browsers, which means one can write the Angular app and enhance it with other outside frameworks and libraries as you desire.

The framework is also described as following: AngularJS was created to solve the problem with creating dynamic views with HTML, which was not really designed to be used in that. AngularJS aims to solve these problems by extending the HTML language with framework specific attributes, rather than abstracting the HTML, CSS or JavaScript code to control the view like some other libraries tend to do (AngularJS: Developer Guide, Introduction.)

AngularJS is a versatile and powerful tool, which gives the developer a chance to quickly create reusable components and views. One objective that should be taken into consideration, at least when starting to use the Angular framework, is to learn to code components in the Angular way, which can be very different from other JavaScript libraries developer may have used in the past.

3.2 Structure of AngularJS framework

3.2.1 Dependency Injection

"*Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.*

The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested." (AngularJS: Developer Guide, Dependency Injection.)

Dependency injection in AngularJS framework enables the different components, such as controllers or services to require other components to be injected to the current scope of the component. Dependency injection in AngularJS consists of two services, the *$injector* and *$provide*.

$provide service handles the requests for Angular services, factories, values etc. All the components listed are in fact just shortcuts to the *$provide* service, which is called when the component is required in the application.

$injector service is responsible for creating the instances for components, such as services that the *$provide* gives. One single $*provide* instance is created for the whole AngularJS application and the it injects the requested components as needed with it's *$get* method. Injectable functions consist of service, directive, filter and factory functions (AngularJS Git hub wiki, Understanding Dependency Injection.)

3.2.2 Scope

The AngularJS's scope is a centre part for the framework. Scope's job is to bind the code behind logic of the controller into the view to display the data to user. This is called data-binding. Each data binding to scope adds a watcher function to the AngularJS's core $watch list. The $watch lists contain every binding the page has, and the list is used to update the actual displayed data during the $digest loop, if the value that has been watched is changed from the previous iteration of the loop. This chain of events is called Dirty Checking.

Objects and functions are bound to the $scope inside the controller function. The controller to be bound to view is described in the view template using the custom *ng-controller* attribute. The data-binding to $scope variables can be either one or two way. In one way binding, the data required to display is checked only once, and changes to the value do not change in the dirty checking part of the process. (AngularJS: Developer Guide, Scopes.)

3.2.3 Controller

AngularJS controller is simply an ordinary JavaScript function, which is injected to web pages DOM. When a new controller is created, a new child $scope object is also created and attached to the controller.

AngularJS controller's job is to control the interaction with user's input and the view. In MVC model, controllers usually land between the model and the view. Controllers also implement the applications business logic. Controllers should be kept as simple as possible so that the logic contained within controller doesn't become too difficult to handle. Simple controllers are easier to track and change through the whole application.

Key points what and what not a controller should do are following (AngularJS: Developer Guide, Controllers):

- Initialize the $scope object and its state
- Add functionality to the $scope object

What it should not do:

- Manipulate DOM
- Format user input
- Filter output
- Share state between other controllers
- Manage the creation of other components

Manipulation of DOM in AngularJS is a task for directives. AngularJS contains also data binding to bind $scope data to templates. User input formatting should be done

through AngularJS' form controls. Filters are used to control the output. Angular services are used to share states between components.

When a controller is kept simple, it is easier to write unit tests for it, and testability is one of the key design points of AngularJS. A controller is created as described in Figure 1.

```
/**
 * Controller for single Event
 */
angular.module('Nestor.Event.Controllers')
    .controller('EventController', [
        '$scope', '_event',
        function ($scope, _event) {
            $scope.event = _event;
        }
    ]);
```

Figure 1: Defining a controller

In the figure above, a new controller is defined to a module *Nestor.Event.Controllers* which can be thought as a namespace where the module is. The module can contain several controllers, and after the module definition, a controller is created by passing it's name as a first parameter. The $scope service which is built into AngularJS itself is always injected to the controller to be able to make two way data-bindings with the view. A _event object is also injected to the controller, which contains some data about the event.

After the initialization, the injected _event is assigned to the *$scope.event* variable and can be displayed in the view.

The structure of controller, and other AngularJS components, can be intimidating for developers new to AngularJS.

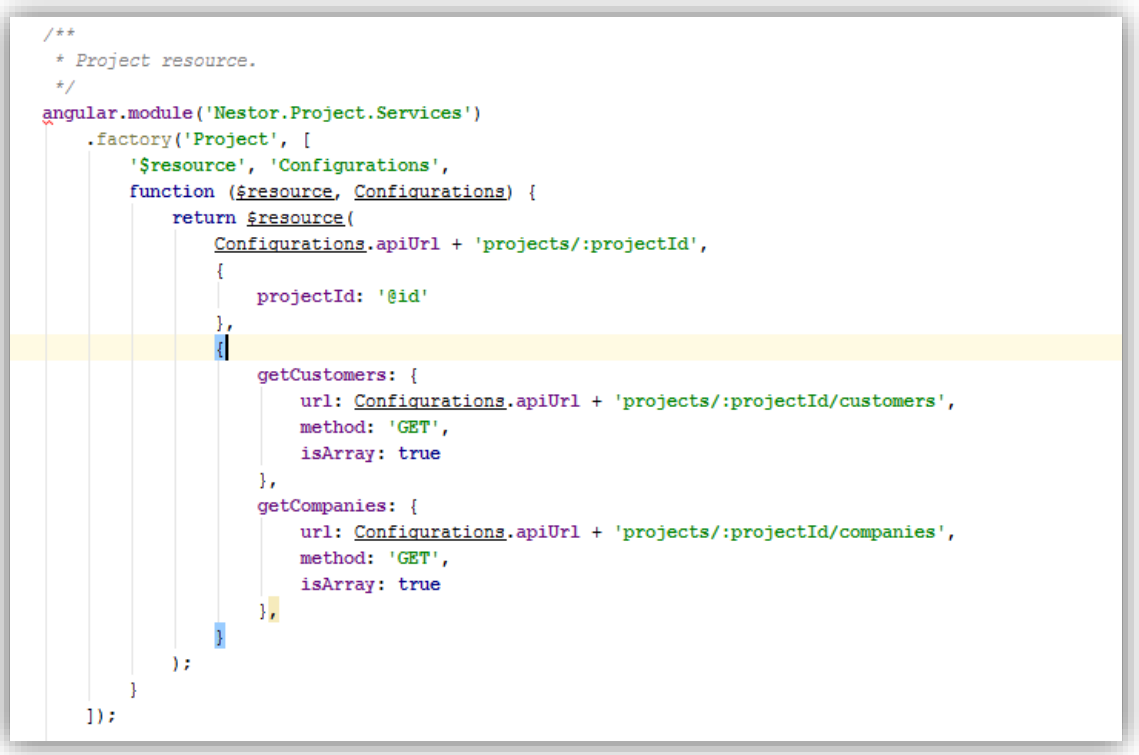
3.2.4 Service

AngularJS services are JavaScript objects that are injected to other angular components by the dependency injector. Services are used to share application state between components. Services can be separated to two categories, services and factories. Both are very similar and the only difference between them is how they are instantiated (AngularJS: Developer Guide, Services.)

Angular services are singleton objects, which means that every usage of the service is referred to single service object, which is created by the service factory. In addition, all services are only instantiated when they are needed for the first time. In other words, they are lazy. Like with controllers, services can also be injected with other services. Dependency injection with services makes the application's logic loosely coupled, and its benefits for one are to simplify unit testing of the functions (Cooper 2013).

Services are created similar to the controller. When a service is created, its factory function is registered to angular application module, and it is instantiated once it is needed somewhere in the application.

An example of service could be a one that implements REST –architectural methods GET, POST, PUT and DELETE for an object by wrapping it into the built-in AngularJS http service (Figure 2).



```
/**
 * Project resource.
 */
angular.module('Nestor.Project.Services')
    .factory('Project', [
        '$resource', 'Configurations',
        function ($resource, Configurations) {
            return $resource(
                Configurations.apiUrl + 'projects/:projectId',
                {
                    projectId: '@id'
                },
                {
                    getCustomers: {
                        url: Configurations.apiUrl + 'projects/:projectId/customers',
                        method: 'GET',
                        isArray: true
                    },
                    getCompanies: {
                        url: Configurations.apiUrl + 'projects/:projectId/companies',
                        method: 'GET',
                        isArray: true
                    },
                }
            );
        }
    ]);
```

Figure 2: A Angular service using $resource service to make calls to backend

3.2.5 Directive

Directives are one of the core functionalities in AngularJS, used to make basic HTML richer by injecting custom properties to common HTML elements. Directives can be used to separate common, repeatable parts of the application, for example lists that require some kind of order or pagination logic. They are 'compiled' - as worded by the Angular development team – when forming the web page's DOM. While the compiling is not the same thing in a sense as compiling in programming languages like C, the process shares some similarity with it. Compiling can be described as follows: "For AngularJS, "compilation" means attaching directives to the HTML to make it interactive. The reason we use the term 'compile' is that the recursive process of attaching directives mirrors the process of compiling source code in compiled programming languages." (AngularJS: Developer Guide, Directives.)

Directives can have a controller defining their own business logic, and like other Angular components, they can be injected with different services. Directives can contain their own HTML templates, which is useful when separating larger HTML pieces into smaller ones. Smaller, isolated pieces of application are usually easier to unit test, as the logic contained within tends to be simpler.

When directives are created, the controller functions like a regular AngularJS controller, with one larger difference. Directives can create their own isolated $scope object, or inherit the scope object from the parent controller where they are defined. Isolated scopes are usually the way to go, because directives can also be injected with parameters from the parent $scope. This keeps the contained logic simpler and the local $scope object does not get polluted by variables it does not care from the containing parent (AngularJS: Developer Guide, Directives.)

New directive is defined as demonstrated in Figure 3:

```
/**
 * Directive that handles articles list functionality.
 */
angular.module('Nestor.Article.Directives')
    .directive("articlesList", [
        function () {
            return {
                restrict: 'A',
                templateUrl: '/Nestor/article/partials/article_list.html',
                replace: true,
                scope: true,
                controller: 'ArticleListController'
            };
        }
    ]);
```

Figure 3: Defining a new directive

The controller can be implemented at the same time as directive, or the controller's name can be passed as a parameter to allow the AngularJS to resolve it. The t*emplateUrl* parameter specifies the path to html file which contains the view part of the directive. The directive is used within HTML as follows (Figure 4):

```
<div class="col-xs-4 frontpage-column">
    <div data-articles-list=""></div>
</div>
```

Figure 4: Injecting directive to view template

4 Software testing

4.1 Introduction to software testing

Software is tested to evaluate and validate that the software created in the process is what was originally wanted and ordered. Software testing qualifies that the software has the required standards and meets the requirements it was set to do. Testing does not limit to a single phase in software development, but is rather a continuous process that includes developers testing of codebase via unit-, integration- and acceptance testing, and also a more abstract testing documentary such as testing plans.

Testing can be seen as the final stamp of approval for the software created as when all the requirements are met and the tests created for them pass, the software is done. Of course, no software can be 100% tested and bug-free, however the large scale of testing minimizes the risks and needs to do more costly repairing processes afterwards in the maintenance phase of the software. It is estimated that the amount of money lost because of insufficient testing in year 2002 in the US was 59.5 billion dollars (Kasurinen 2013, 11).

Software testing is not a new idea in the software industry risen in recent years. The concepts of testing have been around as long as complex programs were beginning to surface. As the level of complexity in software has increased, so has the possibility for human error, thus creating bugs. Bugs are an error in software created by the developer, as there are no ‘hardware’ defects in the code itself unlike in real word applications. The software created these days are much too complex for human mind to keep track on every single possibility and fault in logic. This is where and why testing is needed (Pan 1991.)

4.2 Levels of tests

Software testing includes different types of testing levels specializing in testing certain scale within the software. These levels are unit testing, integration testing, system testing and finally acceptance testing as seen in figure 5. The levels are ordered

from smallest scale to largest. This document focuses mostly on the lowest, unit testing level, but other levels are briefly described also (Software Testing Fundamentals 2011, Software Testing Levels.)

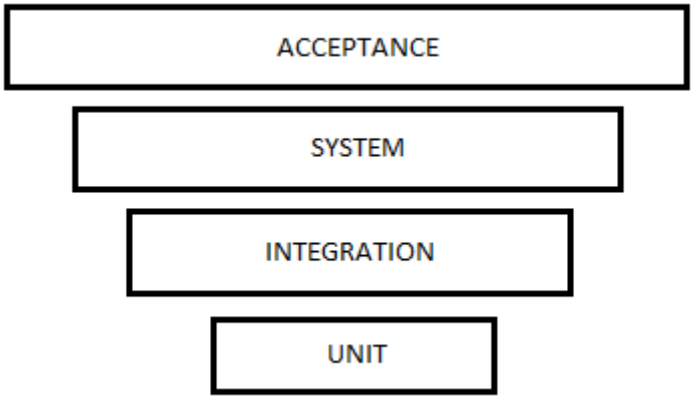


Figure 5: Levels of testing, by scale

4.3 Unit tests

Unit tests are in the lowest section of the testing hierarchy. These tests are aimed to test the smallest testable pieces of code isolated from the rest of the software's logic. Unit tests are created for components that usually have one or more inputs and a single output. Tests can be created either as a white box or black box methods. Unit tests are usually fast to execute, and the most numerous in any software. Unit tests give reliability and validity to the isolated logic behind pieces of software, and serve as a base for other levels of testing.

In standard server side programming languages, such as PHP or C#, unit tests are usually created for each class in the software. Unit tests are done by mocking, stubbing and faking the outside requirements of the component being tested to ignore any possible interaction and failure not related to the current scope of the test. See Figure 6 for example.

On the other hand, some developers prefer to not use the isolation technique of unit testing if not actually necessary, and rather let the outside requirements of the com-

ponent being tested be included in the testing process. This might lead to a test failing because of an outside interference, nevertheless the idea behind this is that the outside component is also tested and should be fixed sooner than later (Fowler 2014.)

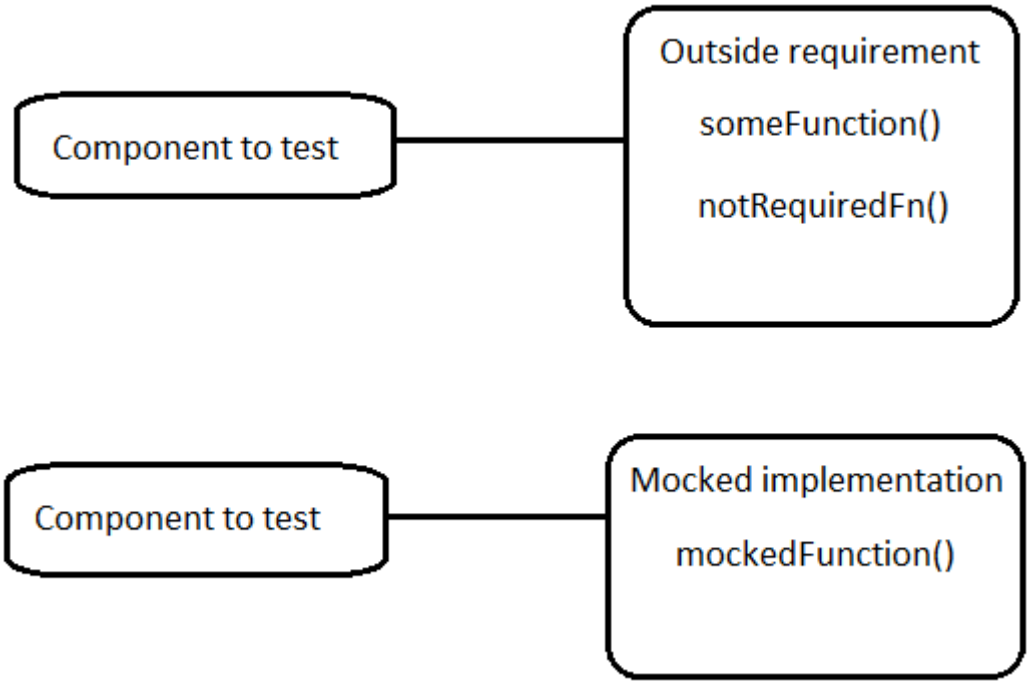


Figure 6: Requirements can be mocked with only required parts implemented

Unit testing is usually (and should be) performed during the development process of the program. Unit tests are created by the developer before, or after the component's creation. Test-first testing development processes, such as Test Driven Development rely on creating the unit (and other levels) tests before the actual component is implemented.

5 Implementation of tests to AngularJS application

5.1 Tools and methods

One of the key advantages in AngularJS when viewed from the unit testing perspective is the dependency injection of the framework. Due to the nature of unit tests and isolation of concepts when designing and implementing unit tests, mocking the dependencies for controllers, services and directives becomes simple with the dependency injection.

AngularJS has a library, *angularMocks*, designed to mock the applications components as its name says. Test runners are used to actually run the tests and generate results. One of the most used test runner for tests created for AngularJS project, as well as other tests written in JavaScript is the Karma (previously called Testacular) test runner library. It is the tool also used in this document.

There are several testing frameworks for AngularJS in which the tests are actually run. Nothing restricts the developer to be retained to a single testing framework; however for the simplicity's sake, one should not use too many different frameworks than necessary. These tests are written with Jasmine framework, which is the one AngularJS team themselves recommend. Other frameworks include, but are not limited to Mocha, Chai, and their extensions, like Chai-As-Promised.

5.1.1 Jasmine

Jasmine framework contains all the necessary components to successfully write tests for a REST client application, such as handling of promises.

Jasmine is described as a "behavior-driven development framework for testing JavaScript code". Jasmine is not restricted to a single JavaScript framework. Jasmine offers a clean and readable way to quickly write unit tests without the developer needing to specify the JavaScript framework used (Jasmine Framework documentation.)

Jasmine tests are ordinary JavaScript functions, which makes them instantly familiar to use for a frontend developer. Tests suites are created by the global function 'describe' and test specs are created by the function 'it'. Describe functions can be nested as needed, so the developer has all the control to the test structure.

Following the unit testing guideline of setup and teardown phases, Jasmine implements them as functions beforeEach / beforeAll and afterEach/afterAll. BeforeEach or beforeAll functions are run as their name says before every test suite, and one should implement the necessary mocking and setuping inside these functions. After the test suite has run, the afterEach or afterAll functions are run, and this is the place for tearing down anything that requires to do so. Using beforeEach and afterEach provides the developer a way to specifically setup/teardown

5.1.2 Spies

While unit testing, isolation is the key and any outside logic not belonging to the currently tested feature should be mocked. Jasmine offers spy functions to mock function's result values and determine if correct functions where called when the tests are executed. In default, spy objects replace the function they are assigned, and as a result, callback functions can be painful to test. Luckily, spies can be delegated to use the actual implementation of the function.

Spies are created as described in Figure 7:

```
// Something that should be mocked
var object = {
    someFn : function() {}
};

// Spy is created to the objects someFn, and a return value is assigned to it
spyOn(object, 'someFn').and.returnValue(1);
```

Figure 7: Creating a jasmine spy

Spies can be assigned to return a certain value as pictured, or they can be assigned to call some other function with the *and.callFake()* function chain. If the mocked function is required to fall back to the actual implementation, the *and.callThrough()* function chain is used.

Evaluating spying results in the end of test is done as seen in Figure 8:

```
// Execute
var result = object.someFn();

// Assert results
expect(result).toEqual(1);

// Expect spy to have been called
expect(object.someFn).toHaveBeenCalled();
```

Figure 8: Evaluating spy after test

The *toHaveBeenCalled()* function has two variations, *toHaveBeenCalledTimes()*, which takes a number as parameter and evaluates how many times the spy has been called, and *toHaveBeenCalledWith()* which evaluates in what parameters the function has been called.

5.1.3 Promises

One of the key parts of a REST-architectural client application written in JavaScript are promises; functions which return values are resolved asynchronously. Promises can be hard to test, as the value they return cannot be mocked as easily as a normal variable. The promises resolve in the real application after a certain timeout after which their values are returned, like in a *$http* call to the backend. In unit tests, no actual calls are wanted to be made to the backend, however, still developers want to test the functions using these promises with similar asynchronous behaviour.

5.2 Implementing Unit Tests

Writing the unit tests proved to be fast and rather simple after the initial setup of the testing framework and runner. As previously noted, testing framework used during this document's process was Jasmine framework, and the test runner used was Karma. The application to which the tests were created has a REST architecture, with a lightweight PhalconPHP PHP-framework serving as a backend, and a MongoDB NoSQL database. The tests written contain only unit tests for the client implemented in AngularJS 1 version and the backend is completely ignored in the test cases.

The tests were written for the controllers interacting with the scope and view, services handling the GET, POST, PUT, DELETE methods to backend, and directives holding a more special functionality. Service testing with Jasmine provided to be rather simple, thanks to the mocked *$httpMock* -service which provided a clean and easy way to mock the return statements of the backend without actually requiring any interaction with backend and the database.

5.2.1 Initial setup

The Jasmine framework and Karma test runner were installed to the application with npm. Installing node packets with it is rather straightforward, requiring few commands to successfully install the packet and its requirements.

Installing Jasmine with command line tool:

npm install –g jasmine

This installs the Jasmine framework globally to the computer and it can be used in the project.

Installing Karma with command line tool:

npm install karma

This installs Karma test runner. Karma development team recommends the Karma test runner to be installed locally to the projects folder. When installed globally, it can sometimes interfere with other installations in the computer, and this was found to be problematic during the creation of tests.

Karma requires a configuration file in order to run. The configuration file contains information such as paths to the applications files and test files, frameworks, browsers in which the tests are run and special Karma pre-processors that might be needed and other runner specific configurations. This configuration file can also be created with tool such as gulp, if the project has it. The configuration file depends on the project's structure, and is rather simple so it is not looked into any deeper in this document.

Any other extensions or frameworks can be installed similarly to the previous examples, using npm, as long as they are contained in the package manager's repository.

Example of the file structure used in the application and testing is seen in Figure 9.

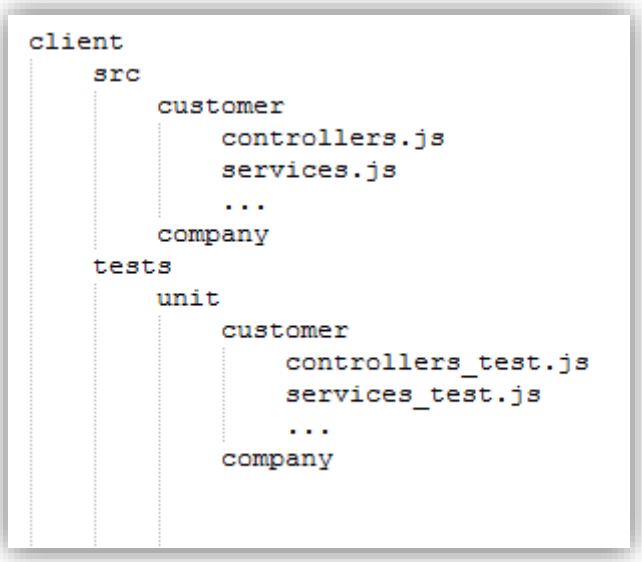


Figure 9: Structure of application and test files

Each component is contained within its own folder, holding possible controller, service, directive files as well as initializing and route configuration specific files. The test files are in a folder separate from the applications source files, however, they are mimicking the file structure of the application, with each component in its own folder and tests split to files containing either controller, service or directive tests.

An argument can be made that the test files should be contained in the same folder as the main application source files. This provides an easier way to share and transfer components not specifically tied to the logic of the application between other applications. This way the tests are shared as well and no additional testing would be

needed. As the code in this application is rather application specific and components with any 'general' usage are rather scarce, tests were placed to a separate folder.

5.2.2 Testing controllers

Because of the structure of the application, most of the controllers hold a similar functionality and logic within them. Most controllers implement a CRUD logic to the entity that they are responsible of, for example creating or editing customer information, listing and deleting customers. Unit testing these kind of features is quite similar. To reduce the need to mock the same request structure used in nearly every controller, it was decided to create a new service which holds the commonly used service methods in controllers (Figure 10). This allows an easier mocking phase for the tests, while allowing test specific parameters and return values to be used. The usage of AngularJS' *$q* service allows to mock the structure of promises, and return a controlled value with the response.

```
angular.module('Nestor.TestHelper')
    .factory('MockResource', [
        '$q',
        function ($q) {
            var mockObject = {
                _id: {
                    $id: "1234"
                }
            };

            var mockResource = function () {};

            mockResource.prototype.$save = function () {
                var deferred = $q.defer();
                deferred.resolve({});

                return deferred.promise;
            };

            mockResource.service = function () {
                return mockObject;
            };

            mockResource.query = function (object) {
                var returnValue = typeof(object) !== "undefined"? object : mockObject;

                var deferred = $q.defer();
                deferred.resolve([returnValue]);

                return {
                    $promise: deferred.promise
                };
            };
```

Figure 10: Mock resource is created to provide reusable methods for mocking.

Testing a controller starts by injecting the required services and other components in the test suites *beforeEach* section (Figure 11). All required mocking for the tests should be also done in this section. Any required *$scope* property should also be mocked. Finally, the new controller is initialized with the required injection parameters. Not every parameter is required to be a mock, however following the idea of ignoring any possible outside interference, it was preferred to use at least empty objects in place of the actual implementation of component if possible.

```
describe('Test ProjectSidebarController', function () {
    var $rootScope, controller, $scope, mockResource, projectMock, customer, company, project;

    beforeEach(inject(function ($controller, _$rootScope_, _MockResource_, _Project_) {
        $rootScope = _$rootScope_;

        // Init new scope
        $scope = $rootScope.$new();

        mockResource = _MockResource_;
        projectMock = _Project_;

        company = {
            name: "TestCompany"
        };

        customer = {
            name: "TestCustomer"
        };

        project = {
            name: "TestProject"
        };

        spyOn(projectMock, 'getCustomers').and.callFake(function () {
            return mockResource.query(customer);
        });

        spyOn(projectMock, 'getCompanies').and.callFake(function () {
            return mockResource.query(company);
        });

        // Get controller
        controller = $controller('ProjectSidebarController',
            {
                $scope: $scope,
                $rootScope: _$rootScope_,
                Project: projectMock,
                _project: project
            });
    }));
```

Figure 11: The *beforeEach* section of the controller test

In the figure above, the benefit of the *mockResource* service can be seen, where there is no need to write separate mocked query functions for the project services *getCustomers* and *getCompanies* method's, however rather pass the value that is

wanted to return to the mockResources query function. The usage of spies allows to verify that these service calls were indeed made during the execution of the tests.

In the last part of the image, the controller is initialized with several mocked parameters passed to the constructor function, overriding the actual implementation of the components.

The actual tests are implemented in 'it' functions, provided by Jasmine. Because of the structure of AngularJS controller and it's relation to *$scope,* to achieve the data-binding features manual use of the $digest function of the $scope is needed to fire the watchers in $scope in the tests (Figure 12).

```
it('should be defined', function() {
    expect(controller).toBeDefined();
});

it("should format scope variables correctly", function() {
    expect($scope.user).toEqual(user);
    expect($scope.company).toEqual(company);
    expect(authMock.getUser).toHaveBeenCalled();
});

it("should call Company.getCustomers on getCustomersList function", function() {
    $scope.getCustomersList();
    $scope.$digest();

    expect($scope.customers).toEqual([customer]);
    expect(companyMock.getCustomers).toHaveBeenCalledWith({companyId: "1234"});
});
```

Figure 12: Implementing tests to controller

5.2.3 Testing services

Writing unit tests for services mainly used for communicating with the backend PHP server is simple by mocking the backend with a *$httpBackend* service contained in *ngMock* AngularJS module. This mock allows to prevent the actual requests to the backend, which are not required in the unit testing context, and to use predefined responses to routes as a placeholder. Service testing with REST applications does not require to actually be interested in the response, however, rather to what path the request is sent, and if it is using the correct HTTP method with the correct data.

The $*httpBackend* contains several functions used to make expectations to different routes and can mock the returned value from that route. Expect functions will fail the test if they are not called within the execution of the test. Similar to expect functions, $*httpBackend* contains a **when** function that can be used the same way to mock responses to requests in certain routes. When functions do not fail the tests when no calls are made towards them, and they can be used in looser unit testing like black box testing as well as integration testing.

Mocking the backend with $*httpBackend* requires it to be injected to the test suite and defining the desired route, possible data that is sent in the request, and giving a response. After the mock and expect function is defined, method of the service which is the target of the testing that is called with necessary data. After the method execution, *$httpBackend.flush()* method is called to mock the asynchronous nature of the http calls and give the mocked response (Figure 13).

Finally, *$httpBackend* has two functions to verify that all requests defined were made and every expectation was fulfilled. These functions, *verifyNoOutstandingExpectation()* and *veryfyNoOutStandingRequest()*, should be done in the *afterEach* section of Jasmine test suite, eliminating the need to repeat the code after each 'it' block (Figure 14). Any request that has not been fulfilled, or some request that was not expected, give errors when the test is run (Figure 15).

```
it('should perform POST on authenticate', function () {
    // Second parameter is the POST data
    $httpBackend.expectPOST(expectedUrl, "userCredentials").respond(true);

    // Perform request
    Service.authenticate("userCredentials");

    // flush requests
    $httpBackend.flush();
});
```

Figure 13: Testing a service method that uses HTTP POST

```
// make sure no expectations were missed in your tests.
// (e.g. expectGET or expectPOST)
afterEach(function() {
    $httpBackend.verifyNoOutstandingExpectation();
    $httpBackend.verifyNoOutstandingRequest();
});
```

Figure 14: Verify that no request remain unfulfilled after each test

Figure 15: Test failed due to wrong expectation

5.2.4 Testing directives

Testing directives consists of testing both the controller that the directive uses, as well as the directive's compiling itself. Testing the directives controller is equal to standard controller testing. One should also test any service related to the directive's functionality that is used in controller.

The testing of directive's HTML and how it appears on the page's DOM can be seen as conflicting with integration and the end 2 end tests done in the browser; however, the same testing of HTML can also be done via unit testing if necessary.

The initialization and *beforeEach* section of directive tests are similar to a controllers test (Figure 16). It requires mocking any outside parameter that is injected to the directives controller during the execution of the test. In addition, directive testing requires the use of AngularJS' *$compile* service to actually generate the html code that the directive displays in DOM. The manual *$digest* call is also recommended to be done in this section for AngularJS to create the html.

```
beforeEach(inject(function(_$rootScope_, _$compile_, _Comment_, _CommentFeed_, _MockResource_) {
    $scope = _$rootScope_;
    $compile = _$compile_;

    commentMock = _Comment_;
    mockResource = _MockResource_;

    comment = {
        belongsToFeature: "articles",
        objectData: {
            title: "Test article",
        },
        createdBy: {
            firstName: "Jack",
            lastName: "Jackson"
        }
    };

    spyOn(commentMock, "feed").and.callFake(function() {
        return mockResource.query(comment);
    });

    element = $compile(
        '<div data-comment-feed-component></div data-comment-feed-component>')($scope);

    // $digest updates the html and the actual directive is created
    $scope.$digest();
}));
```

Figure 16: The *beforeEach* section of directive test

The element variable contains the html code of the directive, and testing should be done against it. The testing of element shares characteristics with a standard acceptance of other tests either done in browser by user or automated. The element can be searched for html components via selectors. In the figure below, a div with class .comment-feed-action is searched, and the test verifies that it was indeed created during the compilation of the directive. The contents of the div are also evaluated to validate the correct functionality of the directive (Figure 17).

```
it('Test comment title and creator is displayed on div', function () {
    var div = element.find(".comment-feed-action");

    expect(div).toBeDefined();
    expect(div.text().trim()).toEqual('Jack Jackson commented on article Test article');

});
```

Figure 17: Executing a directive test

6 Conclusions

The path to understand and learn AngularJS can be a difficult one for a new developer, especially with a limited experience in front end JavaScript coding. The AngularJS framework differs greatly by its usage and functionality from such well-established JavaScript libraries like JQuery. Mastering the AngularJS is a long process, however, in my opinion worth it. The clean and documented structure of each components brings much needed clarity to writing JavaScript, and the business-logic is welcome on the client side. Pairing AngularJS with lightweight backend and focusing both the view and business logic on the client provides a simple structure for a today's single page application.

Testing of AngularJS application proved to be a very similar experience as learning the AngularJS itself. After the basics were mastered and the efficient structure of tests was found, writing unit tests to components was quite fast. Of course the similar CRUD operations contained within each controller or service made the writing of tests also fast. The one feature that made AngularJS tests easy to implement was the strong dependency injection service, which helped mocking and stubbing the features required in controllers and services. Rather than fully implementing mocks to every single dependency, one could mock only the needed parts of the features. The spying features in Jasmine were also helpful in order to verify the requests made during the execution of the tests.

As downside the way of doing tests after the application is released and in maintenance cannot be recommended. The testing of certain components, mostly overly bloated controllers with thousand(s) of lines can be nearly impossible to unit test without massive refactoring. This experiment proved that unit – and why not other stages of – testing should be done during the development process rather than afterwards. Why this was not done with this application remains a mystery. Perhaps it was the unfamiliarity with client side testing in browsers because of past experiences or the rarity of JavaScript testing, I do not know. But after studying the unit testing in AngularJS, I cannot do anything but recommend it.

During the writing of tests to application there were some design flaws found in larger components that could have been reworked during the development. This is

one example of the benefits of test-first processes and can save developers from big refactoring processes afterwards. More lines of code in controller also means more human errors and thus more bugs that can be difficult to find or fix.

This document focuses only on the unit testing part of AngularJS. The unit tests written for this document follow the string unit testing philosophy that excludes all outside interaction of the tests scope. The next step could be writing larger integration tests that interact between the other AngularJS components forming a broader perspective of the application's quality and performance. Automated tests using real browser environments and mimic the user interactions within the page using the protractor selenium web driver could be the next step in testing this application.

Perhaps the most valuable result of this document was to provide information about unit testing of web based client applications to the employer, and to bring some kind of proof why unit testing should be done during the development process, rather than afterwards. The knowledge gained during the process of creating this document can be shared between co-workers, and perhaps one day every developer can do every phase of testing, whether it's done in client or backend, using the most up-to-date tools and practises, and to make software that does not break under stress. The last part is probably not realistic in any way in the world we currently live in.

References

AngularJS Developer Guide. Accessed on 12 April 2016. Retrieved from http://www.angularjs.org.

AngularJS Git hub wiki. Accessed on 16 April 2016. Retrieved from https://github.com/angular/angular.js/wiki/.

Cooper, J, 2013. AngularJS Step-by-Step: Services. Accessed on 16 April 2016. Retrieved from https://www.pluralsight.com/blog/tutorials/angularjs-step-by-step-services.

Fowler, M, 2014. Unit Test. Accessed on 12 May 2016. Retrieved from http://martinfowler.com/bliki/UnitTest.html.

Freeman, A, 2014. Pro AngularJS.

Jasmine Framework documentation. Accessed on 8 May 2016 2016. Retrieved from http://jasmine.github.io

Graetz, B. AngularJS vs EmberJs. Accessed on 8 May 2016. Retrieved from http://angularjs-emberjs-compare.bguiz.com/index.html.

Karma Test Runner documentation. Accessed on 8 May 2016. Retrieved from http://karma-runner.github.io/0.13/index.html.

Kasurinen, J.P, 2013. Ohjelmistotestauksen käsikirja.

Pan, J, 1999. Software Testing. Accessed on 12 May 2016. Retrieved from https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

Software Testing Fundamentals. Accessed on 12 May 2016. Retrieved from http://softwaretestingfundamentals.com.