Dinh Thien Phuc Tran

Design and Implement Scalable Robust Modern Web Application

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

01 May 2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Dinh Thien Phuc Tran Design and Implement Scalable Robust Modern Web Application 53 pages 1st May 2016
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	Software Development
Instructor(s)	Harri Airaksinen, Principal Lecture

Modern web application has grown in popularity because of its ease of access on multiple platforms and devices. In regards to this growth, the expectations for modern web application have arisen over the years. As a result, modern web application nowadays can have large scale and high complexity just as any native desktop or mobile application.

To build a modern web application, there are differences in choice of tools, frameworks and theoretical approaches. Each of the choices has its own advantages and disadvantages, thus, in order to create a robust and scalable modern web application, developers should be able to grasp the technologies provided by the choice, and the relation between the project requirements and the features of which technologies have to offer.

Recognizing the vital aspects of using tools and frameworks for modern web application development, this study aims to conclude and provide an approach to utilize a certain tool and framework, which are AngularJS and TypeScript in particular, in order to produce a highly scalable and maintainable development approach. In order to achieve this, the study analyses the current states of modern web, assesses the tools and frameworks in question, and through a case project, demonstrates their features with a detailed architectural implementation.

Throughout the study, AngularJS has proved its expressiveness, reusability and testability in its components. TypeScript has also proved its essential role in the development process of the large web application for its type system. The combination induced a high level of abstraction to the development process and slowed the pace at first, but in the long run, from the perspective of scaling and maintaining, the combination itself has made the development process much more effortless and at the same time, more reliable.

Keywords	AngularJS, TypeScript, SPA, modern web application

Contents

List of abbreviations, acronyms and terms

AngularJS	Refers to the 1st version of AngularJS or “Angular 1”
Angular	Refers to the 2nd version of AngularJS or just “Angular”
API	Application Programming Interface
SPA	Single Page Application
ECMAScript	A trademarked scripting-language specification standardized by ECMA International in ECMA-262 and ISO/IEC 16262
DOM	Document Object Model
XHR	XMLHttpRequest
MVC	Model-View-Controller architectural pattern
MVVM	Model-View-Viewmodel architectural pattern
MVW	Model-View-Whatever architectural pattern
ASP.NET	An open-source server-side web application framework designed for web development to produce dynamic web pages
Ruby on Rails	A web application framework written in Ruby
Laravel	A free, open-source PHP web framework following the MVC architectural pattern
UML	Unified Modelling Language
UI	User interface
ES6	ECMA-262 6th edition
ES5	ECMA-262 5th edition
OOP	Object-oriented programming
IoC	Inversion of control design principle
RESTful	The characteristic of having REST (Representational State Transfer) software architectural style.

1 Introduction

The thesis focuses on finding the method and best practices to design and implement scalable and robust modern web application with AngularJS and Typescript. At the moment, the majority of AngularJS - also known as Angular 1 - applications are developed with JavaScript. JavaScript provides many advantages with prototypal inheritance, however, classical inheritance is widely required across the industry because of the nature of the enterprise application industries, as well as the high standards classical inheritance approach can provide.

In enterprise software market, customers as well as developers, are expecting more and more complex features from a web application, as if they were desktop software. Therefore, improving the implementation processes and methods is a key step towards better and more robust web applications.

The case company of this study - for non-disclosure purpose, from now known as company X - is a multinational exchange of trading electrical energy. It offers trading, clearing, settlements and associated services in both day-ahead and intraday markets across nine European countries. As the largest power exchange in Europe, X provides wide range of services for power market development. X's IT department is located in Helsinki, Finland. It is where most of the power market systems and trading portals are developed and maintained. With regard to the topic of this study, Company X faces a challenge of how to expand its web trading platform to support new kind of goods and new methods of trades, without breaking the old system or reworking the old application. In other words, the platform has a scalability problem.

The problem of this study related to how the application was structured, in which the legacy structure evolves around a heavy codebase, where components are tightly coupled and maintenance and expansion became unnecessary challenges over the years. Thus the objective of this study is to develop an architecture approach for the application that better supports future development of the application. To develop a new approach to the architecture of the application, the study will firstly consider the current state of modern web frameworks to create a strong foundation for AngularJS advocacy. Under those circumstances, since AngularJS is a relatively large framework and TypeScript itself is a new language. A major part of the study including sections 2.3 and 3 will discuss

the theoretical frameworks of these technologies along with usage of TypeScript inside AngularJS. The rest of the study will then demonstrate the actual implementation, deriving from the theories, with the help of a mock case project, which is the aforementioned trading platform web application from company X.

2 Modern web application and development tools

2.1 Industry demand and the current state

2.1.1 From desktop and mobile to the web

The World Wide Web (from now referred as the web) has emerged as an essential part of everyday life. For one thing, the web enables us to complete tasks and acquires information easily with a few clicks. As technology advances, almost everything is now available on the web as a service. It can be shopping for groceries, remote learning or even medical consulting. It has also come to the point that when a person wants to look for a service or a solution, he or she assumes options, including the best ones can easily be found on the internet, not as a desktop application installation, nor link to download mobile application, but a web application that can be used directly on the browser.

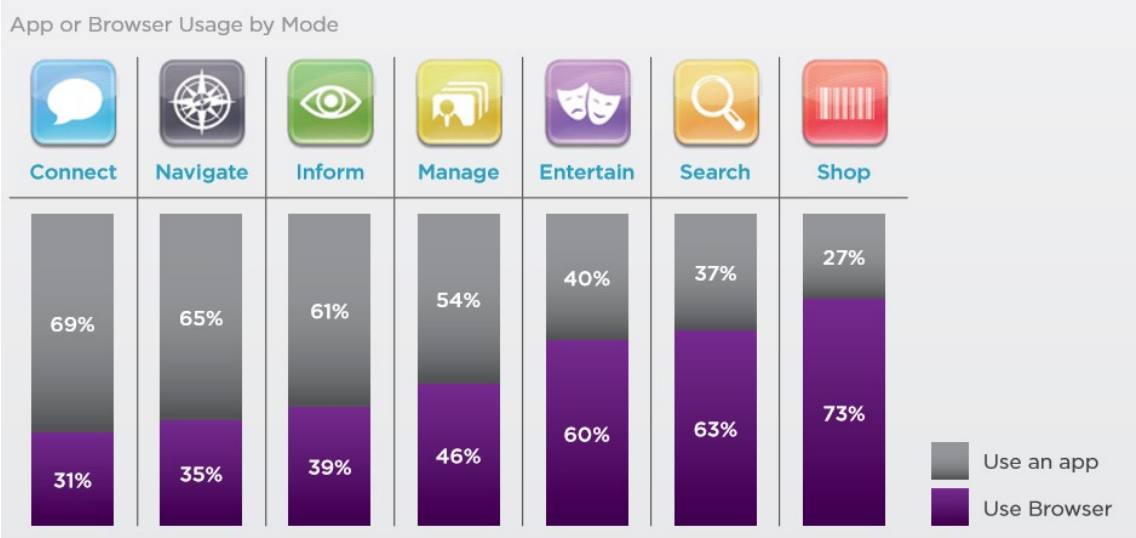

Figure 1. App or browser usage by mode. Reprinted from Yahoo! Inc. (2011) [1].

Figure 1 shows that beside the device-dependant usage like connecting, navigating, informing, browser is the more popular option for shopping, searching and entertainment. This is evidence to the basis for a rule of thumb - an application that can be made as a web application in the browser should be made as a web application. From this information, a question arose which is: can the web platform - the browsers - enable all the features needed by modern web application? The features in questions are the common features, those can be found in native desktop application and native mobile application such as:

- Local data storage and management.
- Hardware related feature e.g. geolocation, proximity, device orientation.
- Low-level TCP API for protocol on top of it e.g. IMAP, IRC, POP.
- Graphical drawing.

The short answer to the question is: it can. The again, the longer answer that might disappoint us: it was not originally designed to do so. The next section will discuss and evaluate the limitation of the browsers to support common features in modern web application.

2.1.2 Premature support and browsers inconsistency

Based on technical principles, all of the features mentioned earlier in section 2.1.2 can be done in the JavaScript implementation provided by the browser implementation as a medium to the host of the browser, which can be a mobile device or a personal computer. Nevertheless, those types of implementation involve very low-level process without any supports from APIs or any level of abstraction. The biggest blocker is that there are not enough, or not yet standardized implementations of web APIs, which are essential to modern web application.

API names	Short description	Status
Network Information API	Provides basic information about the current network connection, such as connection speed.	STANDARD
Bluetooth	Provides low-level access to the device's Bluetooth hardware.	NON-STANDARD
Network Stats API	Monitors data usage and exposes this data to privileged applications.	NON-STANDARD
TCP Socket API	Provides low-level access to the device's Bluetooth hardware.	NON-STANDARD

Table 1. Current status of Communication APIs. Modified from Mozilla Developer Network (2015) [2].

Table 1 describes the current state of some Communication APIs' implementations. Most of these APIs are experimental technologies and not yet stabilized. These APIs do not have long term supports and they are subjected to rapid change in the future depending

on the browser changes as well as the changes in ECMAScript language specification. The APIs in the table are implemented and documented by Mozilla, one of the pioneers in modern and open source web. Consequently, most of these APIs are compatible only with Mozilla made browser with Gecko engine e.g. Firefox, which leads us to the second biggest blocker: inconsistency features support between browsers and platforms. Figure 2 points out the differences in regards to features support, available platforms, implementation of engines, and view area across browsers.

	Google Chrome	Mozilla Firefox	Internet Explorer	Microsoft Edge	Safari
Market share (2015/08)					4.71%
Market share (2015/02)	48 %	13 %	12 %		16 %
Features					
Cloud bookmark sync.	✓ with a Google account	✓ via plugin	✗		✓ with a MobileMe account
Session management	✓	✓	✓ via plugin		✓ through history menu
Private browsing	✓	✓	✓		✓
Download manager	✓	✓	✓		✓
Full screen mode	✓	✓	✓		✗
Tabs on the side	✓	✓ Via Plugins	✗		✗
Custom extensions	✓	✓			✗
Platforms					
Windows	✓	✓	✓	✓	✓
Mac OS	✓	✓	✗	✗	✓
Linux	✓	✓	✗	✗	✗
Android	✓	✓	✗	✗	✗
iOS	✓		✗	✗	✓
Windows phone	✗	✗	✓		✗
Technical details					
Layout engine	WebKit	Gecko	Trident	EdgeHTML	WebKit
Javascript engine	V8	TraceMonkey	Chakra	Chakra	Nitro
Open source	✓	✓	✗	✗	✗
Viewing area					
Default UI height	89 px	113 px	132 px		96 px
Minimal UI height	59 px	102 px	114 px		80 px

Figure 2. Features support across browser. Modified from Social Compare (2015) [3].

One of the purposes of having a universal web application instead of multiple device-dependant application is the consistency for users from different devices and platforms. However that purpose is strongly denied by the inconsistency of features support between browsers as indicated in the figure above. Chrome and Firefox are known as front runners for modern browser support and Microsoft Edge (and Internet Explorer) and Safari are known as playing by the standards browsers. Although this issue is not in the scope of this study, it is necessary to understand that this inconsistency between browsers are made of different causes, such as:

- Different groups of browser user lead to different demands in features

- Backward compatibility to e.g. legacy corporate intranet
- Differences in browser engines because of the nature of the devices and platform

By understanding the causes of this issue, one can acknowledge the fact that the mentioned inconsistency will persist in the development of web browsers, simply because it is also an important incentive for further feature discovery and development, as well as competition in the world of browsers.

2.2 History of frameworks and benchmarking

What has the web community been doing in order to deal with the current state mentioned in the previous chapter 2.1? They have been producing a lot of libraries and frameworks. Table 2 lists 10 of the most popular JavaScript frameworks and libraries [4].

Name	**Short description**
AngularJS	AngularJS lets you use HTML and extend HTML syntax to express your application components clearly and succinctly.
React	React is a JavaScript library for building user interfaces.
Meteor	Meteor is an ultra-simple, database-everywhere, data-on-the-wire, pure-JavaScript web framework.
Redux	Predictable state container for JavaScript apps.
Ember.js	Ember.js is a JavaScript framework that does all of the tasks that are common to every web app, so you can focus on building killer features and UI.
Moment	Parse, validate, manipulate, and display dates in JavaScript.
Three.js	The library provides <canvas>, <svg>, CSS3D and WebGL renderers.
Ionic	Advanced HTML5 Mobile App Framework. A beautiful front-end framework for developing hybrid mobile apps in HTML5.
jQuery	jQuery is a light weight JavaScript library which provides fast and easy way of HTML DOM traversing and manipulation, event handling and client side animations.
Backbone	Give your JS App some Backbone with Models, Views, Collections, and Events.

Table 2. Popular JavaScript libraries, frameworks, and plugins. Modified from JavaScripting (2014) [4].

jQuery is one of the first libraries in the history of web technologies that not only deals with the browser compatibility issues, but also introduces a modern approach to web application development with leaner DOM manipulation, event handling, XHR implementation with Ajax, etc. The library also enables developers to create abstractions for low-level UI behaviours, as well as underlying functionalities of the browser engine. As a matter of fact, JQuery library has a considerable influence on how modular the frameworks are nowadays.

jQuery is also used as a base library to develop a framework for web application such as SproutCore. Nonetheless, it is straightforward to perceive that putting a layer abstraction on top of native JavaScript, with the sole purpose of creating a framework, which will result in performance down weight in the future. The need for frameworks written in vanilla JavaScript – plain JavaScript without additional libraries - has become apparent. For the architectural scope of this study, I am going to introduce and compare shortly three prominent frameworks in the past, today, and in the near future: Backbone.js, AngularJS, and Ember.js.

Backbone.js may not be the first MVC framework, but it is the first modern one. However, it only partly follows the MVC pattern as the framework provides key-value binding as models and collections of models. View and controller, to some extent, overlap in functionalities, which violates the MVC pattern slightly. The heart of Backbone, which makes it stand out from others, is its Events model. Regarding the support for single page web applications, the well-defined Events model in Backbone.js can help developers manage, dispatch and listen to popular browser events, as well as custom events throughout the application life cycle.

The next candidate is AngularJS framework, also known as Angular 1. AngularJS also embraces the model and the view in the MVC design pattern, however, in AngularJS, models are not just only models, as they also can contain business logic in the domain data. The strongest features of AngularJS are data binding, HTML syntax extension as directives, and its dependency injection approach. Data binding eliminates the coupled process of view manipulation through traditional DOM manipulation and HTML directive makes the view part of the application highly expressive and declarative, which helps a lot with the modularity of the application and also the debugging process. Among with

those features, dependency injection helps removing a lot of the redundant code, embracing reusability and more importantly, improving performance with for instance, the singleton approach provided by its dependency injection approach.

That leaves us Ember.js, the oldest framework of all three, which originates from the JQuery based framework Sproutcore. Being also an MVC framework, what really distinguishes Ember.js from the other two is its focus on data model performance and its favour of convention over configuration. To begin with, Ember.js would automatically determine the name of the route and the according controller when a router resource is created, even if you have not defined one. This integrates pleasantly with any other RESTful JSON APIs those have similar convention such as ASP.NET Web API, Ruby on Rails, and Laravel. Ember.js also includes a built-in adapter for setting up fixtures for developing against mock API and testing, which can considerably speed up parallel development in a web application development process.

The following section will more closely compare and examine some factors that affect the decision of developers when choosing a framework for the next application. The first factor is about the community since all the frameworks are open-source. Table 3 indicates the current state of the community of each framework. AngularJS, backed by Google, is no doubt the winner with more attention than both Ember.js and Backbone.js combined, in terms of stars on GitHub, third-party modules, Stackoverflow questions, YouTube search results and so on.

Metric	AngularJS	Backbone.js	Ember.js
Starts on Github	40.2k	18.8k	14.1k
Third-Party Modules	1488	256	1155
StackOverflow Questions	104k	18.2k	15.7k
YouTube Results	~93k	~10.6k	~9.1k
GitHub Contributors	96	265	501
Chrome Extension Users	275k	15.6k	66k
Open Issues	922	13	413
Closed Issues	5,520	2,062	3,350

Table 3. Community status of web frameworks. Modified from Airpair (2015) [5].

Closely related to the current state of the community, the next factor reflects how fast the popularity is growing for each framework, which is expressed in figure 3. As the first modern framework, Backbone.js was no doubt the preferred version from the beginning, however since 2013, a steep decrease can be seen in Backbone.js's popularity. It is also the same period when AngularJS 1.3.x and 1.4.x, the most popular version in Angular application nowadays, was released.

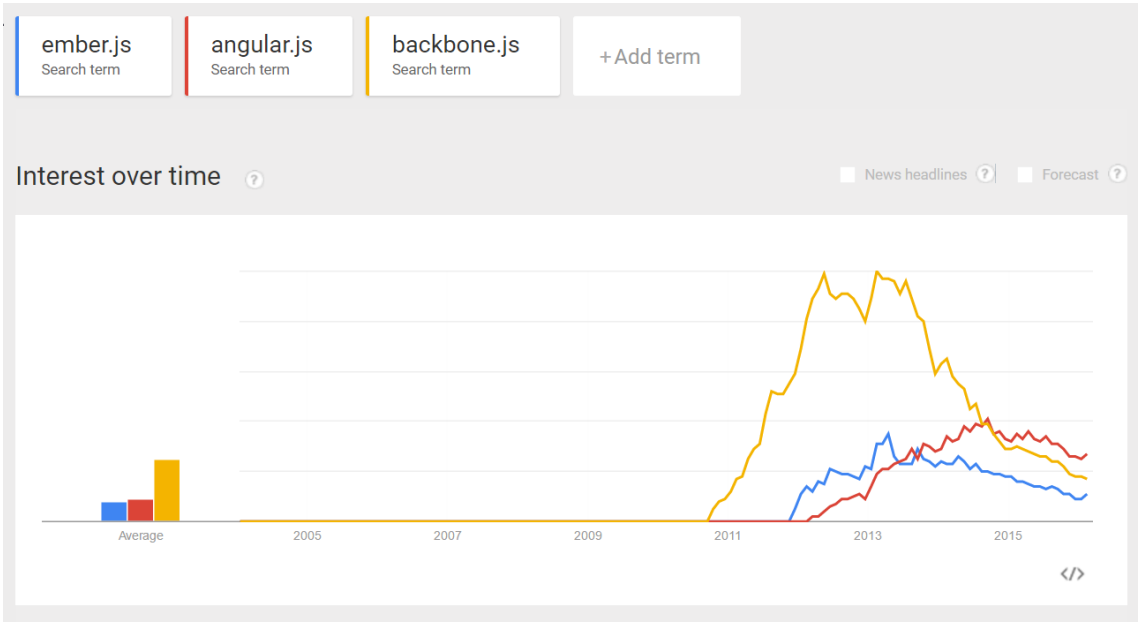

Figure 3. Interest over time of frameworks. Reprinted from Google Trends (2015) [6].

The last factor to be considered in the comparison is size. The decision of choosing a framework can be also considerably affected by the size factor, as load time is crucial to the overall performance of any websites. Table 4 compares the sizes of the frameworks with and without required dependencies. Backbone.js wins in the net size, since the framework is just a skeleton, however in most cases a project requires various core third party modules as dependencies. AngularJS wins overall with the smallest size with required dependencies, with its own built in implementation of jQlite and many AngularJS static methods that can mimics the features provided by Underscore library.

Framework	Net Size	Size with required dependencies
AngularJS 1.2.22	39.5kb	39.5kb
Backbone.js 1.1.2	6.5kb	43.5kb jQuery + Underscore, 20.6kb Zepto + Underscore
Ember.js 1.6.1	90kb	136.2kb jQuery + Handlebars

Table 4. Size for web framework. Reprinted from Airpair (2015) [5].

Based on the factors mentioned above and own experience, AngularJS no doubt stands out of all the web frameworks in the past and in the present, and it also is the choice that I have made for my most recent projects, as well as one of the main topics for this study. The detailed features and advantages of AngularJS, as well as drawbacks, will be discovered later mainly in part 3 and throughout the rest of the study through implementation examples.

2.3 TypeScript in the picture

TypeScript is a typed superset of ECMAScript 2015 6th edition syntax that compiles to plain JavaScript, the version of which by default is based on ECMAScript 2009 5th edition specification. In this way, every valid JavaScript program is also a valid TypeScript program, and the default output by TypeScript compiler should work in all modern browsers just like the current version of JavaScript. In this section, I will present the features of TypeScript type system, the enhancement on development process provided by TypeScript, as well as the limitation of the this programming language.

2.3.1 TypeScript Type System

In term of type system, a static typed language such as Java or C# requires you to specify the type in a variable declaration, which later acts as a basis for type constraint in order to enhance code quality, code stability and code comprehensibility. Meanwhile, in a dynamic typed language like JavaScript, there is much more flexibility since there is no type constraint. Also there is a space for inconsistency and errors caused by type-related issues.

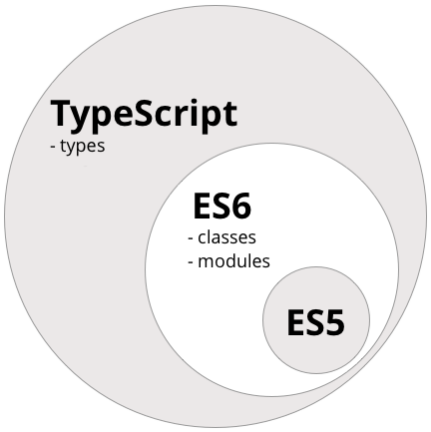

Figure 4. The scope of TypeScript compared to ES6 and ES5. Modified from Lerner (2016) [7].

Figure 4 reveals the scope of the TypeScript type system. What can been seen from figure 4 is that TypeScript does not make the JavaScript implementation strictly type, because it is a super set of ES6, which is also a superset of ES5. In other words, in TypeScript you can have statements of both strictly typed and dynamic typed. Like primitive types in classical OOPs languages, TypeScript provides basic types such as numbers, strings, structures, booleans, etc. With regard to reference types, TypeScript brings the concept of Classes, Interfaces and Modules to JavaScript. However, it is necessary to remark that TypeScript itself does not have an engine implementation like JavaScript, as it will be compiled into JavaScript. To put it differently, TypeScript will not bring any new type system to the JavaScript implementation. Figure 5 demonstrates how TypeScript type system is compiled into the valid JavaScript.

Example.ts

```typescript
interface IGreet {
    greet(): string;
}

class Greeter implements IGreet {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

var stringToGreet: string = "world";
var greeter: Greeter = new Greeter("world");
```

Example.js C:\Users\dinht\Desktop

```javascript
var Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());

var stringToGreet = "world";
var greeter = new Greeter("world");
```

Figure 15. How TypeScript types are compiled into JavaScript. Screenshot.

Reference type is the main factor that embraces TypeScript's core principle of type-checking. For instance, interfaces help defining contracts within your code, in order to ensure consistency between components in your code and also for working with external code outside of the current codebase. Class system helps building reusable components and is essential for object-oriented class-based approach for client web application. The last but not least most important concept is modules. Module in TypeScript is similar to the concept of namespace in languages such as C# and Java. The Module provides an organization schema which helps keeping track of types and preventing name collisions and naming pollution in the global namespace.

2.3.2 Context-aware supports for TypeScript

The term context-aware support in coding refers to the features of listing members and parameter info, autocompleting word, and listing quick information. Since TypeScript is made by Microsoft, Intellisence is no doubt the de facto solution for TypeScript context-aware support. Intellisence for TypeScript feature can be found in the recent version of Visual Studio such as variant of Visual Studio 2013 and Visual Studio 2015. For a similar implementation with a lightweight editor, Intellisence for TypeScript is also included in each version of Visual Studio Code. Apart from Intellisense, the other option is plugin or combination of plugins. For heavyweight IDE, WebStorm provides relatively high quality support for context-awareness of TypeScript. For simpler development environment with only an editor, all popular code editors such as Sublime Text and Atom, provide packages and plugins for basic context-aware support of TypeScript. Figure 6 demonstrates the feature of auto-completion and syntax suggestion in Visual Studio when working with TypeScript. What you can also see from Figure 7, is Sublime's ability to suggest function signature for TypeScript code.

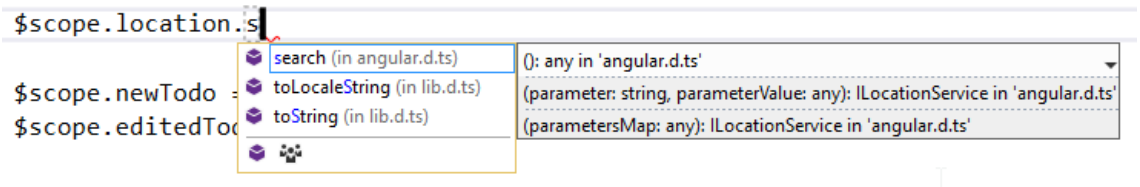

Figure 6. Context-aware for TypeScript with Intellisense in Visual Studio. Screenshot

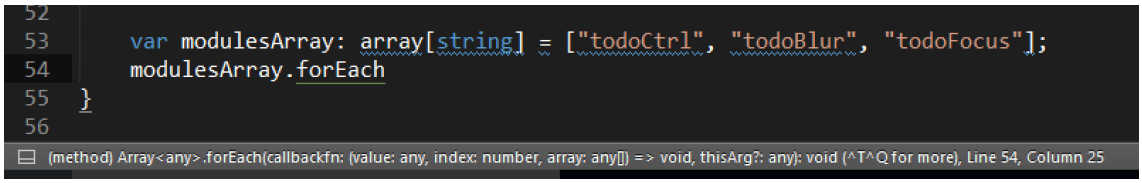

Figure 7. Function signature suggestion for TypeScript in Sublime Text. Screenshot.

2.3.3 Type Definition

Very often a client side web application utilizes multiple third party modules from libraries and frameworks. These modules are originally written in JavaScript. Bringing JavaScript to the current TypeScript application, while they can still interoperate, would break the purpose of a type safe ecosystem. Hence, type definitions are needed to ensure the type-safe consistency across the web application. From a C# or Java background, these codes would be the class and interface definitions you can see from third party components, with the implementation excluded. The code in figure 8 is an excerpt from AngularJS type definition. What you can see in the excerpt is not the implementation of AngularJS functions, but only the function signature and the parameter information.

```
getJSON(url: string, success?: (data: any, textStatus: string, jqXHR: JQueryXHR) => any): JQueryXHR;
/**
 * Load JSON-encoded data from the server using a GET HTTP request.
 *
 * @param url A string containing the URL to which the request is sent.
 * @param data A plain object or string that is sent to the server with the request.
 * @param success A callback function that is executed if the request succeeds.
 */
getJSON(url: string, data?: Object, success?: (data: any, textStatus: string, jqXHR: JQueryXHR) => any): JQueryXHR;
/**
 * Load JSON-encoded data from the server using a GET HTTP request.
 *
 * @param url A string containing the URL to which the request is sent.
 * @param data A plain object or string that is sent to the server with the request.
 * @param success A callback function that is executed if the request succeeds.
 */
```

Figure 8. Excerpt of AngularJS type definition. Screenshot from Github (2016) [8].

TypeScript type definition file is a valid TypeScript file with “.d.ts” extension, of which “d” is short for definition. These files have one single purpose, to support context-awareness, when actual implementation of these external modules are used, such as demonstrated in figure 9.

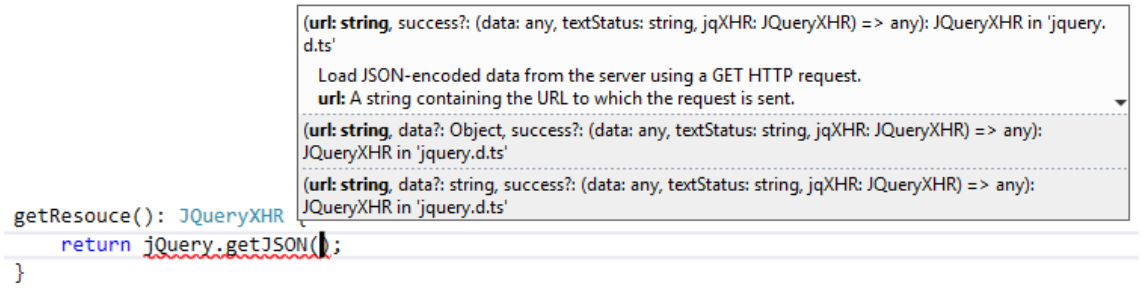

Figure 9. Example use of TypeScript type definition. Screenshot.

2.3.4 Limitation of TypeScript

The concept of external type definition brings up an issue: What if a desired library does not have a type definition? Most of the type definitions are available at DefinitelyTyped, an open source project which is a repository for high quality TypeScript definition. However, number of libraries and new versions grow just as fast as or even faster than the production speed of the community behind DefinitelyTyped. At a result, in rare cases when a type definition cannot be found, plain dynamic typed codes are inserted between static typed codes, which is considered a bad practice in web application written in TypeScript as mentioned in section 2.3.3. As demonstrated in figure 10, the purpose of the mapping function is to create a new instance of the derived type *ExpandedStrictType*, However, by using plain JavaScript, the environment does not have enough type information to infer type and warn the user about the type of property "*sise*", which should have been "*size*". Correspondingly, the result instance is inferred as a general Object type, which is clearly not the original intention of the author.

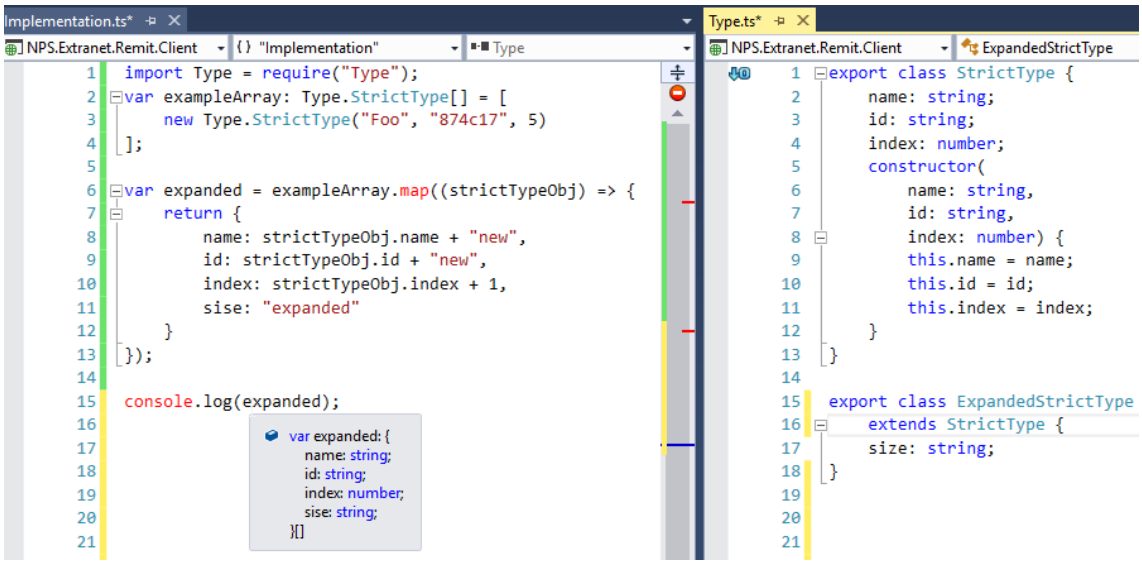

Figure 10. Effect of inducing JavaScript into TypeScript. Screenshot.

The other limitation of TypeScript is the nature of a language's superset: it does not eliminate any of the base languages. Any language has strengths and weaknesses, as is JavaScript, and TypeScript did not try to eliminate the weaknesses since all legal JavaScript programs are also legal TypeScript programs.

Having discussed the characteristics of TypeScript language in closely related to the JavaScript development environment, in the next chapter, I will firstly examine the AngularJS as a framework. Based on that foundation, I will then demonstrate the usage of TypeScript in modern web application context, created by AngularJS in particular.

3 AngularJS and TypeScript as a framework

First of all, it is important to distinguish the difference between a framework and a library in web development. A library provides a set of objects and functions to solve problems in a particular area of development, for example the jQuery library exposes the jQuery object with multiple functions that can simplify DOM manipulation or encapsulate XHR-related processes. Underscore library through the Underscore object exposes useful functional programming helpers without extending the built-in objects such as native Array Object in JavaScript. A framework, on the other hand is a collection of libraries that centres on a particular development approach and covers all areas of application development related to that approach. For example the Meteor framework aims at a total coverage of any web application ecosystem, by providing a platform to build web application entirely in JavaScript. To cover all area of web application development, Meteor provides all kind of libraries, from libraries for creating views, managing data collections, managing dependencies, to libraries for managing user session and for controlling the running environment of the application. In the next section, I will discuss the development approach of AngularJS framework and some of the core features it provides.

3.1 What is AngularJS

AngularJS is a JavaScript framework developed and maintained by the Google Angular team and the open source community. It is a structural framework for building robust and dynamic web applications. Angular is primarily used to build single-page web applications, although it can be used also for generic web pages. In order to achieve this, the framework provides developers' features, those are essential of modern web application and the development process, such as:

- Separation of concerns for client side code
- Wrapper of asynchronous operation
- IoC with dependency injection
- Wrapper of browser object of which functionalities varies across browsers
- Giving full controller of testing process

  And more [12].

Regarding architectural approach from 2012, it is evident that AngularJS is a MVW framework, which stands for Model-View-Whatever. Whatever stands for “whatever works for you”, because in the most recent implementation, there is a lot of flexibility to separate presentational logic and business logic. In other words, the component called Controller in the framework, the classical functionality of which is to expose models to the view. Now it can be utilized as a decorator function that is used to decorate the scope of the view by declaring public properties and behaviours in the scope. This characteristic ultimately makes the approach closer to MVVM rather than MVC [13].

3.2 Components of Angular application

3.2.1 Conceptual Overview

Before proceeding to the Angular components, let’s preview the overall flow of modern web application to determine the place and use of Angular. Figure 11 gives a general view on an SPA and its ecosystem. In a modern web application, some of the business logic can be also implemented on the client side, such as validating input form, processing and aggregating data before sending to the server, and also filtering data for the presentation based on the current session.

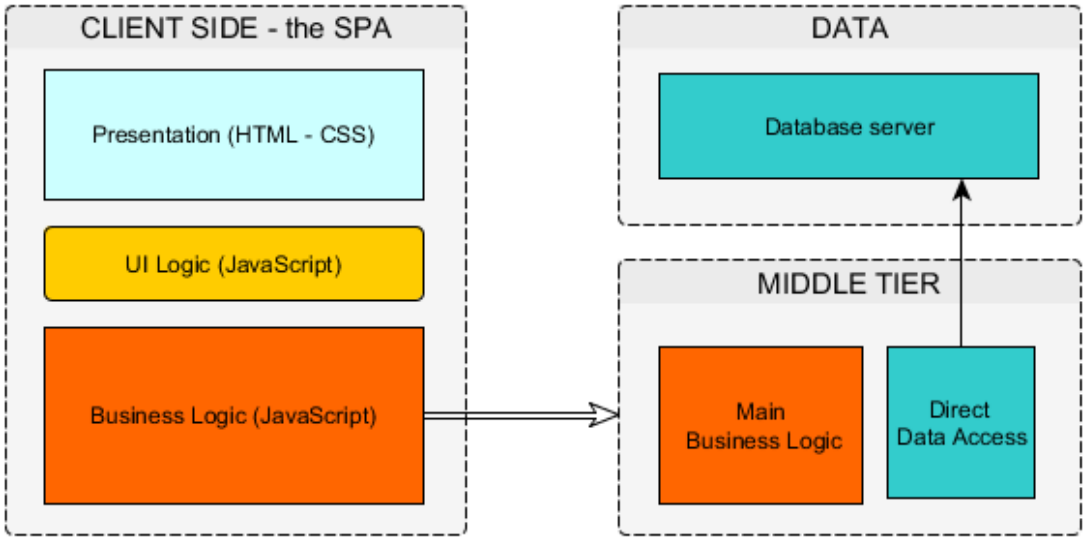

Figure 11. Three layers application with Single Page Application on the client side. Screenshot.

The mentioned client-side business logic can be implemented only with JavaScript. As the application expands, it makes sense to have a clear architectural approach to manage the business logic along with UI logic and presentation. This is where AngularJS framework comes in. To implement the MVW architecture into client-side of this ecosystem, AngularJS introduces the concept of Model, View, and Controller. The simplest concept is View, which is the DOM that user the sees. Model is the data shown to the user in the View and with which the user interacts. Controller contains business logic behind the View, its data and the user interaction. The next paragraphs will discuss how these components interact with each other.

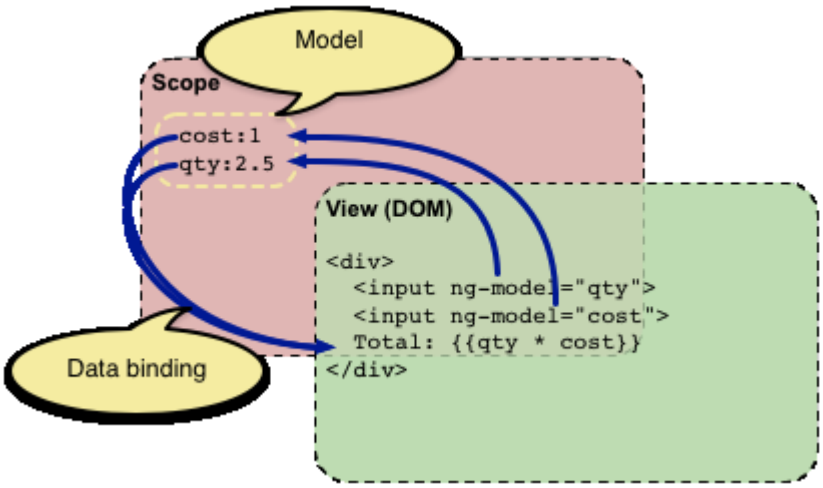

Figure 12. Relationship between View and Model in Angular. Reprint from AngularJS 2010 [14].

Figure 12 demonstrates the relationship between the View and the Model. In the figure, the View (DOM) contains what is known as a template, which is simply plain HTML with additional markups. The additional markups in question are the attribute *ng-model* and the double curly braces: *{{ }}*. The first one is called Directive, which is an Angular component that can apply special behaviour to attributes or elements in the template [14]. In this case, the behaviour that attribute directive applies is to bind the value of that input field to a model with the given name. The second kind of additional markups is called an expression. This expression will be noticed by AngularJS when it traverses the DOM, and will be evaluated as JavaScript code as if it was a JavaScript expression. With the help of these additional markups, models in the view can be evaluated. However, what was not discussed is where the model is contained and where the controller is in the picture.

Each template in the application has a certain scope, which gives access to certain models. The extent of this scope is defined by a Controller. Figure 13 describes how I can use different controllers to manage the available models in different partial templates of the document. The important feature arising from this example and also from Figure 12 is that whenever the model value changes, the expressions in the view will be automatically evaluated and the DOM is updated with the new values. The concept behind this feature is called two-way data binding and it is one of the most important features of AngularJS as aforementioned in section 3.1. I will not go into the details of the concept because it is related to the internal implementation of AngularJS, which is not in the scope of this study.

Figure 13. Relationship between Scope and Controller. Reprinted from AngularJS 2010 [15].

Other than Model, View, and Controller, Service is also one of the core concepts in AngularJS. The next section will discuss briefly this service with some examples. However, before proceeding to AngularJS services, the AngularJS concepts discovered, are recaptured in table 5 below. It is necessary to grasp these core concepts of AngularJS before going deeper into AngularJS services and the example application further in chapter 4. The concepts are presented in the same order as they are presented in this chapter.

Name	Short description
Model	The data shown to the user in the view and with which the user interacts.
View	The DOM that the user sees.
Controller	Container for the business logic behind views.
Directive	Additional markup that extend HTML with custom attributes and elements.
Expression	Valid JavaScript expression that is used to access variables and function from the scope.
Template	HTML with additional markup such as Angular Directive.
Scope	The context where the models are stored.
Two way data-binding	The mechanism that sync data between model and the view

Table 5. AngularJS conceptual overview. Modified from AngularJS 2010 [14].

3.2.2 Angular Services

Before discussing services, I would like to explain the context why services are created. Set aside the DOM part of a web application, a web application is then composed of objects that collaborate with each other to get things done. Some of the types were discussed are Controller, Scope and Model. In the application implementation, these objects need to be created and then wired together for the app to work. In an AngularJS application, the Controller, Scope and Model objects are always related to a certain context or a template. Because the scope of each template is different, models from one template are by nature, not accessible from another. What if states and data need to be shared between different contexts, and what if some states need to be stored in the global context, or even in the browser cache, so another template can reutilize it now or in the future? The answer is AngularJS services.

AngularJS services are substitutable objects that are wired together using dependency injection [16]. Services can expose methods that help the application persist states and behaviours for its lifetime, and communicate across controllers, or even across other services. Because of the essence of being sharable, AngularJS services are singletons. In other words, any AngularJS component that depends on a service gets the same reference to the single service instance, which is used for the whole lifetime of the application. For optimization, a service in angular is only instantiated until a component depends on it, which is also known as lazy instantiation.

Before proceeding with the details of AngularJS services, the dependency injection concept needs to be understood. Dependency injection is a form of Inversion of Control (IoC), which in short is a pattern about removing dependency from the code [17]. Figure 14 presents an abstract model of how dependency injection works. Instead of ultimately declaring *MovieFinderImpl* as a dependency of *MovieLister*, an interface can be used to describe the dependency, and the runtime dependency is resolved by mapping the interface to the desired class. Later when a dependency is needed, the injector, or in the figure 14, the *Assembler* will replace the dependency with the configured instance.

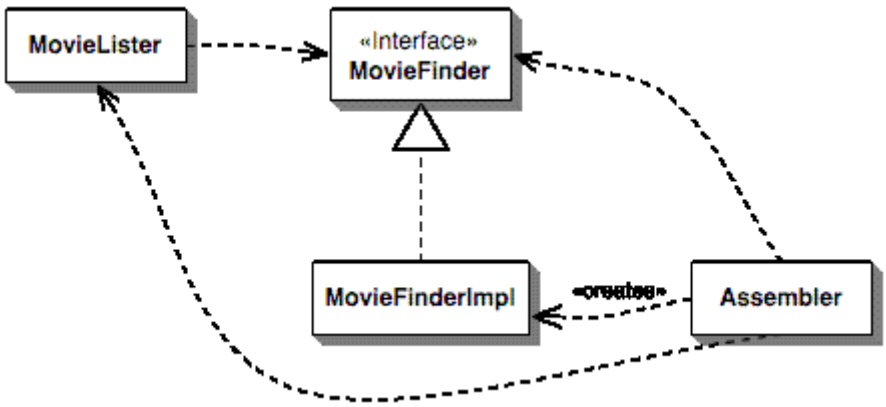

Figure 14. An example of Dependency Injection. Reprinted from Fowler 2004 [18].

Recalling from the previous section 3.2.1, a Controller is in fact dependant on a *$scope* object. This kind of dependency is resolved at runtime involving two main parties. The first one is the *$inject* property that exists in any AngularJS JavaScript component, of

which value is an array of dependency names. The other party is the *$injector* – a built-in AngularJS service that is responsible for retrieving object instances as defined by providers, instantiating types, invoking methods, and loading modules [19]. The creation of Angular services and the dependency on Angular components on Angular services will be resolved using the same dependency injection concept as described earlier.

AngularJS comes with a lot of built-in services and the name of a built-in service always starts with the dollar sign *$*. These services are the pillars of any modern web application. For instance *$http* service is one of the core Angular services that encapsulates the low-level implementation of XHR by exposing methods to communicate with remote server. In relation to and often used in pair with *$http* service is *$q* service, which is a higher level implementation of promises and deferred objects to help you run functions asynchronously, and use their return values when they are done processing [20]. Figure 15 demonstrates the APIs of these services.

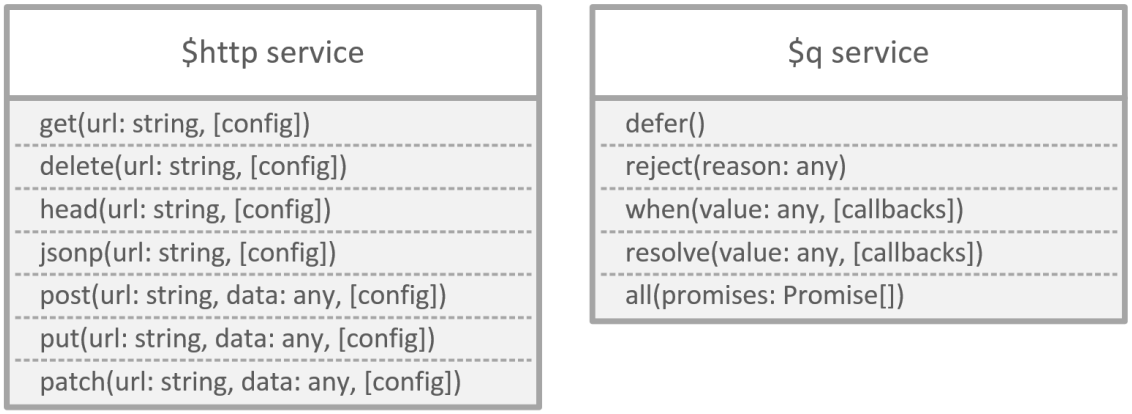

Figure 15. APIs of Angular built-in services *$http* and *$q*

Despite the fact that there are many built-in services included with the framework, these services are intuitively more generic than tailored to a particular use case in practice. AngularJS developers are able and in most cases will create their own services by providing the service name and the service recipe function. The code example in listing 1 demonstrates how a helper service can be created to utilize the Local Storage API from a browser.

```
class LocalStorageService {

    static $inject = ["$window"];
    constructor(private $window: ng.IWindowService) {}
```

```
    get<T>(key: string): T {
        var serialized: string = this.$window.localStorage.getItem(key);
        var data: T = <T>angular.fromJson(serialized);
        return data;
    }
    set<T>(key: string, data: T) {
        var serialized: string = angular.toJson(data);
        this.$window.localStorage.setItem(key, serialized);
    }
    remove(key: string) {
        this.$window.localStorage.removeItem(key);
    }
}

var myApp: ng.IModule = angular.module("myApp", []);
myApp.service("localStorageService", LocalStorageService);
```

Listing 1. Creating a service with service recipe function.

Service recipe function is one of the many ways to create a service. This recipe function provides a very intuitive way to induce object-oriented code into the client side web application. The function *LocalStorageService* is the recipe in question, or can be also conceptually considered as the constructor function to create the instance of the service. When the *localStorageService* is needed, Angular *$injector* invokes the *LocalStorageSevice* with the *new* operator [16], which will result in an instance of the service that is used for the whole lifetime of the application. The constructor function can also take arguments, which are the dependencies of the current service. In the example in figure 16, *localStorage* service depends on *$window*, which is a built-in AngularJS service that references the browser window object. The next example will show another way of creating a service by using factory recipe function. Factory recipe function looks very similar to the service recipe function. However, the main difference is that the service will be instantiated, not using the *new* operator, but by calling the function itself and returning the result as the service instance. The code example in listing 2 demonstrates the process of creating a request interceptor service using the factory function. The interceptor is then added to the interceptors' container, using the *config* function, with the help of *$http-Provider*, a built-in Angular service used to change the default behaviour of the *$http* service.

```
// Given OauthSessionService is a predefined service
// that persist authorization data of the current session
myApp.factory('sessionInjector', ['OauthSessionSerfvice',
    function (OauthSessionService) {
        var auth = OauthShimService("twitter").getAuthResponse();
        var sessionInjector = {
            request: function(config) {
```

```
                if (!OauthSessionService.isAnonymous)
                    config.headers["x-session-token"] = auth.access_token;
                return config;
            }
        };

        return sessionInjector;
    }
]);

myApp.config(["$httpProvider", function($httpProvider) {
    $httpProvider.interceptor.push("sessionInjector");
}]);
```

Listing 2. Creating a service with service factory function.

One code example in this section is the code presented above and it is written in TypeScript and the other one is written in plain JavaScript for ease of transition without a strong TypeScript background. The next session will discuss how TypeScript can be used across an Angular application. After section 3.3, most of the examples will be demonstrated with TypeScript, including annotation when necessary.

3.3 Angular Component with TypeScript

There is a common factor involved when an AngularJS component is created, be it services, controllers or other angular components such as directives and filters: a blueprint. Regardless of how a blueprint will be used, the result will be an object based on that blueprint. This also applies for built-in AngularJS service such as *$timeout* service – a wrapper for *window.setTimeout* functionality. Listing 3 is the implementation of built-in AngularJS service *$timeout*. Source code is an excerpt from GitHub Angular 2016 [21].

```
'use strict';

function $TimeoutProvider() {
    this.$get = ['$rootScope', '$browser', '$q', '$$q', '$exceptionHandler',
        function ($rootScope, $browser, $q, $$q, $exceptionHandler) {

            var deferreds = {};
            ...
            function timeout(fn, delay, invokeApply)...
            ...
            timeout.cancel = function (promise) {
                if (promise && promise.$$timeoutId in deferreds) {
                    deferreds[promise.$$timeoutId].reject('canceled');
                    delete deferreds[promise.$$timeoutId];
                    return $browser.defer.cancel(promise.$$timeoutId);
                }
```

```
                return false;
            };

            return timeout;
        }];
}
```

Listing 3. Implementation of built-in AngularJS service $timeout [21].

This raises the question of what if the blueprint need to be extended, or reused as a base to create other blueprints that share a mutual contract so that it can be replaced by each other in the future. The blueprint and the related matters resemble the class concept in class-based programming, which is not the paradigm that can be easily implemented in JavaScript, because JavaScript is a prototypal-based programming language. This is where TypeScript classes and interfaces come to the rescue. Let's approach AngularJS in the class-oriented way. The next TypeScript code example in listing 4 demonstrates that inheritance can be utilized with the help of TypeScript class to minimize the duplicate of method implementation and enable polymorphism by overriding those methods in derived controller class, as well as add derived class specific properties.

```
class BaseDropDownController {
    constructor(
        $scope: ng.IScope,
        $timeout: ng.ITimeoutService,
        type: string = "")...

    get numberItemsShown()...
    open()...
    close()...
    select()...
}

class SelectDropDownController extends BaseDropDownController {
    constructor($scope: ng.IScope, $timeout: ng.ITimeoutService) {
        super($scope, $timeout, "select");
    }
    open()...
}

class TypeAheadDropDownController extends BaseDropDownController {
    constructor($scope: ng.IScope, $timeout: ng.ITimeoutService) {
        super($scope, $timeout, "typeahead");
    }
    get typeaheadLimit()...
    clearSuggestions()...
}
```

Listing 4. Inheritance and polymorphism capabilities with TypeScript.

Side by side with this class is TypeScript interface. In AngularJS, along with context-aware support, TypeScript interfaces can utilize the dependency injection's IoC concept with the help of seam objects, as demonstrated in the UML model in figure 19. By using *IReportService* as a seam object in *ReportController* constructor function, the dependency can be replaced, for instance, with *ReportServiceMock* to later serve the unit testing of *ReportController*.

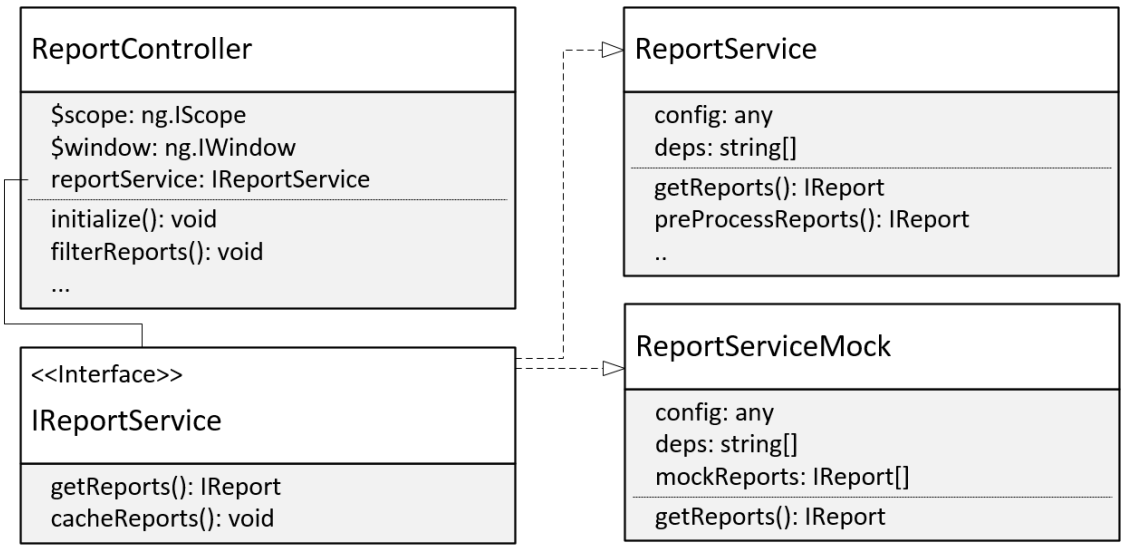

Figure 16. Inversion of control with TypeScript using seam object. Screenshot.

It is important to remark that there is no concept of class nor interface in the JavaScript language specification. This is made possible in TypeScript because in the compiling process to JavaScript, the compiler takes care of converting class-based object-oriented code into valid prototype-based object-oriented JavaScript code. The internal of how TypeScript is compiled to JavaScript in not included in the scope of this study. In the next chapters, only TypeScript will be used for code examples for the overall architectural purpose.

4 Designing the case project

The trading platform provided by Company X encompasses a wide range of operations such as trading, clearing, settlements, reporting and so on. As a typical enterprise level application the platform is implemented with multiple layers, one of which is the trading portal to the user – the web single page application. Figure 17 provides a quick overview of the application layers in the platform. The diagram in figure 17 focuses on the layers related to the topic of this study. In a real scenario, these layers can also include components such as load balancer, proxy API, blob storage, and so on.

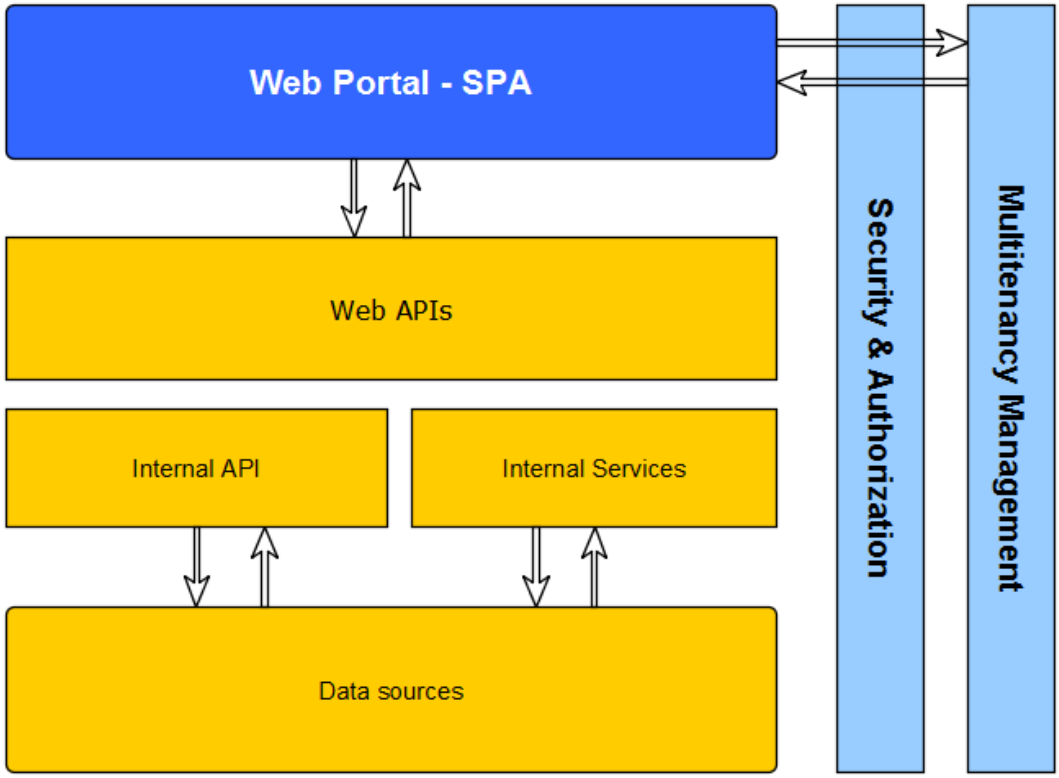

Figure 17. Diagram of the case project's application layers. Screenshot.

As can be seen from the diagram, the external endpoints that the web client interacts with directly, are the Web APIs, the Security API, and the multitenancy API. Security and authorization are common among any web application. Multitenancy enables partner users of Company X to use the platform with their own UI-related customization. Through multitenancy, an application can be reused with different branding appearance. Also multitenancy can work along with authorization service to, for instance, determine the scope

of the certain user in relation to accessibility to data in the platform. These endpoints will also be considered when designing the structure of the SPA web application. These endpoints are also essential to the user flow of the web portal, which is demonstrated in flow diagram in figure 18.

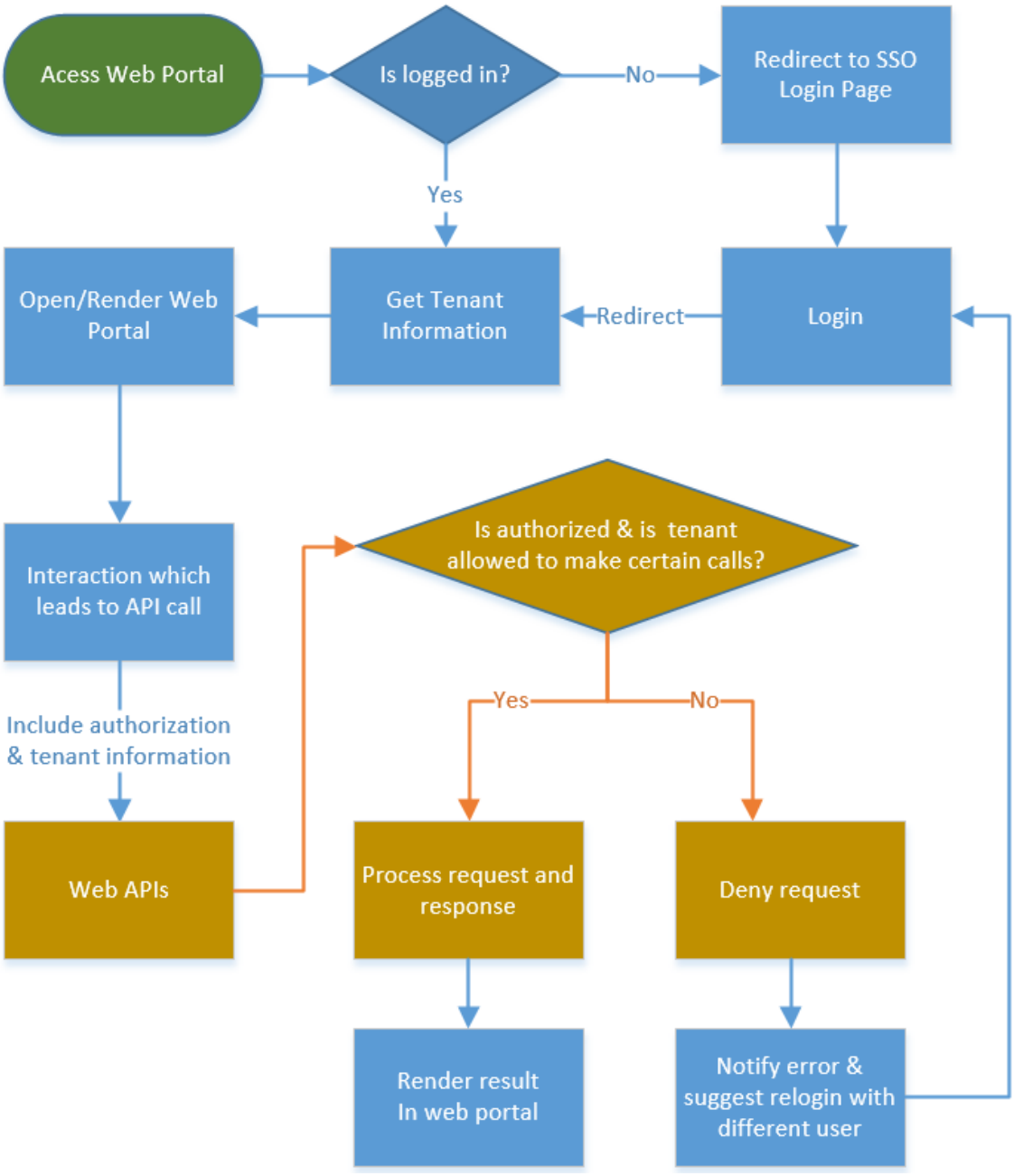

Figure 18. Common user flow starting at the web portal. Screenshot.

Modern enterprise level web applications often involve many build tools such as code linter, live reload system, module loader, and automated tests runner. The web app of

this chapter is no exception. These build tools definitely add a considerable load of preliminary work before actually developing the application, however in the long run and for large web application, they are no doubt beneficial from the development and maintenance perspectives. Table 6 gives a quick glance regarding the roles of some of the most common build tools.

Tool	**Roles**	**Examples**
Code linter	Flag suspicious usage of constructs and alternative syntaxes	TSLint (TypeScript Lint), JSLint (JavaScript Lint)
Module loader	Structure code, manage dependencies and load order	RequireJS, CommonJS
Concatenation	Reduce the number of resource file to minimize HTTP request	Gulp-concat, Grunt-contrib-concat
Minification and mapping	Reduce the size of resources, map source files even after concatenation and minification to help with debugging	Grunt-contrib-uglify, Gulp-minify, Gulp-sourcemaps
Streaming development system	Watch the resources and refresh the browser when resources have changed	Gulp-livereload, Grunt-contrib-livereload
Automated tests runner	Have tests executed the browsers of your choice and the platform of your choices which highly configurable options.	Karma, Testem

Table 6. Build process and their roles in a large web application.

After preparing the development system, it is time to start with the core application. For a web application, one of the many ways to define the detailed requirements of the application is to review the states of the UI and the requirements arises from the UI. Figure 19 is a screenshot of the default state of the application UI.

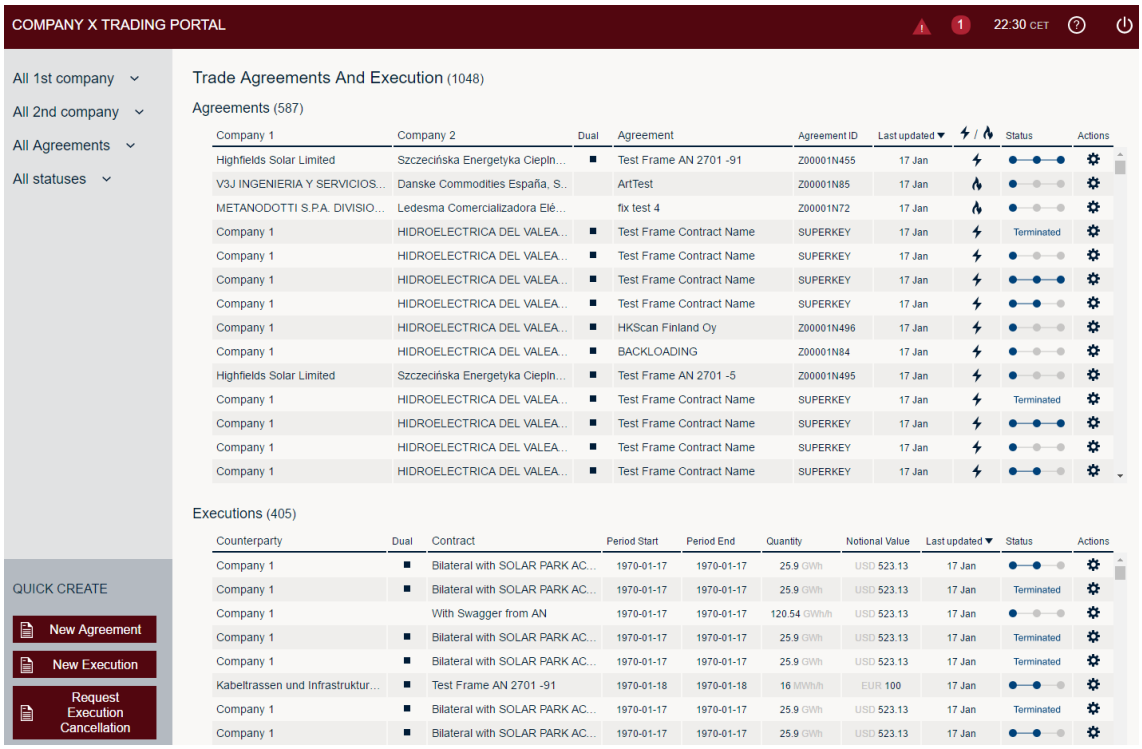

Figure 19. Default state of the application user interface. Screenshot.

As specification definition is not the main topic of the study, other UI states will not be demonstrated and most of the requirements will be assumed for a trading platform, so that the study can continue to focus on the architecture approach. Based on the UI, the core components of the application can be determined as follow:

- Application bar component: to render tenant related information on the top bar and also to display messages and errors.
- Grid component: to display grid of agreements and executions
- Filter component: to display the filter sidebar and to filter the visible agreements and executions by different criteria.
- Modal component: to display modals so user can use to create a new agreement, execution, or to cancel an execution.

In relation to external endpoints mentioned in beginning of chapter 4 based on diagram in figure 17, these services are also required:

- Authorization service: to cache or access authorization information and to authorize the API calls made from the user.

- Multitenancy service: similar to authorization service, to fetch related UI information of the current user and inject them to the application UI and to the API calls' headers as well.

At this point it is clear what components will be needed for the SPA web application, with an assumed ready development environment. The next chapter will discuss the architectural approach on how to, in practice, implement these components in AngularJS with the help of TypeScript. With the help of TypeScript, the components' codebase should be semantically understandable and cohesive, while the application is also scalable and is easy to maintain.

5 Implementation of application components

5.1 Architecture approach

Organizing components are a crucial step in starting the development of an SPA application. The term component does not refer only to AngularJS component, it can also represent the abstract concept of an application component, that fits in the application as a whole, and even better, can be understood semantically in the actual code implementation. For a concrete example of this argument, I will use the visualization of component hierarchy and code demonstration. Based on the provided UI in figure 19 in chapter 4, figure 20 shows the intuitive perception of the components' hierarchy.

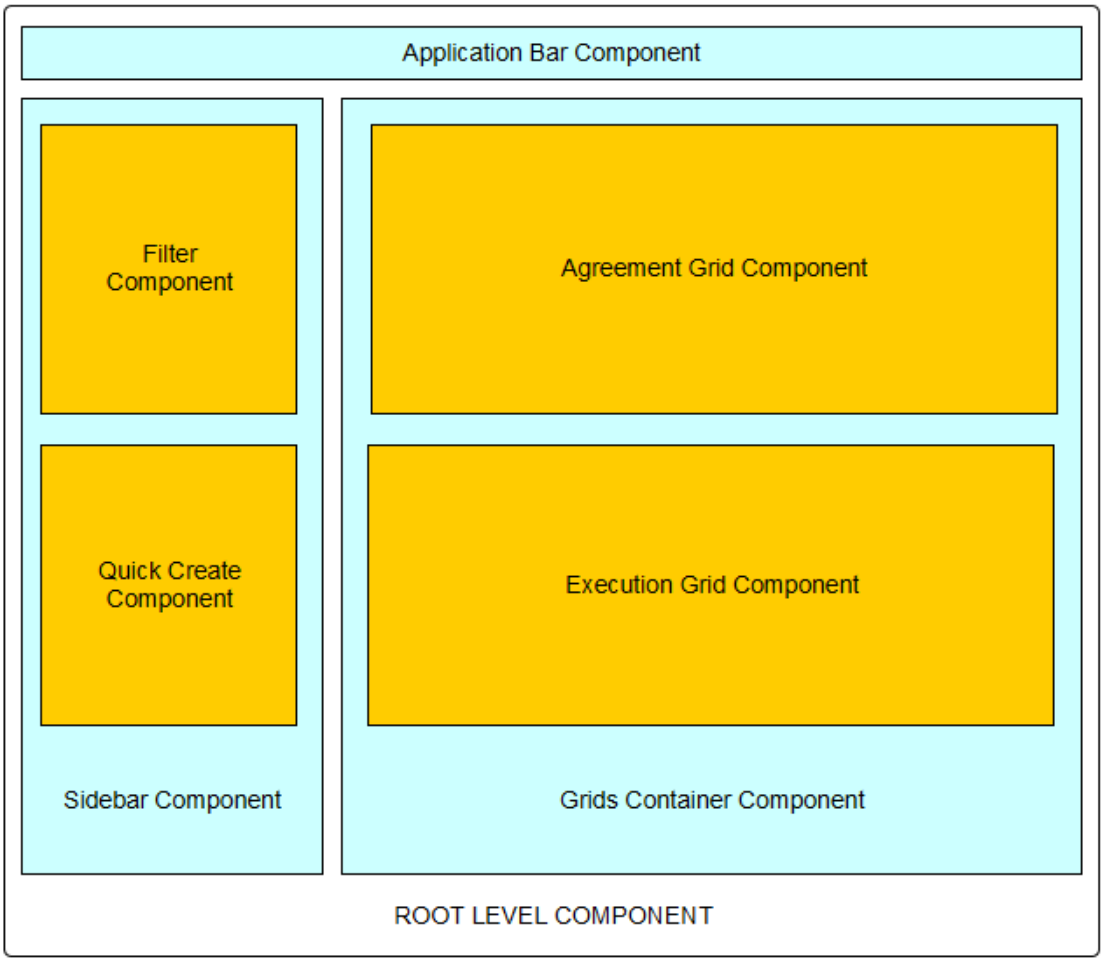

Figure 20. Application components

The HTML code in listing 5 demonstrates how the components hierarchy would look like, as if each component is an AngularJS directive.

```
<application-layout>
  <sidebar-component>
    <filters></filters>
    <quick-create></quick-create>
  </sidebar-component>
  <grids-container-component>
    <agreements-grid></agreements-grid>
    <executions-grid></executions-grid>
  </grids-container-component>
</application-layout>
```

Listing 5. Application components as AngularJS directive.

An alternative to organizing the component is to group the sidebar and the grids container into one component named e.g. *"main component"*, instead of dividing them right at the root application level. I agree with the intuition in both of the approach, however there is a mechanism in AngularJS that would strongly support the former solution over the latter. The mechanism is event broadcasting, emitting, and listening with the *$scope* component.

The event in event broadcasting and emitting within a scope context is simply an event name of type string. Emitting is the process of dispatching an event name upwards through the scope hierarchy and notifying the registered listeners. Broadcasting is the other way around, it dispatches an event name downwards to all child scopes and their children, and also notifies the registered listeners. The related method for these processes are *$emit* and *$broadcast* in *$rootScope.Scope* component [22]. From the same component, *$on* method can be used to create event listeners. The method takes the first parameter as the event name and a function as the second parameter to handle the event. It is important to take into notice that, with the descriptions of *$emit* and *$broadcast*, the only two methods that can propagate events, it is not possible to broadcast an event to a sibling component scope. The broadcast and emit path can only be vertical across the scope hierarchy, not horizontal.

With the event mechanism in mind, let's take a look at how it would help with the functionalities of the application. First of all, to provide a context, these are some straightforward examples of event in the current case application:

- Filters have changed.
- An agreement/execution has been created.
- Agreements received and grid finished rendering.
- Filters' layout has changed.

Figure 21 demonstrates the usage of events propagation in the case application.

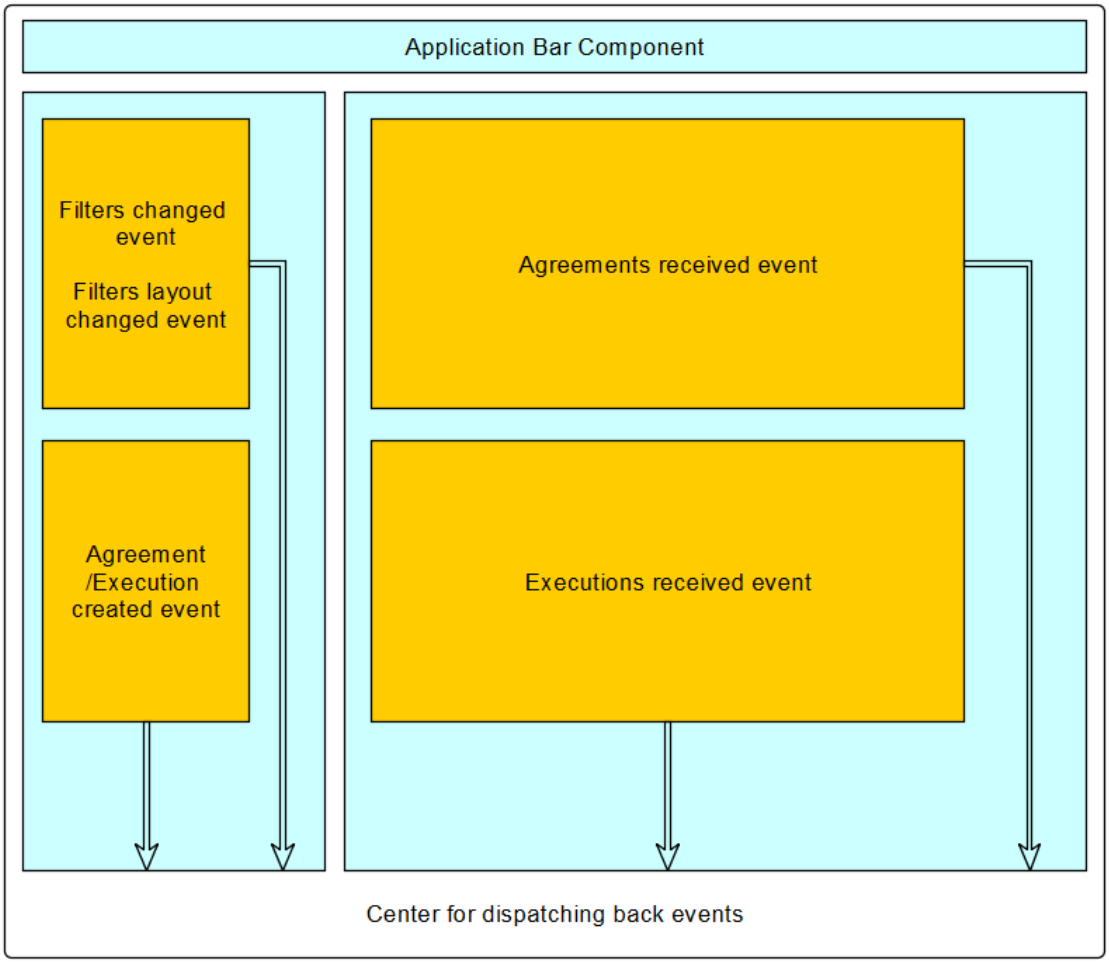

Figure 21. Event propagation in the current case application. Screenshot

Component names are hidden in order to events to be easily comprehended. For a review of the components hierarchy, please visit the diagram in figure 20. It shows that all events emitted will eventually find a common parent scope, which in this case is the scope of the root level component. To reach this common parent scope, "Filters Changed Event", for instance, is also emitted through "Sidebar Component" scope. The reason that I favour the component hierarchy demonstrated in figure 20 among others is that events will then need to be emitted through the least component scopes, in order to

reach the common parent scope. This way, each scope is not polluted with events that have nothing to with it. Now with a common ground for dispatching event, the rest of the work is very simple and intuitive. Figure 22 demonstrates the application flow for the event of filters being selected.

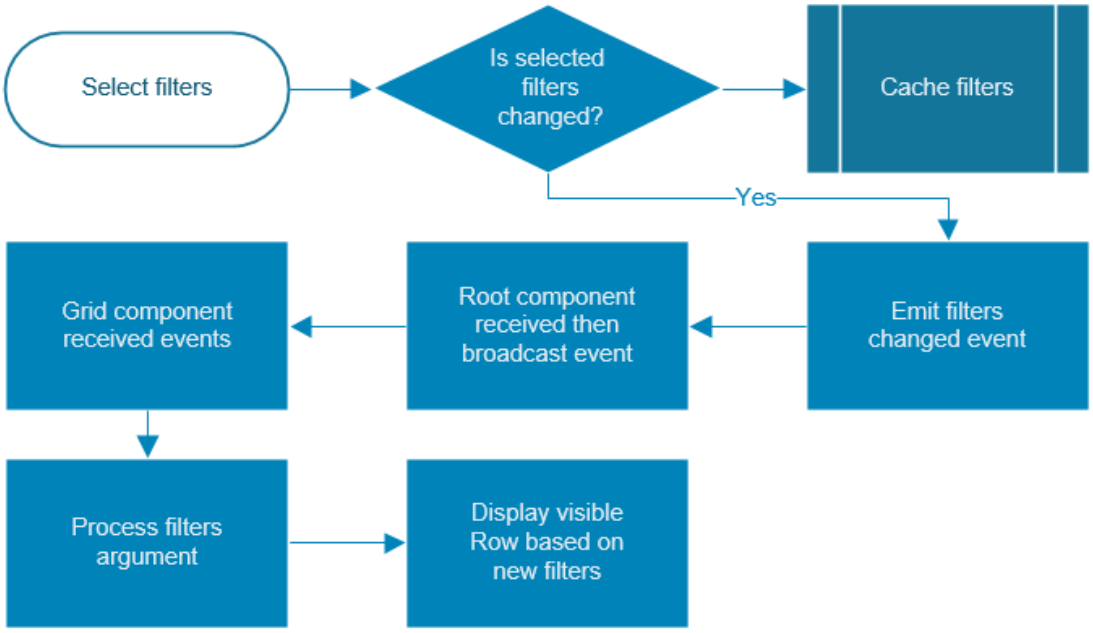

Figure 22. Flow of processes when new filters are selected. Screenshot

The example in figure 22 is a very good use case of observer pattern, which is relatively common among MVC framework, in this case is Angular. The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods [27]. It is mainly used to implement distributed event handling systems.

Without the use of events, the other way to easily communicate between components in different branches is through the *$rootScope*. Every AngularJS application has a single *$rootScope* and all of the component scopes are descendant of *$rootScope*. *$rootScope* also includes all the *$watch* functionalities that are discussed with *$scope*. *$watch* functionality comes with a different depth. Related to this example, it is necessary to understand the performance difference between different depths, so that one can decide when to use *$watch* functionality with *$scope* or *$rootScope* and when to use event model to communicate between components. *$watch* functionality can operate with three different depths. Figure 28 demonstrates how *$watch* functionality can behave differently. Green

tick means *$watch* can detect the change. Red mark indicates the failure of the change detection.

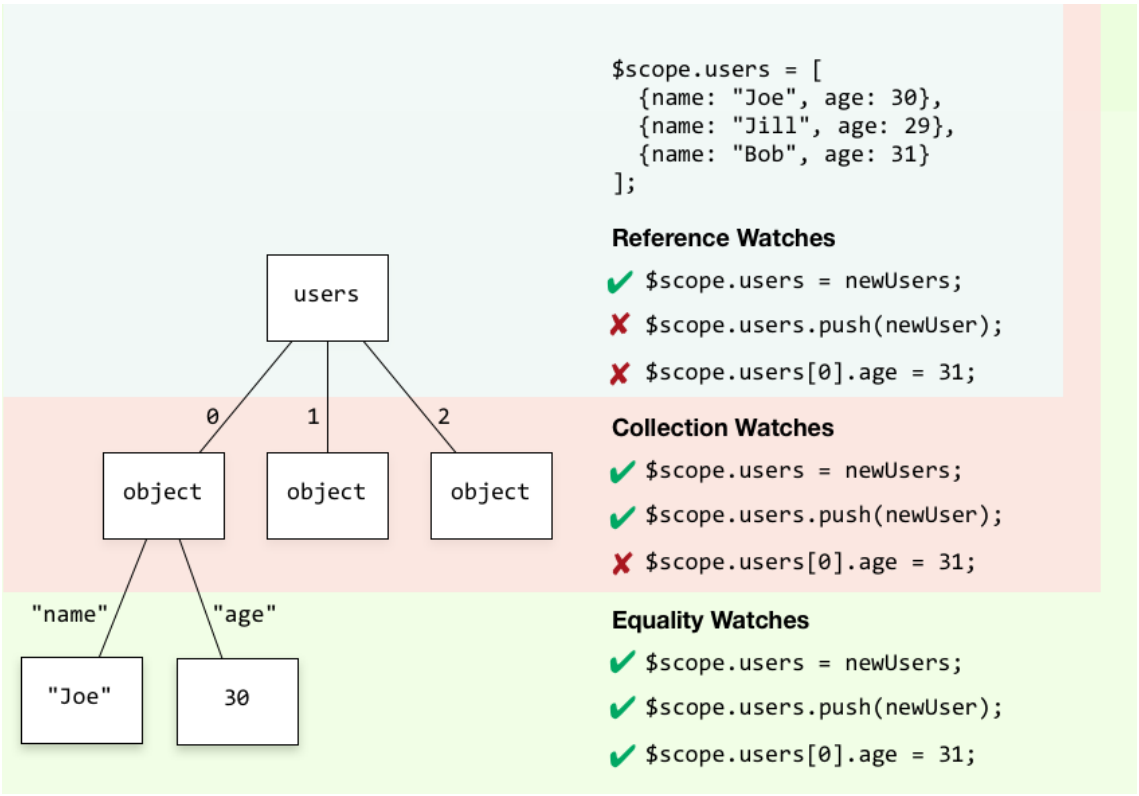

Figure 23. The three watch depths of AngularJS. Reprinted from Tero [28].

The list below is the detailed description of how *$watch* functionality of each depth would function.

- ***scope.$watch*** (*watchExpression*, *listener*) : detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected.
- ***scope.$watchCollection*** (*watchExpression, listener*) detects changes that occur inside an array or an object: When items are added, removed, or reordered. The detection is shallow - it does not reach into nested collections. Watching collection contents is more expensive than watching by reference, because copies of the collection contents need to be maintained.
- ***scope.$watch*** (*watchExpression, listener, true*) detects any change in an arbitrarily nested data structure. It is the most powerful change detection strategy, but also the most expensive. A full traversal of the nested data structure is needed on each digest, and a full copy of it needs to be held in memory.

Back to the current case application, a modern web application can have a lot of states, even for a single page application. Those states must be organized related to where it is used and how it is monitored. *$rootScope* is an ideal place to hold states, which are relevant across the application and are useful too all components. However, deep watching with *$rootScope* can sometimes be too expensive for the performance of the application, if the state is related to a change in the large model, such as an object that contains a data set from a remote server. This is where the event model where the process of broadcasting and emitting will be only relevant to the listeners.

To understand better how components can be organized, let's proceed with the detailed structure of these components. In the next section, I will discuss the structure of each component, their dependencies related to the AngularJS services.

5.2 Component detail structures and services

For clearer understanding, I would like to separate the current application components into 2 categories: UI-components as angular directive, and helper components as angular services. In order to provide the component in the hierarchy provided in the diagram in section 5.1 figure 20, AngularJS providers, such as directive, controller and service must be created. Figure 24 proposes a structure for some of the UI-components.

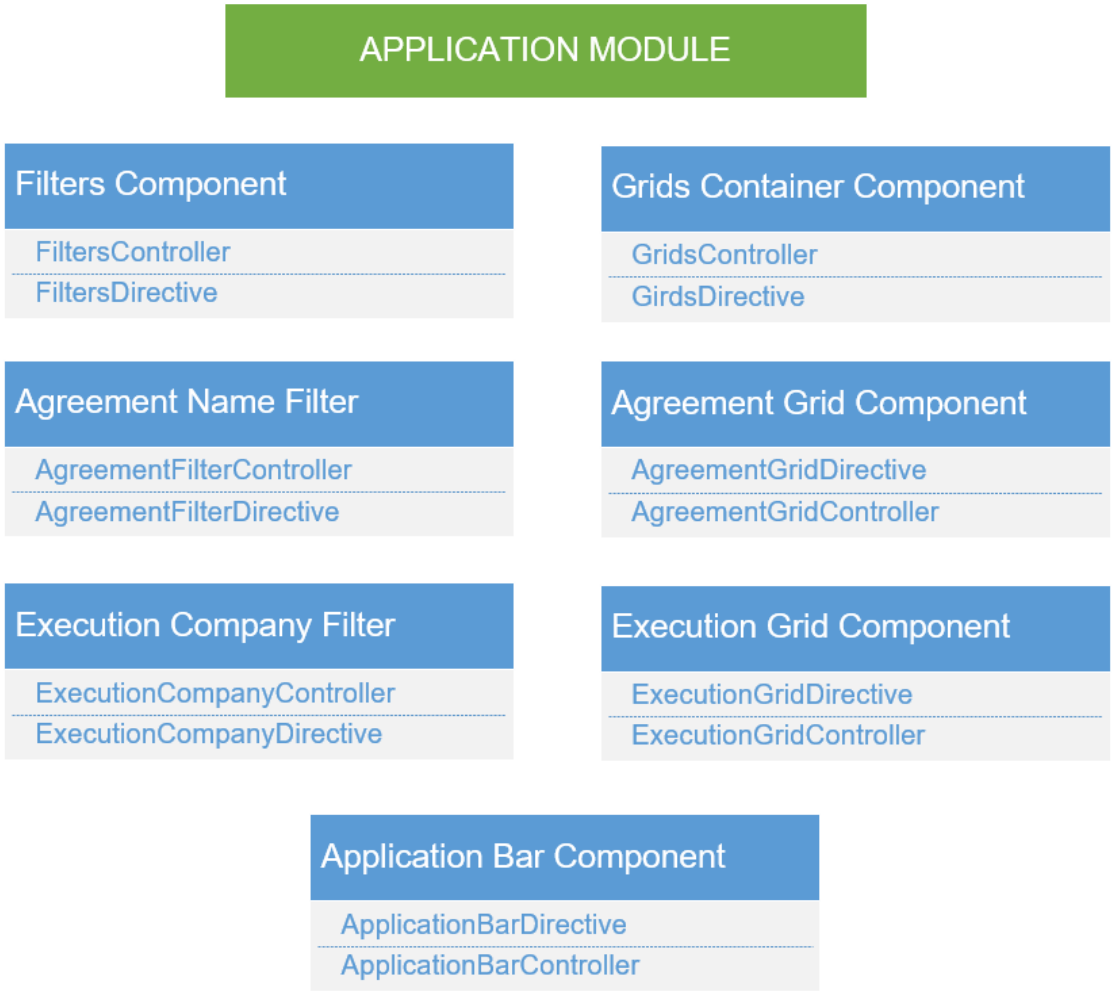

Figure 24. Structure of UI-components. Screenshot.

As can be seen in the component structure in figure 24, filter components are very similar to each other, and the same context applies for grid components. However, we must remember that component is an abstract concept and the concrete one consists of actual providers. Since every provider needs a recipe, or a class, it is quite intuitive to create a base class, for instance, for *ExecutionGridController* and *AgreementListController*. Before demonstrating the base class in an actual implementation, I will go through the other type of component in the section: AngularJS services, because the providers mentioned above, will in fact be dependent on one or many of these AngularJS services. Figure 25 describes the dependencies of providers on services, including dependencies of services on other services.

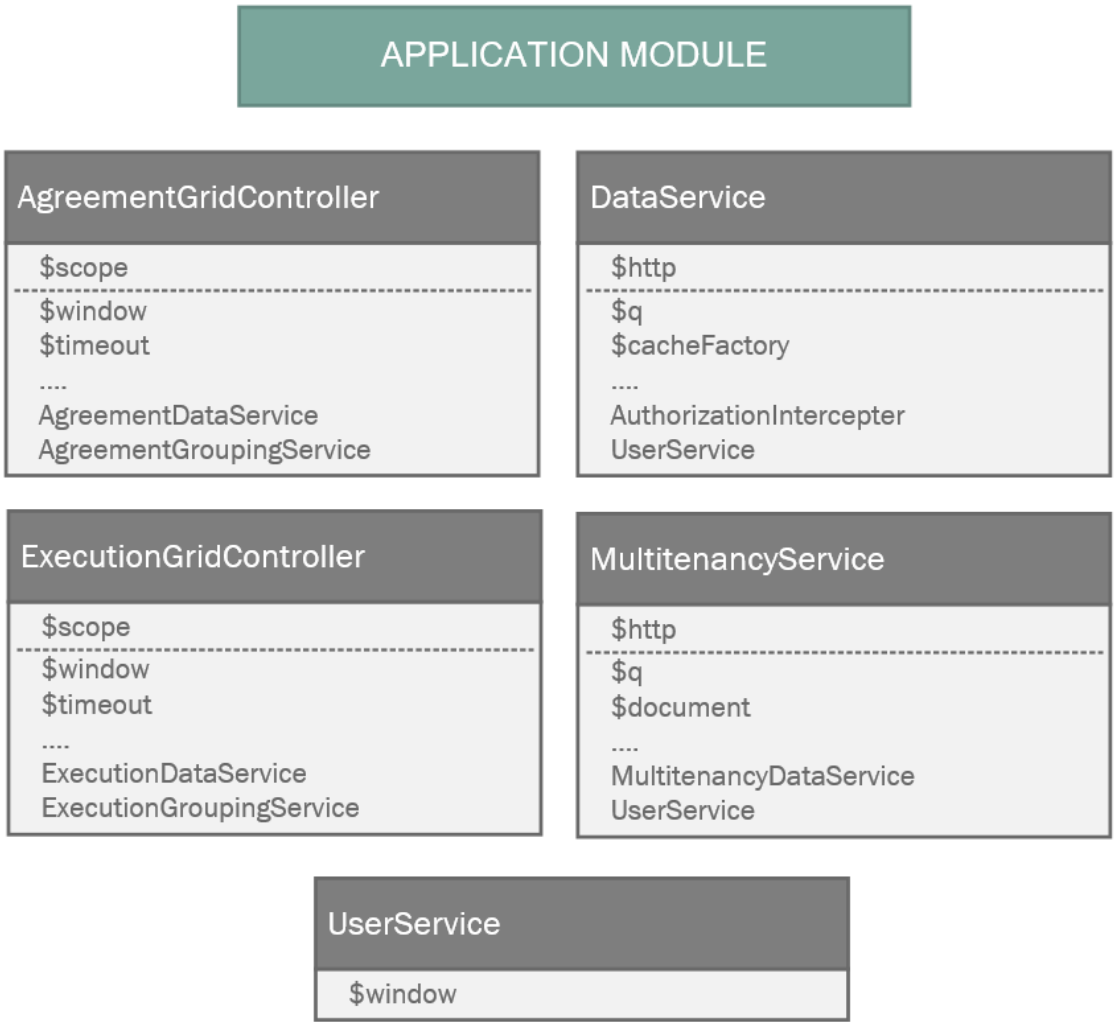

Figure 25. Dependencies of providers on services.

In the UML-like model in figure 30, I have changed the theme of tables to distinguish with the model from figure 29 because these are not anymore abstract components. These are actual AngularJS components which are the core of the current case application. To have a better understanding of the context of these services, table 7 provides the purposes for some of the custom services, since built-in AngularJS services – the ones start with a dollar sign – are already reviewed in section 3.2.2 Angular Services.

Services	Roles
DataService	Wraps the low-level implementation of REST API method call, provided by *$http* & *$q* service. If possible, use this service to cache method call's data through this services.
Agreement-DataService	API usually return generic results those can be used for different presentation. This service should exposes grouping and aggregation method for Agreement data.
UserService	With help from localStorage in *$window*, stores expose user information, e.g. session token to other services in the current application life cycle.
Authorization-Intercepter	Intercepts outbound, and possible inbound requests. Main use is for outbound request, to authorize the request by injecting the current session information.
Multitenan-cyDataService	Derived from data service, this service should fetch and expose specific tenant related information, and through e.g. *$document*, modify UI according the tenant branding information

Table 7. Purposes of services

I have named most the services in the case project implementation above rather generically because they are common in any enterprise web application. Imagine having a user using the service demonstrated in user flow in section 5.1, figure 22. To demonstrate the flow related to services, when the user is logged in, in the initializing process, *UserService* in most cases initialized with first priority to analyse the current session of the user. With the result from the *UserService*, *MultitenancyService* then can fetch related UI information for the specific brand related to the user. At this point, UI components such as ApplicationBar, Filters Layout and Grid Containers can already be initialized. *AgreementDataService* and *ExecutionDataService*, which are based on the *DataService*, can utilize session information from *UserService* through *AuthorizationIntercepter*, to make authorized request to the back end.

5.3 Inheritance and consistency

In the current case application components, the possibility for inheritance and ensuring consistency presents itself between the similarity between components, its providers, and especially in its dependency. This section of the study will present examples in using classes and interface types which are supported by TypeScript. The code examples will

have a medium level of specificity since first of all, TypeScript is rather new to developers, with its first stable release in January 2016 [23], and secondly, AngularJS has its own syntax to implement components, so if one is not already familiar with the way of creating AngularJS component, it would be difficult to comprehend the concepts covered by another level of abstraction through TypeScript.
The first example addresses *ExecutionGridController* and *AgreementGridController*. Let's determine the common states and methods expected from a grid controller. Nominated states can be:

- Grid name
- Grid data type
- Visibility of grid
- Grid options
- Rows data
- Filtered row
- State of initialization: loaded or not loaded

Expected behaviours from grid controllers can be:

- Get data
- Group data
- Update data based on filter changes
- Filter row based on new filters

Based on this specification, UML model can be designed as shown in figure 26.

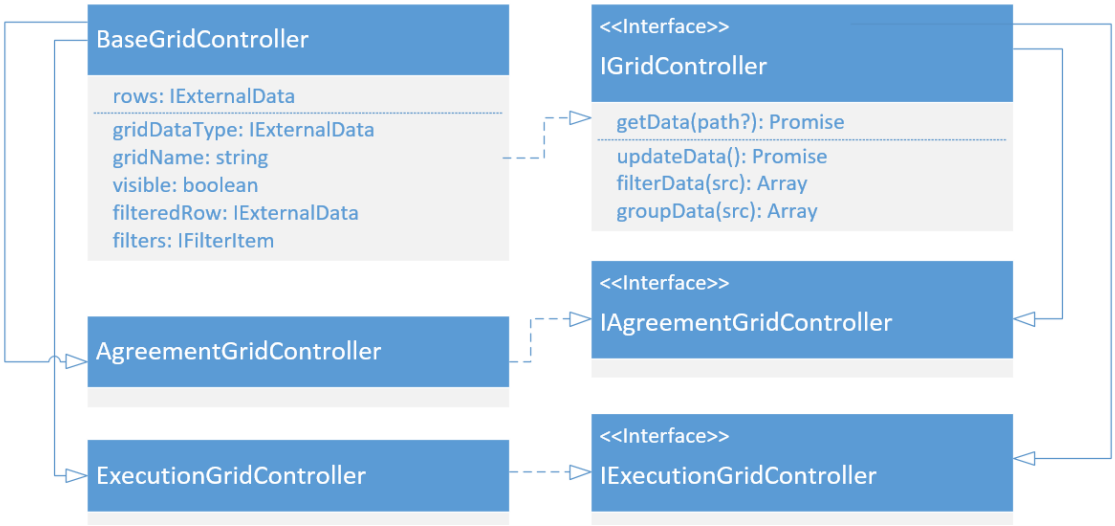

Figure 26. UML for *BaseGridController*, related interfaces and derived classes. Screenshot.

Interfaces in TypeScript can be used to declare both methods and fields, however, from a class-based programming perspective, such as Java, the purpose of interface is to specify the public API, not states. The following code examples in listing 6, which are based on the UML diagram in figure 26, will have the states inheritance implemented in base class and the expected behaviours declared in a public API. Some of the data level interfaces are skipped to focus on the use of classes and interfaces for these components.

```
interface IGridController<TData extends IExternalData> {
    getData(path?: string): ng.IHttpPromise<TData>;
    updateData(): ng.IHttpPromise<TData>;
    groupData(src: TData[]): TData[];
    filterData(src: TData[]): TData[];
}


class BaseGridController<TData extends IExternalData>
    implements IGridController<TData> {

    private rows: TData[];
    public gridDataType: IExternalData;
    public gridName: string;
    public visible: boolean = false;
    public gridOptions: ui.grid.IOptions;
    public filteredRow: TData[];
    public isInitialized: boolean = false;
    public filters: IFilterItem[];
    public path: string;

    static $inject = ["$scope", "$q", "DataService", "GroupingService"];

    constructor(
        public $scope: ng.IScope,
        public $q: ng.IQService,
```

```
        public dataService: IDataService,
        public groupingService: IGroupingService) {

        this.initialize();
        this.$scope.$on("filterChanged", this.filterChanged.bind(this));
    }

    private initialize() {
        this.getData(this.path);
    }
    private filterChanged(newFilters: IFilterItem[]) {
        this.filters = newFilters;
        this.initialize();
    }
    getData(path?: string): ng.IHttpPromise<TData> {
        path = (path) ? path : "";
        var promise: ng.IHttpPromise<TData> = this.dataService.get(path);
        promise.success((json: ng.IHttpPromiseCallbackArg<TData[]>) => {
            this.rows = this.groupData(json.data);
            this.filteredRow = this.filterData(this.rows);
        });
        return promise;
    }
    updateData(): ng.IHttpPromise<TData> {
       return this.getData();
    }
    groupData(src: TData[]): TData[] {
        var groupedByDate = this.groupingService.groupByDate(src);
        return this.groupingService.groupByName(groupedByDate);
    }
    filterData(src: TData[]): TData[] {
        return this.groupingService.multiFilter(this.filters);
    }
}
```

Listing 6. Construction of *BaseGridControllerClass*.

Before extending this base class to create recipe for *ExecutionGridController* and *AgreementGridController*, I will annotate the code example. The static property $inject is one way to define dependencies in AngularJS recipe. It is also presented by AngularJS documentation as the safe way for defining list of dependencies, especially when the code is minified in production [24]. Since models of e.g. Execution Grid' components views are defined in the controller, *private* and *public* keyword is used to indicate which of the components are exposed publicly and should be used as models in view, and which are mainly for the internal methods of the class. The separation of *public* and *private* members can indicate quickly to the new developer the roles of these members in a model perspective. However, for the purpose of unit testing, which will be discussed in section 5.4, some of the fields and methods, which are not used of models, have also been made *public*. Listing 7 demonstrates how the derived controllers' recipes can be done.

```
class ExecutionGridController extends BaseGridController<IExecution>
    implements IExecutionGridController {
    public gridDataType: IExecution;
    public gridName = "List of executions";
    public visible = true;
    public path = "Executions/All";

    static $inject = BaseGridController.$inject;
    constructor(
        public $scope: ng.IScope,
        public $q: ng.IQService,
        public dataService: IDataService,
        public groupingService: IGroupingService) {
        super($scope, $q, dataService, groupingService);
    }

    // Overriding grouping
    groupData(src: TData[]): TData[] {
        return this.groupingService.groupByName(src);
    }
}

class AgreementGridController extends BaseGridController<IAgreement>
    implements IAgreementGridController {
    public gridDataType: IAgreement;
    public gridName = "List of Agreement";
    public visible = true;

    static $inject = BaseGridController.$inject;
    constructor(
        public $scope: ng.IScope,
        public $q: ng.IQService,
        public dataService: IDataService,
        public groupingService: IGroupingService) {
        super($scope, $q, dataService, groupingService);
    }
}
```

Listing 7. Implementation of derived controller classes from *BaseGridController* class.

Inheritance is simple and intuitive as can be seen in the listing. The overriding possibility enables developers to create similar components with their own specific UI information and internal logic. If it needs more dependencies, the $inject property can be extended accordingly. However, an apparent caveat that can be seen from the derived class example is how the constructor has to be rewritten over again, even when the derived components have the same dependencies and are constructed at the same time. At the moment of writing, no plugin nor potential support from future TypeScript versions can be found to simplify this issue. On the positive side, it can be seen directly from the derived recipe, what the dependencies of the current component are.

With interfaces designed from the beginning, the derived components are ensured to have the needed methods. I will also demonstrate the capability of TypeScript tools, for example, TypeScript in Visual Studio, to make sure that the interfaces or contracts are implemented in the components. Figure 27 demonstrates the capability for type safe awareness at compile-time.

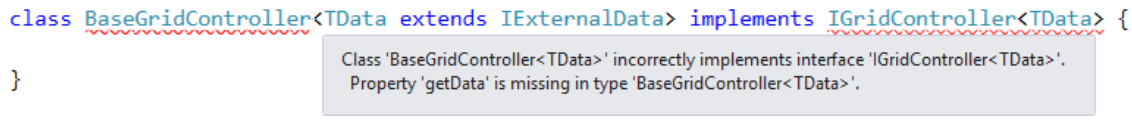

Figure 27. Incorrectly interface implementation warning in Visual Studio. Screenshot.

In response to this error at compile-time, Figure 28 shows the quick actions' help from TypeScript tool in Visual Studio. This quick action is very common among IDE for other classical class-based programming languages such as Java and C#.

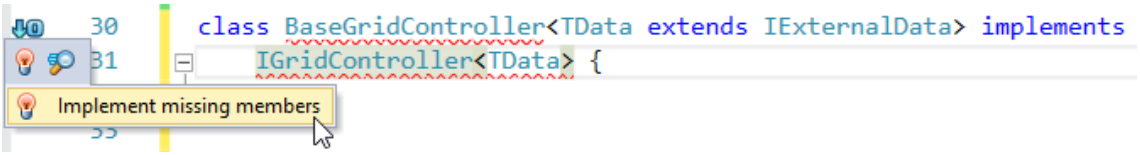

Figure 28. Quick action feature to correct compile-time error in Visual Studio.

Listing 8 is the result for automatic implementation of missing members.

```
class BaseGridController<TData extends IExternalData> implements
    IGridController<TData> {
    getData(path?: string): angular.IHttpPromise<TData> {
            throw new Error("Not implemented");
    }
    updateData(): angular.IHttpPromise<TData> {
            throw new Error("Not implemented");
    }
    groupData(src: any[]): any[] {
            throw new Error("Not implemented");
    }
    filterData(src: any[]): any[] {
            throw new Error("Not implemented");
    }
}
```

Listing 8. Automatic implementation of missing members by quick action feature.

Other than to help ensuring contracts between components, interfaces can also help with the unit test by enabling the use of object seam, so mock components can be injected

freely while still maintaining the contract, in order to test an Angular unit in its own isolated environment. The next section will discuss shortly unit testing in AngularJS, in the context of current case project, with the help of TypeScript classes and interfaces.

5.4 Unit testing with object seam

For the unit testing of this case project, I will use Jasmine (http://jasmine.github.io/) - a behaviour driven development framework for JavaScript that has become the most popular choice for testing AngularJS applications - and ngMock, an AngularJS module that provides support to inject and mock Angular services into unit test [25]. For the scope of this application, I will not cover the basics of unit testing with Jasmine nor ngMock. However, to understand the next code examples, some related concepts are presented as follow:

Keywords	**Roles**
”describe”	is used to group tests together into a test suit
“it”	is used to define a test specification
“beforeEach”	is used to set up the global environment to server the test
“ngMock”	built-in component that provides support to inject and Mock AngularJS services

Table 8. Important keywords in unit testing with Jasmine and ngMock.

Continuing from the conclusion of section 5.3 that is related to the use of object seams with the help of TypeScript interface, an object seam by definition is a place that allows developers to modify the behaviour without modifying the code. Object seem is a very useful feature from the IoC approach, where at the point of constructing a component, the parameter type will not be locked by not specifying a class, but an interface which the parameter’s class implements. Fortunately, every constructor for AngularJS components, including all providers’ type works in an IoC manner. Code example in listing 9 is the constructor of *BaseGridController*:

```
constructor(
      public $scope: ng.IScope,
      public $q: ng.IQService,
      public dataService: IDataService,
      public groupingService: IGroupingService) {
      super($scope, $q, dataService, groupingService);
}
```

Listing 9. Example constructor of an AngularJS component.

By this mean, any new data service and grouping service can be arbitrarily injected into the construction, just by making sure the new parameters still implement the right interfaces, and the construction is still valid. In this particular case for the unit testing of *BaseGridController*, since the original *GroupingService* should expose only synchronous methods, which can still use it in an isolated environment. However, data service involves asynchronous operation with e.g. Http Request, so to make the testing environment isolated, a mock data service must be provided. Listing 10 is an example of how the mock service can be implemented.

```
class MockDataService implements IDataService {
    static $inject = ["$q"];
    constructor(public $q: ng.IQService) { }

    get(path: string): ng.IHttpPromise<IExternalData[]> {
        var defer = this.$q.defer();
        var promise: ng.IHttpPromise<IExternalData[]> = defer.promise;
        defer.resolve({
            data: [
                {
                    id: "decafbb0-ad3f-4e64-b8a3-7a4733bf6af5",
                    name: "A example grid item"
                }
            ]
        });
        promise.success =
            (fn: ng.IHttpPromiseCallbackArg<IExternalData[]>) => {
                promise.then
                    ((response: ng.IHttpPromiseCallbackArg<IExternalData[]>) => {
                        fn(response.data);
                    });
            return promise;
        };

        return promise;
    }
}
```

Listing 10. Mock implementation for an AngularJS service.

With the *MockDataService* class, when get method is called, the data is resolved asynchronously still, but instantly in the next event loop because I have manually resolved

the data. With the mock data in place, the functionality of the *BaseGridController* can now easily be tested in an isolated context – the spirit of a unit test. The example functionality to test here is the filtering feature of the grid. The code example is demonstrated in listing 11.

```
describe("BaseGridController", () => {
    var scope: ng.IScope;
    var $q: ng.IQService;
    var mockDataService: MockDataService;
    var controller: BaseGridController<IExternalData>;

    beforeEach(angular.mock.inject([
        "$rootScope", "$q", "GroupingService",
        (
            $rootScope: ng.IRootScopeService,
            _$q_: ng.IQService,
            groupingService: IGroupingService
        ) => {
            $q = _$q_;
            scope = $rootScope.$new();
            mockDataService = new MockDataService($q);

            controller = new BaseGridController(scope,
                $q,
                mockDataService,
                groupingService
                );
        }
    ]));

    it("should filter data based on Id", () => {
        var filters: IFilterItem[] = {
            id: ["decafbb0-ad3f-4e64-b8a3-7a4733bf6af5"],
            name: [""]
        };

        controller.$scope.$parent.$broadcast("filterChanged");
        expect(controller.filteredRow.length).toEqual(1);
        expect(controller.filteredRow[0].name)
             .toEqual("An example of a grid item");
    });
});
```

Listing 11. Example test specification for *BaseGridController* test in an isolated context.

I will quickly annotate the code example. Firstly, the controller is created with needed dependencies through *angular.mock.inject*, plus the mock dependencies *MockDataService* I have created earlier. Then in the test specification, a filter is predefined and *"filterChanged"* event is also simulated from the parent scope. The rest of the work should be done by the controller because it contains a listener for the event. Based on this test specification, I can already assert the overall functionality of the controller.

AngularJS components are designed carefully with testability in mind. Testing and unit-testing in particular are not new concepts, however the JavaScript syntax may not be too familiar for developers with strong class-based programming background. One should quickly notice which of the codebase can be considered a unit and should be included in unit test. The AngularJS way with the help of Jasmine definitely made that easier, but now with TypeScript and its type system, everything can be very natural and intuitive for developers with a strong class-based programming background, as well as the developers with an already strong JavaScript background.

6 Conclusion

The web is developing rapidly, even for the niche market of single-page web applications. EmberJS, AngularJS, BackboneJS is just a part of the current picture, as the picture changes over time and new parts will replace the old ones. At the moment of writing this study, Angular – also known as AngularJS 2 – is already in a late beta state. Therefore, this study has been focusing more on the abstract concepts that AngularJS and TypeScript present to provide knowledge and research that matter now, and also in the future.

Regarding the objective to develop an approach for the trading platforms that can support future scaling and maintenance, AngularJS and TypeScript have by no doubt been a valid choice. Through implementation, AngularJS has proved its expressiveness through extending HTML with directives and its reusability and testability in its components with the help of TypeScript. TypeScript itself has also proven very useful in the implementation with the type system and the context awareness support. Productivity has been also improved with TypeScript thanks to the support of IDE to prevent compile-time and syntax errors, as well as to refactor the code base.

Scalability, maintainability and consistency, among other issues, are always compromised in any project related to for example the projects' scale, timeline and resources. TypeScript and AngularJS is a strong combination, but it is no doubt a heavyweight solution for any project because AngularJS itself has a steep learning curve, and to have a strong grasp of TypeScript code's behaviour, one must really understand the internal aspects of JavaScript. Because of this, the development pace of the example case project was slow at first, but the later phase of the development was rather intuitive and effortless since the application was well designed and implemented in the beginning with the help TypeScript and AngularJS.

The concepts discussed in AngularJS are already very modular, and as a natural progress, the Angular – or Angular 2 – provides an even cleaner component approach, which is even more modularized. Unfortunately, Angular approach has drastically evolved from AngularJS, thus it is not discussed in the scope of this study. As for AngularJS, it will be still maintained and developed by the Google Angular Team, since AngularJS still has become an essential part of modern web applications, and as for TypeScript, it is definitely becoming the real thing with more and more improved versions released recently [26].

References

1. Inc., Yahoo!, 2011. Mobile Modes. How to Connect with Mobile Consumers, [Online]. 2, 8. Available at: http://www.rafaelfelipesantos.com.br/download/pesquisa-yahoo-mobile-modes.pdf [Accessed 13 March 2016].

2. Mozilla Developer Network. 2015. WebAPI | MDN. [ONLINE] Available at: https://developer.mozilla.org/en/docs/Web/API. [Accessed 13 March 16].

3. Social Compare. 2015. Web browser comparison. [ONLINE] Available at: http://socialcompare.com/en/comparison/web-browser-comparison. [Accessed 13 March 16].

4. JavaScripting. 2014. The database of JavaScript libraries. [ONLINE] Available at: https://www.javascripting.com. [Accessed 13 March 16].

5. Airpair. 2015. Javascript Framework Comparison. [ONLINE] Available at: https://www.airpair.com/js/javascript-framework-comparison. [Accessed 13 March 16].

6. Google Trends. 2016. Web Search interest: ember.js, angularjs, backbone.js - Worldwide, 2004 - present. [ONLINE] Available at: https://www.google.com/trends/explore?hl=en-US#q=ember.js,+angular.js,+backbone.js&cmpt=q. [Accessed 13 March 16].

7. Lerner, Ari, 2016. ng-book 2. 1st ed. Helsinki: FULLSTACK.io.

8. GitHub. 2016. DefinitelyTyped. [ONLINE] Available at: https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/angularjs/angular.d.ts. [Accessed 13 March 16]

9. Mozilla Developer Network. 2015. WebAPI | MDN. [ONLINE] Available at: https://developer.mozilla.org/en/docs/Web/API. [Accessed 13 March 16].

10. Inc., Yahoo!, 2011. Mobile Modes. How to Connect with Mobile Consumers, [Online]. 2, 8. Available at: http://www.rafaelfelipesantos.com.br/download/pesquisa-yahoo-mobile-modes.pdf [Accessed 13 March 2016].

11. Mozilla Developer Network. 2015. WebAPI | MDN. [ONLINE] Available at: https://developer.mozilla.org/en/docs/Web/API. [Accessed 13 March 16].

12. Lerner, Ari, 2013. ng-book - The Complete Book on AngularJS. 1st ed. Helsinki, Finland: Fullstack io

13. Igor Minar. 2012. MVC vs MVVM vs MVP. What a controversial topic that many developers can spend hours and hours debating and arguing about.. [ONLINE] Available at: https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV. [Accessed 21 March 16].

14. AngularJS. 2010. Developer Guide: Conceptual Overview. [ONLINE] Available at: https://docs.angularjs.org/guide/concepts. [Accessed 21 March 16].

15. AngularJS. 2010. Developer Guide: Scope. [ONLINE] Available at: https://docs.angularjs.org/guide/scope. [Accessed 21 March 16].

16. AngularJS. 2010. Developer Guide: Services. [ONLINE] Available at: https://docs.angularjs.org/guide/services. [Accessed 21 March 16].

17. Betts, Dominic, 2013. Dependency Injection with Unity. 1st ed. Helsinki, Finland: Microsoft patterns & practices.

18. AngularJS. 2010. API: $injector. [ONLINE] Available at: https://docs.angularjs.org/api/auto/service/$injector. [Accessed 21 March 16].

19. Martin Fowler. 2004. Inversion of Control Containers and the Dependency Injection pattern. [ONLINE] Available at: http://martinfowler.com/articles/injection.html. [Accessed 21 March 16].

20. AngularJS. 2010. API: $q. [ONLINE] Available at: https://docs.angularjs.org/api/ng/service/$q. [Accessed 21 March 16].

21. Github Angular. 2016. angular.js/timeout.js at master · angular/angular.js. [ONLINE] Available at: https://github.com/angular/angular.js/blob/master/src/ng/timeout.js. [Accessed 21 March 16].

22. AngularJS. 2010. API: $rootScope.Scope. [ONLINE] Available at: https://docs.angularjs.org/api/ng/type/$rootScope.Scope. [Accessed 21 March 16].

23. Wikipedia, the free encyclopedia. 2016. TypeScript. [ONLINE] Available at: https://en.wikipedia.org/wiki/TypeScript. [Accessed 31 March 2016].

24. AngularJS: Tutorial. 2016. 5 - XHRs & Dependency Injection. [ONLINE] Available at: https://docs.angularjs.org/tutorial/step_05. [Accessed 31 March 2016].

25. AngularJS: API. 2016. ngMock. [ONLINE] Available at: https://docs.angularjs.org/api/ngMock. [Accessed 31 March 2016].

26. Microsoft/TypeScript Wiki. 2016. Roadmap. [ONLINE] Available at: https://github.com/Microsoft/TypeScript/wiki/Roadmap. [Accessed 31 March 2016].

27. Wikipedia, the free encyclopedia. 2016. Observer pattern. [ONLINE] Available at: https://en.wikipedia.org/wiki/Observer_pattern. [Accessed 4 April 2016].

28. Tero Parviainen. 2014. The three watch depths of AngularJS. [ONLINE] Available at: http://teropa.info/blog/2014/01/26/the-three-watch-depths-of-angularjs.html. [Accessed 4 April 2016].