MIGUEL ESTEBAN RAMOS MONTILLA

SOFTWARE DEVELOPERS' PERCEPTIONS ABOUT ANGULARJS

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte

March 2016

Agradecimentos

Agradeço a minha família pelo amor e apoio oferecido. Eles são responsáveis pela minha formação como pessoa e pela inspiração para ir em busca dos meus sonhos. Mesmo na distância, suas palavras de suporte sempre me fizeram sentir como se estivessem ao meu lado. Agradeço a Stefanya, pela companhia, o carinho e apoio oferecido durante este tempo. Agradeço a todas as pessoas que fizeram com que minha estadia no Brasil fosse cheia de alegrias. É muito mais fácil fazer um curso de mestrado quando se está rodeado de pessoas tão incríveis como as que eu conheci aqui. Agradeço ao professor Marco Túlio pela orientação que me ofereceu durante o curso de mestrado. Especialmente, agradeço a paciência, o tempo investido e as dicas oferecidas para desenvolver um trabalho de alta qualidade. Agradeço aos colegas do grupo de pesquisa ASERG por terem me escutado e apoiado quando tive dificuldades. Agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) e a todos os professores que contribuíram no meu desenvolvimento acadêmico. Por fim, agradeço ao Programa PEC-PG e ao CNPq pelo apoio financeiro oferecido.

Resumo

A construção de aplicações Web atualmente exige o uso de novas tecnologias e modelos de desenvolvimento para oferecer uma melhor experiência aos usuários. Um exemplo é o modelo *single-page* que é usado para o desenvolvimento de interfaces de usuário Web. Esse modelo usa comunicação assíncrona (*AJAX*) para evitar recarregar a página inteira a cada ação do usuário que modifica o estado da aplicação. As aplicações desenvolvidas com esse modelo são chamadas de *Single-Page Applications (SPAs)*. Existem importantes desafios no desenvolvimento de *SPAs*. Dado que uma maior parte das funcionalidades das aplicações é implementada no lado do cliente, maiores e mais complexas bases de código JavaScript são necessárias. Isso requer, por exemplo, tomada de decisões sobre como estruturar o código para melhorar sua manutenibilidade. Para resolver esse tipo de problema, uma nova família de frameworks JavaScript surgiu para fornecer estrutura usando o padrão de desenvolvimento arquitetural MVC (ou uma variação dele). Desses frameworks o mais popular é AngularJS. Apesar dessa popularidade, não há um conhecimento claro sobre a forma como as suas propriedades arquiteturais e funcionalidades afetam a experiência de desenvolvimento de aplicações Web. Dado que esse conhecimento pode contribuir para a evolução das ferramentas e a adoção de melhores práticas no desenvolvimento, nesta dissertação de mestrado apresenta-se um estudo empírico sobre a experiência de desenvolvimento fornecida pelo AngularJS baseado em percepções e experiências de desenvolvedores. Para isso, usamos dois *surveys* como principais instrumentos de pesquisa. O primeiro *survey* foi utilizado para pesquisar aspectos gerais de AngularJS e o segundo para pesquisar o desempenho do framework. Recebemos 460 e 95 respostas, respectivamente. As principais contribuições deste trabalho incluem a identificação das características mais apreciadas de AngularJS, os aspectos mais problemáticos do framework, práticas comuns para lidar com problemas de desempenho, e causas gerais e técnicas desses problemas.

Palavras-chave: MVC, frameworks, AngularJS, arquitetura de software, experiência de desenvolvimento.

Abstract

The construction of modern Web applications demands the use of new technologies and development models to provide better user experiences. An example is the single-page model used for the development of Web UIs. This model uses asynchronous communication (AJAX) to avoid reloading the entire page with every user action that modifies the state of the application. Applications developed with this approach are known as Single-Page Applications (SPAs). Important challenges appear in the development of SPAs. Since a greater part of features of the applications are implemented in the client side, larger and more complex JavaScript code bases are necessary. For example, developers have to make decisions about how to structure their code to be maintainable. To solve these problems, a new family of JavaScript frameworks has emerged to structure applications using the MVC pattern (or a variation of it). From these frameworks, the most popular is AngularJS. This fact can be evidenced by comparing the number of searches in Google, the number of contributors in GitHub, and the number of questions and answers in Stack Overflow. In spite of the popularity of AngularJS, there is not a clear knowledge about how its architectural properties and features affect the development experience of Web applications. Since this knowledge can contribute to the evolution of tools and the adoption of better practices in the development of Web-based systems, in this master dissertation we report an empirical study about the development experience provided by AngularJS based on developers' perceptions and experiences. We used two surveys as the main research instrument. The first survey was used to investigate general aspects of AngularJS and the second one was used to investigate the performance of the framework. We received 460 and 95 responses, respectively. Our principal contributions include the identification of the most appreciated features of AngularJS, the most problematic aspects of the framework, common practices to deal with performance problems, and general and technical causes of these problems.

Keywords: MVC, frameworks, AngularJS, architecture, development experience.

Contents

Chapter 1

Introduction

This chapter introduces this master dissertation. Initially, we discuss the motivation of the work in Section 1.1, which includes the problem that we want to solve and the benefits that the results can bring to software development. In Section 1.2, we detail the work proposed in this dissertation. Finally, we present the outline of the dissertation in Section 1.3.

1.1 Motivation

JavaScript is a fundamental piece in the construction of Web applications. It is an interpreted language characterized by its dynamic nature and flexibility. Some of its most important properties include the use of first-class functions, which allows functions to be passed as parameters or returned as values; loose typing; dynamic objects, whose structure can be modified at run time; and expressive literal notation, which can be used to create objects and arrays by simply listing their components [Crockford, 2008]. JavaScript is an object-oriented language in which objects inherit properties directly from other objects by using prototypical inheritance. However, it also supports imperative and functional programming styles [Flanagan, 2006].

It is the *de facto* programming language interpreted by Web browsers [Crockford, 2008]. It was initially designed as a scripting language that could be embedded into Web pages to extend their functionality with small executable code. However, due to the increasing number of devices that use a Web browser to reach the Internet, today JavaScript is one of the most popular and relevant programming languages [Kienle, 2010; Nederlof et al., 2014]. Additionally, with platforms of hybrid development such as PhoneGap[1], it is possible to use JavaScript to develop mobile applications. The

[1] `http://phonegap.com/`

list of non-browser environments in which JavaScript is used is large[2]; however, the most popular use still remains in the implementation of client-side components of Web applications.

Both the use of JavaScript and the language itself have evolved along with the Web. The network that was used to retrieve and post static documents has become a network that is also used to deliver software services throughout the world, inviting new user interactions [Hall and Tiropanis, 2012; Hendrickson and Fowler, 2002]. Modern Web applications are used for multiple purposes including e-commerce, information distribution, entertainment (games), and collaborative working. However, this type of applications also demands better user interfaces (UIs) with better user experiences. This requirement is reflected in the birth of new technologies and tools (including JavaScript libraries and frameworks) to solve common problems present in the development of client-side components.

As an example, we can mention the problems faced by the developers with the Document Object Model (DOM), which is an interface that allows programs and scripts to dynamically access and update the content, structure and style of Web documents [W3C, 2005]. Some studies have reported that many client-side JavaScript bugs are related with methods of the DOM API [Ocariza et al., 2013]. The DOM API is the source of many incompatibilities due to its inconsistent implementation through different browsers. Fortunately, this problem has been addressed with DOM manipulation libraries such as jQuery[3], which provide simple abstractions to interact with the DOM and removes the need to write code to support each implementation of this API.

Another technology that has contributed to the improvement of the user experience provided by the current Web applications is AJAX (Asynchronous JavaScript And XML). This technology allows to asynchronously interchange data between the server and the client without interfering with the display and behavior of the existing page [Garrett, 2005]. This characteristic has been used to create an alternative to the traditional multi-page model for the development of Web interfaces. In this new model, there is no need to refresh the entire user interface with every request made by the client to the server. Instead, the interface is in a single page composed of individual components that can be updated or replaced individually, depending on the state of the application. However, since new application states are retrieved from or sent to the server using asynchronous communication (AJAX requests), there is no need to reload the entire page with every user action [Mesbah and Deursen, 2007b]. Applications developed with this approach are known as Single-Page Applications (SPAs).

[2] `https://en.wikipedia.org/wiki/JavaScript#Uses_Outside_Web_pages`
[3] `https://jquery.com/`

SPAs provide higher levels of interactivity, responsiveness and user satisfaction [Mesbah and Deursen, 2007a]. However, new challenges in their development also appear. Since a greater part of the features of the application are implemented in the client side, larger and more complex JavaScript code bases are necessary. Due to the flexibility of JavaScript, developers have to make design decisions about how to structure their code, where to write code to manipulate the DOM, how to maintain the consistency between the state of the application and the UI, how to manage the dependencies of the application, etc. The new UIs have to be designed from the outside to be usable and from the inside to be maintainable.

To address the aforementioned problems a new family of JavaScript frameworks has emerged to give structure to Web applications by using the famous Model-View-Controller (MVC) architecture pattern (or variations of it) [Osmani, 2012]. These frameworks are known as client-side or JavaScript MVC frameworks. The MVC pattern, which is one of the oldest for the construction of UI, has been proposed multiple times as a key part in the development of maintainable Web systems [Leff and Rayfield, 2001; Knight and Dai, 2002]. However, the difference is that JavaScript MVC frameworks work only with the components in the client-side of the application.

Some of the most recognized client-side MVC frameworks are AngularJS, Backbone.js, and Ember.js. Among them, AngularJS is the most popular one. This fact can be evidenced by comparing the number of searches made with the Google search engine (Figure 1.1), the number of contributors to the project in GitHub (Figure 1.2), and the increasing number of questions and answers in Stack Overflow (Figure 1.3).

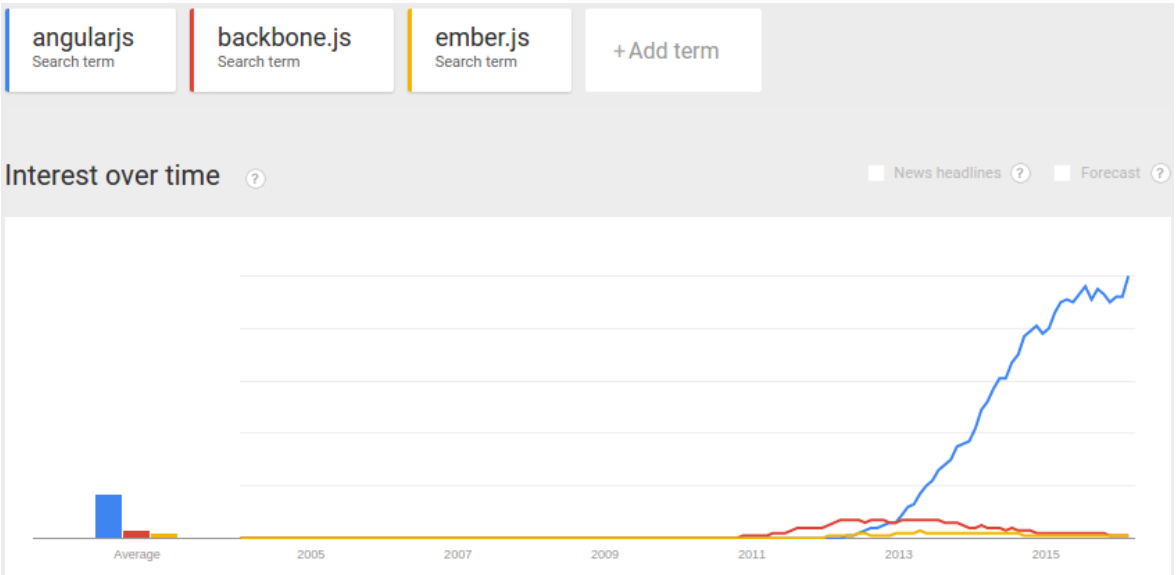


Figure 1.1. Goggle searches for different client-side MVC frameworks

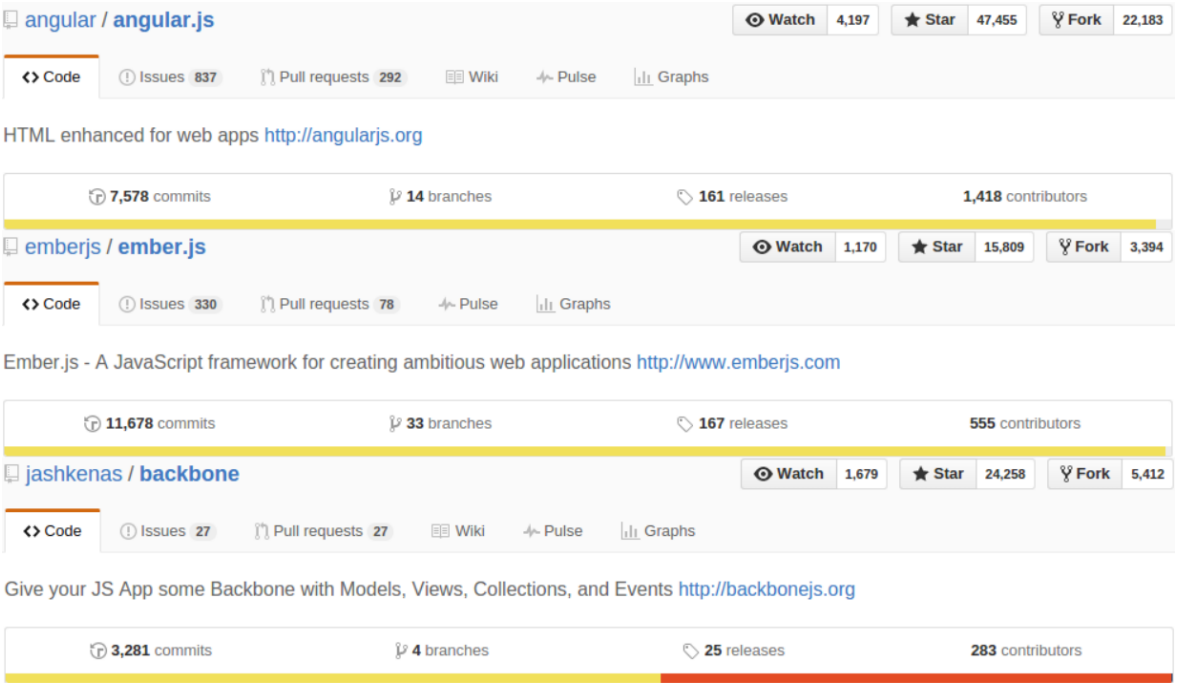


Figure 1.2. GitHub Contributors of different client-side MVC frameworks

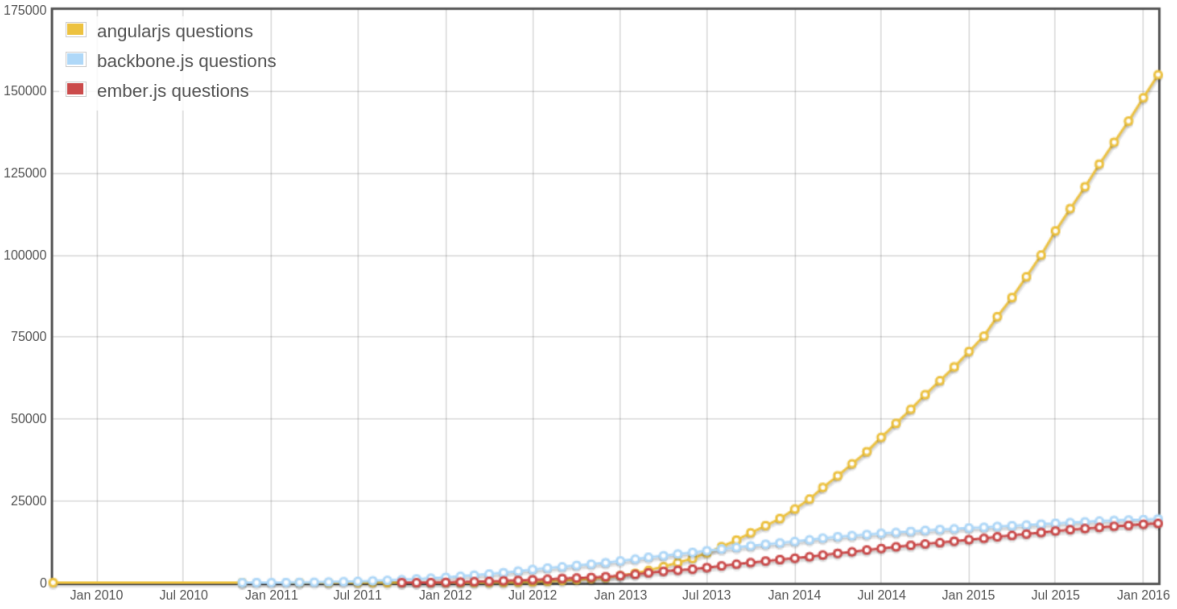


Figure 1.3. Number of SO questions for different client-side MVC frameworks

One of the key characteristics of AngularJS is its capability to give more expressiveness to the HTML language by augmenting its vocabulary to write dynamic applications. AngularJS gives developers the opportunity to create custom HTML elements and attributes to define the composition and the behavior of the user interface.

Another characteristic that distinguishes AngularJS from other MVC frameworks is that it manages the dependencies between the internal components of the application by using the dependency injection pattern. AngularJS helps in the construction of SPAs not only by providing structure, but also by providing features to accomplish common tasks such as DOM manipulation and communication with the server.

In spite of the popularity of AngularJS, there is not a clear knowledge about how its architectural properties and features affect the development experience of Web applications. More specifically, it is not clear what are the most appreciated characteristics of AngularJS, what are the principal problems faced by developers when they use it, and which aspects of the framework can be improved. Having such information can be useful for different reasons:

- Developers who do not yet use JavaScript MVC frameworks can better understand the benefits and problems related to their use by reviewing the case of AngularJS. Additionally, they can make better decisions about what framework to use avoiding a hasty selection based just on the popularity of the framework.
- Developers who already use AngularJS can learn to better use the features provided by the framework and to avoid bad practices that can lead to implementation and maintenance problems.
- Designers of client-side MVC frameworks can use this information to build more powerful frameworks by strengthening the most appreciated features, or adding support to features often implemented by developers, or improving the usability of APIs that are difficult to use, or revisiting architectural designs.

These reasons motivated us to design and execute an empirical study about the development experience provided by AngularJS.

Previous work related with the topic of this dissertation includes the revision of three different AJAX frameworks to understand their architectural properties and introduce a new architectural style [Mesbah and Deursen, 2007a]; the proposal of a tool to detect identifier inconsistencies in JavaScript MVC applications [Ocariza et al., 2015]; and a technical review about the level of risk in projects using AngularJS [Bijlsma and Ghijsen, 2014].

1.2 Proposed Work

As we have mentioned before, understanding the impact of the features and architectural properties of AngularJS in the development of Web applications is an important

contribution to the evolution of tools and the adoption of better practices in the construction of Web-based systems. To achieve this goal, in this dissertation we propose an empirical study of AngularJS based on software developers' perceptions. This approach allows the identification of trends in the software development experience of AngularJS developers, which can be used to draw conclusions about various aspects of the framework. This study uses two surveys as the main research instrument.

Two important activities were performed to build the questionnaires of the surveys. First, we carefully reviewed the documentation of the framework to understand its principal features and constructions. Additionally, we strengthened our knowledge about the framework by checking different examples and tutorials of its adoption. Second, we executed of literature review looking for assumptions, findings, opinions and problems related with AngularJS. We gathered information collected from digital libraries, blogs, and on-line communities. We further analyzed this information to find common themes that were then used as a base for the construction of the two surveys.

The first survey was used to collect AngularJS developers' perceptions about different aspects of the framework. We included questions to find out what are the most relevant characteristic of the framework and what are the most frequent problems faced by developers who use it. We also set some questions to identify common practices followed by these developers. The possible participants of this survey were selected from Stack Overflow[4] (SO), which is an active software community to post questions and answers related with software development. We proposed and implemented a procedure to identify developers with some experience in AngularJS from the community. Based on the experience of the developers (which was determined using a heuristic proposed by us) and the possibility to contact them, we selected a smaller group to promote the survey. First, we made a pilot with 30 developers, which helped us to identify and correct errors in the construction of the questionnaire. Then, the final version of this survey was sent via email to 3,060 developers and we received a total of 460 responses (15% response rate). The main findings of this part of the study include the identification of the most appreciated features of AngularJS, the AngularJS components that are the most difficult to test, and the most problematic aspects of the framework.

The second survey was used to study developers' perceptions and experiences specifically related with the performance of AngularJS. We decided to approach this theme in a separate survey due to the frequency with which we found blogs and SO questions related with it. In fact, the results of the first survey show that tackling

[4] `http://stackoverflow.com/`

all the details that can lead to performance degradation is the most complex aspect of AngularJS according to developers. Unlike the strategy used to promote the first survey, in which we sent the survey to specific developers, this survey was promoted by publishing the questionnaire link in open and well-established communities of AngularJS. Again, we ran first a pilot phase using Brazilian communities, which was used to identify and correct errors in the survey instrument. The final version of the survey was promoted in other AngularJS communities, including the ones established with the Twitter social network. We received a total of 95 responses. The main findings from this part of the study include the identification of common practices to deal with performance problems, the purpose of third-party components used to deal with performance problems, and general and technical causes of these problems.

1.3 Outline of the Dissertation

We organized the remainder of this work as follows:

- **Chapter 2** covers two central concepts related to this dissertation. In the first part we describe the Model-View-Controller pattern and its advantages. In the second part, we use a famous application in the JavaScript community as an example to explain the fundamental concepts of AngularJS.
- **Chapter 3** presents the design and execution of the first survey supporting the empirical study about perceptions and experiences with AngularJS. We describe in detail the two parts that compose this study: (1) The *Mapping Study* and (2) the *Survey design*. Furthermore, we use this chapter to discuss the results of the survey and present the possible conclusions that can be drawn from them.
- **Chapter 4** presents the design and execution of the second survey supporting the empirical study about perceptions and experiences with AngularJS performance. As in chapter 3, we describe in detail the two parts that compose this study: (1) The *Mapping Study* and (2) the *Survey design*. We also present a discussion regarding the results of the survey and the conclusion that we can draw from them.
- **Chapter 5** presents the final considerations of this dissertation, outlining principal findings and future work.

Chapter 2

Background

In this chapter we provide a general review of the central topics needed to understand the work presented in this dissertation. First, Section 2.1 presents a description of the MVC pattern since it represents the concept in which are based the client-side MVC frameworks. In Section 2.2, we discuss what is a client-side MVC framework and its relevance in the current state of the Web development. Finally, Section 2.3 explains the most important features and concepts of AngularJS by using a sample application.

2.1 The Model-View-Controller Pattern

The Model-View-Controller (MVC) pattern was introduced the first time by Trygve Reenskaug in 1978, as a solution to support the user's mental model of the relevant information related with the application domain and to enable the user to inspect and edit this information.[1] Nevertheless, it was not until 1988 that this concept was formally presented in the implementation of the *Smalltalk-80* user interface. In this implementation, the MVC pattern was used as a strategy to clearly represent *information*, *display* and *control* for the set of reusable system components used by the programming environment [Kransner and Pope, 1988].

MVC is a pattern to structure applications that encourages the separation of components representing (1) the model of the domain data (as an abstraction of the real world), (2) the way in which that data is presented to the user, and (3) how the user interacts with that model. The MVC architecture allows the implementation of the idea behind *Separated Presentation*, in which there is a clear division between the objects representing the domain (data and behavior) and the objects representing elements of

[1] This information may be found in *The pages of Trygve M. H. Reenskaug* which are public in the site `http://heim.ifi.uio.no/~trygver/index.html`

the user interface. Moreover, the presentation is able to access the domain objects, but the latter must remain without reference to the code used for the presentation [Fowler, 2001].

The three components of the MVC architecture are the *model*, the *view* and the *controller*. The model is related with the domain code, while the view and the controller are tied in to presentation aspects. A more detailed description of each component is presented next.

The Model: These components implement the software elements related with the specific domain. They contain the domain-specific data and provide the interface used to update it. The model objects should stay completely agnostic of the user interface using no reference to presentation objects (views and controllers). They may be as simple as an integer (used as a model for a counter) or as complex as an object containing different types of data and methods to implement more complex abstractions.

The View: This component presents the model's information to the user. The view has the ability to request information from the model to update the displayed data when some aspect of it changes. Several views can be attached to the same model as a way of presenting the same information in different forms to the user, as illustrated in Figure 2.1.

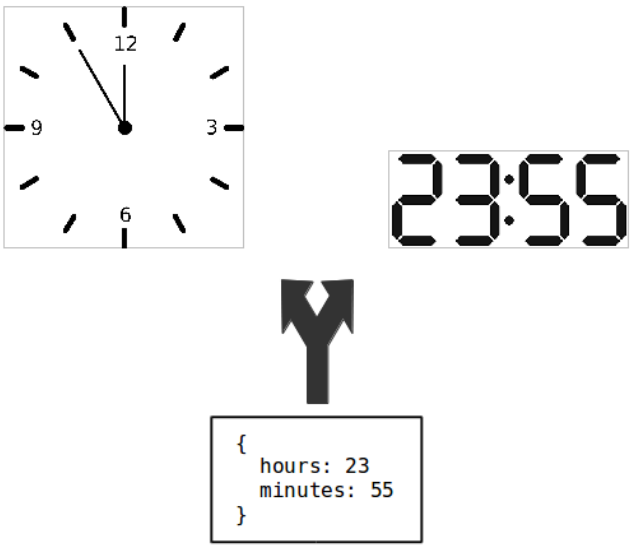


Figure 2.1. Clock model with different view displays

The Controller: The controller provides the communication interface between the input

devices and the model and view components. Its main function is to handle events of input devices representing user actions and translate them into messages requesting services from the model or the view.

The division of an interactive application into these three components brings as an advantage the separation of the core functionality from the user interface. Considering that user interfaces are prone to change over time, it is highly discouraged to tie the user interface with the functional core of the application [Buschmann et al., 1996]. Therefore, maintaining the essence of the application (the model) separated from the user interface benefits code reuse. It is also possible to use different and even simultaneous data visualization styles using the same model.

Since the controller and the view are responsible for handling the input and output from and to the user, we can say that they make up the user interface. Views and controllers may be thought of as a working unit. There is a one-to-one relation between them. Views and controllers have a single model associated, but a model can have one or more view-controller pairs associated with it.

Both the controller and the view have access to the model. The controller accesses the model to perform actions on behalf of the user and the view to request data when a change occurs. On the other hand, the model should not access any of the view-controller pairs associated with it. It has to know nothing about them. The model only provides functions to access its data and to perform application-specific processing used by the view and the controller.

Another important communication is the one going from the controller to the view. This communication is useful when there are user actions that are intended to change the display without affecting the underlying model. Examples of these actions are scrolling and highlighting of menus.

Whenever there is a change in the model all the views and controllers dependent on it have to be notified. When these components are notified they use their connection to the model to inquire for its new state. Views use the result of this query to update the displayed data while controllers use it to change the behavior of some of their methods depending on the state of the model.

In order to propagate a change in the model to all the interested components, the "classic" MVC architecture implemented by *Smalltalk-80* uses a mechanism employing a dependency list in the model. These lists register all the view-controller pairs that need to be notified when a change in the model occurs. When some change in the state of the model is triggered by any controller, a message is sent by the model to all its dependent objects informing about the change. This message can be customized with parameters to inform the nature of the change so that the listed components can

determine if they are interested or not on the specific change and respond properly. This mechanism is the only link between the model and their views and controllers.

Figure 2.2 depicts the principal components of the MVC architecture and how they interact. Typically, the user starts the interaction through one of the input devices and the controller intercepts and interprets the event using the event-handling mechanism (1). This component then sends a message to the model to change itself in the required way (2). The model executes the requested operations, possibly resulting in a change of its state. Both views and controllers registered in the dependency list of the model are notified about the change with information related to its nature (3). As is shown by the dotted arrows, this process is made using a mechanism that involves no direct access from the model to their dependent components. All the controller-view pairs associated with the changed model inquire the model about its new state (4). Views use this data to properly update their displays (5). On the other hand, controllers can use the information to modify some of their functionality depending on the model state.

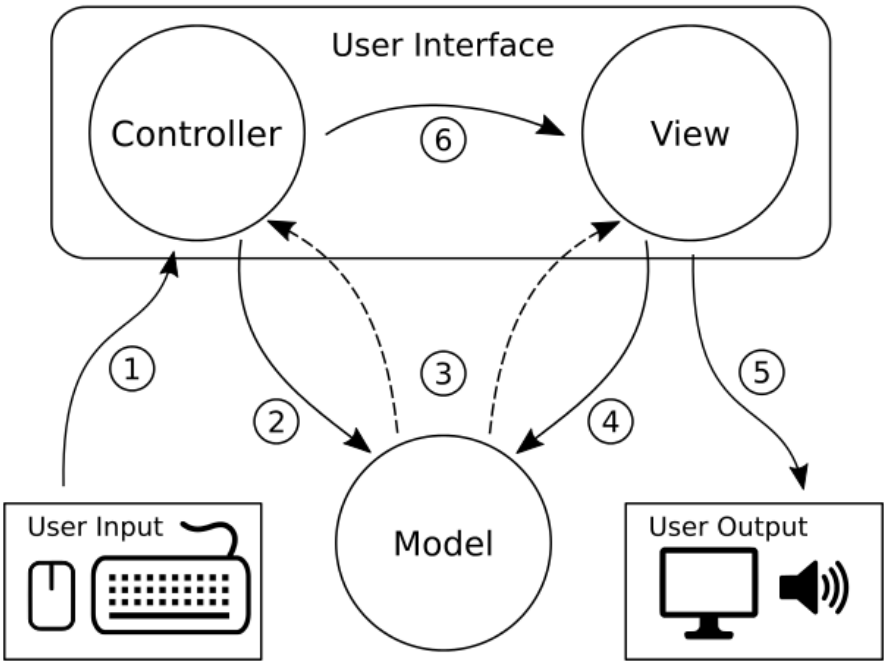


Figure 2.2. Intercation between MVC components

It is worth mentioning that sometimes this interaction cycle does not need to modify the underlying model because the user interaction will result just in a modification in the display of the application. In such cases, the controller sends a message

directly to its associated view to make the corresponding modification on the display and then returns (6).

2.2 JavaScript MVC Frameworks

Initially, when small and light websites or applications with lower complexity in the client-side were built, it was possible to implement, using plain JavaScript, some of their most recurrent requirements such as form validation, AJAX communication and DOM manipulation. However, websites needed to evolve to manage a greater quantity of data and provide better user interaction, and it was necessary to use increasingly larger amounts of code to satisfy those requirements. Therefore, in order to deal with the low-level API to traverse the DOM, the complex way to manage AJAX requests and the incompatibilities between browsers, some JavaScript libraries like jQuery were created to solve the problem. Even so, when it comes to the current large scale applications with more complex interfaces that demand not rendering the whole page for every request (single-page applications), just relying on DOM manipulation libraries leads to a big amount of nested and unmaintainable code which has no structure at all. As a result, JavaScript MVC frameworks were created to provide structure and organization to those applications [Osmani, 2012].

JavaScript MVC frameworks are intended to adapt the famous Smalltalk-80's MVC architectural pattern to the language to provide an abstraction layer that simplifies the implementation of separation of concerns. It is worth mentioning that even when those frameworks are known as MVC among the JavaScript community, the vast majority do not strictly follow the original MVC pattern. On the contrary, they may be considered as promoters of similar patterns such as MVP [Potel, 1996], MVVM [Smith, 2009], or even other variations where opinionated components are added, or there is more or less responsibility in charge of the *View* or the *Controller*. Nevertheless, all of them encourage the idea of Separated Presentation described by Fowler [2001], in which domain code and presentation code are kept in separate layers with the domain code unaware of presentation code. For simplicity, in this work all of them will be referenced as MVC frameworks.

Nowadays, there is a large number of JavaScript MVC frameworks whose characteristics make them more or less appropriate to use in a project depending on the characteristics and requirements of it. There are MVC frameworks with different levels of abstraction, flexibility, size, maturity, documentation and supporting community. Considering the multiple options of frameworks, it is important to keep in mind the

existing trade-off between the number of frameworks to be analyzed and the depth with which they are studied. In this work we will maintain that balance by limiting the analyzed frameworks to the most popular one: AngularJS.

2.3 AngularJS

In this section we will explain basic concepts about AngularJS. To achieve this goal we will use a sample application to exemplify the use of the principal components and features of AngularJS. It is important to note that we will not be reviewing complex details about the operation of the framework since they are out of the scope of this chapter.

2.3.1 TodoMVC: A Sample Application

To explain the principal concepts of AngularJS we decided to use a famous application in the JavaScript community called *TodoMVC*. This application belongs to a project founded by Osmani et al. [2011], which brings together different implementations of the same application using different frameworks. The main goal of this project is to help in the decision about what framework to use by providing comparison between multiple frameworks. Although the project has a document with all the specifications of the *TodoMVC* application, we will use a simple version of this application since our focus is to explain AngularJS core concepts. Figure 2.3 shows the interface of the application.

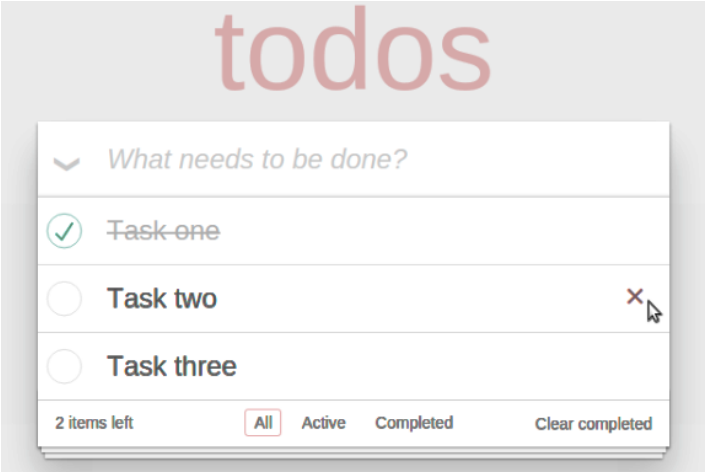


Figure 2.3. UI of the *TodoMVC* application

TodoMVC is a simple task manager. The main feature of the application is the registration of tasks that must be done in the future. These tasks are appended to a

list in which the user can mark tasks as completed. The features of this version of the application are summarized in the following list:

- *Addition of new tasks:* To create a new task ("todo"), it must be written in the input at the top of the application. When the enter key is pressed, the new task must be appended to the list of tasks and the input field must be cleared.
- *Deletion of tasks:* To remove a task from the list, the remove button (x) in the item must be clicked. Additionally, there is a button at the bottom of the application to remove all the completed tasks.
- *Completion of tasks:* Each item in the list has a check box to mark the task as completed.
- *Counting incomplete tasks:* At the bottom of the application there is a counter showing how many tasks are incomplete.
- *Persistence of state:* To avoid loss of information when the browser is closed, the state of the application must be persisted by using `localStorage`.

2.3.2 Important Features and Concepts of AngularJS

In this section we will describe how to implement the *TodoMVC* application using AngularJS and at the same time we will present some concepts behind AngularJS by looking at its different features and components.

2.3.2.1 Modules

An AngularJS application is always built as a set of modules. They can be thought of as containers for the different parts of the application. They are intended to pack reusable components of the application [AngularJS Developer Guide, 2015b]. When a module is declared it can list other modules as its dependencies. Our sample application has two modules: `storage` and `todomvc`. The `storage` module contains the code to deal with the persistence of the application (using `localStorage`) and the `todomvc` module is the main module of the application and contains the rest of components of it. Listing 2.1 shows how these two modules are declared in AngularJS.

```
1 var storage = angular.module('storage',[]);
2 var todomvc = angular.module('todomvc', ['storage']);
```

Listing 2.1. Module declaration with AngularJS

The modules are defined by using the `module` method provided by AngularJS. Two parameters are passed to this method: (1) a string containing the name of the new module and (2) an array of strings specifying the names of other modules that it depends on. This method returns the new created module. In Listing 2.1, line 1 defines the `storage` module with an empty array as second parameter because it has no dependencies. On the other hand, the `todomvc` module, which is defined in line 2, has the name of the `storage` module in its array of dependencies. When a dependency relation is established between two modules it implies that the required module needs to be loaded before the requiring module is loaded.

2.3.2.2 Services

AngularJS services are objects that encapsulate functionalities related with an specific concern. They are instantiated only once by using factory or constructor functions. The created singleton is shared by the different components that depend on it (e.g, controllers, directives, filters and other services). When a service is needed by an other component of the application, the name of the service must be specified in the other component, and AngularJS injects the service into the component for its utilization. Most of the time, AngularJS services are stateless objects with a set of methods to deal with specific aspects of the application such as server requests, manipulation of arrays, asynchronous operations, etc. The main benefit of services is the separation of concerns. AngularJS has some built-in services to deal with common concerns in the development of Web applications. Examples are `$http`, `$filter`, and `$timeout`.

In the *TodoMVC* application we use a service to encapsulate persistence concerns, as shown in Listing 2.2. The name of this service is `todoStorage` and it has two methods: (1) the method `get` to retrieve the list of tasks from the `localStorage` and (2) the method `put` to save a list of tasks into the `localStorage`. Since this service will be created using a factory function and we also want to place the service within the `storage` module, we use the `factory` method provided by the `storage` module to register the service. This method receives as parameters the name of the service and the factory function that will instantiate the service.

```
1 storage.factory('todoStorage', function () {
2
3   var STORAGE_ID = 'todos-angularjs-perf';
4   return {
5     get: function () {
6       return JSON.parse(localStorage.getItem(STORAGE_ID) || '[]');
7     },
```

```
8
9       put: function (todos) {
10        localStorage.setItem(STORAGE_ID, JSON.stringify(todos));
11      }
12    };
13  });
```

Listing 2.2. Persistence service

It is important to note that AngularJS will instantiate this service only if it is listed by other components of the application as a dependency.

2.3.2.3 Templates

Web applications use HTML to define the structure of the document presented to the users. The HTML documents are parsed by the browsers to generate the DOM, which models the final document presented to the user. The DOM can be dynamically accessed by programs and scripts to update the content, structure, and style of this document [W3C, 2005]. AngularJS uses this fact to support DOM-based templates. These templates are written in HTML and contain proprietary elements and attributes to render the dynamic interface that a user sees in the browser [AngularJS Developer Guide, 2015d]. Listing 2.3 shows some parts of the template of the sample application for illustration purposes. It includes the definition of the document (`html` element in line 1) and the markup for the main input of the application, which is represented by the `form` element in line 5.

```
1  <html lang="en" ng-app="todomvc">
2  ...
3    <header id="header">
4      <h1>todos</h1>
5      <form ng-submit="addTodo()">
6        <input placeholder="What needs to be done?" ng-model="
       newTodo"/>
7      </form>
8    </header>
9  ...
10 </html>
```

Listing 2.3. Template sample of the *TodoMVC* application

In this template we can find two important constructions that are specific to AngularJS: (1) directives and (2) expressions. The directives can be easily distinguished because all of them appear as HTML attributes with the "`ng`" prefix. Expressions, on the other hand, appear as the values of the aforementioned directives. Like any

other HTML document, AngularJS templates are parsed by the browser generating the corresponding DOM. This DOM is traversed by AngularJS to find the directives that work as instructions to modify the original DOM and add event handlers depending on the values of the expressions. This process is referenced as compilation in the AngularJS documentation. As an example, we can mention the `ng-app` attribute (line 1), which represents an AngualrJS directive that specifies the root node of the application. Only elements nested into this node will be considered by AngularJS as being part of the application. This directive also allows the programmer to declare the name of the root module, which is called `todomvc` in Listing 2.3.

2.3.2.4 Expressions

AngularJS expressions are code snippets placed templates, which can be found inside double curly brackets (`{{expression}}`) or as the values of some directive attributes. Literals (e.g, arrays (`[]`), objects (`{}`)), operators and variables are examples of possible elements that can be used in an AngularJS expression. It is also possible to make function calls and use the ternary operator (`?`). Expressions are very similar to JavaScript expressions, but they can not contain high level statements such as function declarations or control flow statements. As an example, we show the use of an expression in Listing 2.4 and the result after its compilation in Listing 2.5.

```
1 <span>{{"Hello" + " " + "world!"}}</span>
```

Listing 2.4. Expression used in a template

```
1 <span>Hello world!</span>
```

Listing 2.5. Result after template compilation

AngularJS expressions are evaluated against a context represented by an object called scope. In other words, variables and functions used in expressions must be defined in the scope object. During the compilation of the template, when the `ng-app` directive is found, AngularJS creates an object to represent the main scope of the application. This object is referenced as the `$rootScope`. However, new scopes that inherit from `$rootScope` can be created by other directives. Expressions are always evaluated against a local scope created by a specific directive. For this reason, expressions placed in different parts of the template may use different scopes for their evaluation.

As we have seen, the value of an expression depends on both the values of the properties referenced by the expression and the returned values of the methods called by the expression. When these values change during the execution of the application,

AngularJS takes care of maintaining updated the whole value of the expression in the view.

2.3.2.5 Controllers

AngularJS controllers are used to manage the state of the application related with the presentation (the *View Model* in the MVVM pattern). They are in charge of initializing the state of the application and providing the necessary interface to update it. For this purpose, most of the time controllers are used with the `ngController` directive. When this directive is used in the template, it receives as value the name of any controller. The new scope created by this directive is passed as a parameter to the specified controller. The function of the controller is to populate the scope object with properties and methods that will be available in the template for the evaluation of expressions.

To illustrate this situation we can think of the variables needed to represent the state of our sample application. We need an array to store the list of added tasks, a variable to represent a new task (the text written in the main input), and a variable to keep the count of incomplete tasks. Listing 2.6 shows how these variables are defined in the controller of the *TodoMVC* application to initialize the view model. As we can see in line 1, the `controller` method is used to register a controller in the `todomvc` module. This method receives as parameters the name of the controller (`TodoCtrl`) and the function that defines the controller.

The controller function is injected with three parameters: (1) the scope created by the directive that will reference the controller (`$scope`), a proprietary AngularJS service with filter features (`$filter`), and (3) the `todoStorage` service. In Listing 2.6, lines 2-4 show how the `$scope` object is populated with the necessary properties using the injected services. The `todoStorage` service is used to retrieve the registered tasks from memory and the `$filter` service is used to filter these tasks to obtain the number of incomplete tasks.

```
1 todomvc.controller('TodoCtrl', function TodoCtrl($scope, $filter,
    todoStorage) {
2   $scope.todos = todoStorage.get();
3   $scope.newTodo = '';
4   $scope.remainingCount = $filter('filter')(todos, {completed:
     false}).length;
5   ...
6 }
```

Listing 2.6. Initialization of the state of the *TodoMVC* application

To provide an interface that allows the manipulation of the application state, we also need to add some methods to the `$scope` variable, which can be called using expressions in the template. Listing 2.7 shows the addition of two methods to the `$scope` object. The first method (lines 4-16) is in charge of adding the new task referenced by the property `$scope.newTodo` to the list of tasks and saving the updated list in memory (using the `todoStorage` service). In addition, it has to update the count of incomplete tasks (`$scope.remainingCount`) and clean the variable that references a new task (which will also clean the input of the application). The second method (lines 19-23) is in charge of removing a task from the list. It has to update the count of incomplete tasks, remove the task from the list, and save the updated list in memory. It is important to note that our sample application has another method to mark a task as complete, which was not showed in Listing 2.7 for simplicity.

```
1  todomvc.controller('TodoCtrl', function TodoCtrl($scope, $filter,
       todoStorage) {
2    ...
3    //Add a new task
4    $scope.addTodo = function () {
5
6      var newTodo = $scope.newTodo.trim();
7
8      todos.push({
9        title: newTodo,
10       completed: false
11     });
12     todoStorage.put(todos);
13
14     $scope.remainingCount++;
15     $scope.newTodo = '';
16   };
17
18   //Remove tasks
19   $scope.removeTodo = function (todo) {
20     $scope.remainingCount -= todo.completed ? 0 : 1;
21     todos.splice(todos.indexOf(todo), 1);
22     todoStorage.put(todos);
23   };
24   ...
25 }
```

Listing 2.7. Methods to manipulate the state of the application

It is also important to note that controllers can have more variables or functions,

but only the ones attached to the `$scope` can be referenced by expressions in the template. Now we can use expressions and directives to define both how the UI reflects the state of the application and how the state of the application can be modified by the UI.

2.3.2.6 Directives

Directives are specific HTML markers used in the template to define the UI behavior. When directives are executed by AngularJS they have the ability to modify the structure of the DOM and register event handlers. They can be found in templates taking the form of HTML constructs such as element tags, attributes, CSS classes, and even comments. During the compilation process, AngularJS traverses the DOM generated for the template by the browser, and searches for all the HTML markers that represent a directive. Then, these directives are executed in order of priority generating the final DOM presented to the user.

When a directive is used in an element it can define the scope of the expressions wrapped by this element. There are three different ways in which a directive can define scopes: (1) they can use the same scope of the parent element, (2) they can create a scope that prototypically inherits from the scope of the parent element, or (3) they can create a completely isolated scope.

By inspecting the template of our sample application we can better explain the aforementioned concepts. Listing 2.8 shows a fragment of the template that describes two important parts of the UI: (1) the input to insert a new task and (2) the list of tasks.

The first directive that we can find is the `ngController` directive (line 1). This directive creates a new scope that will be used to evaluate the expressions wrapped by the `section` element. The value passed to this directive is the name of the controller that will be used to populate the aforementioned scope. We have already described this process in Section 2.3.2.5.

The second directive that we find in the template is the `ngSubmit` directive used in the `form` element (line 4). The value passed to this directive is an expression that indicates the action that must be performed when the form is submitted. In this case, the function `addTodo` must be called. Since this expression is wrapped by the `section` element, the scope used to evaluate it is the one created by the `ngController` directive. Therefore, the method `addTodo` is the one defined in the `TodoCtrl` controller. To achieve this behavior, during the template compilation this directive is interpreted by AngularJS as an instruction to add an event handler to the submit event of the

`form` element. This handler executes the action represented by the expression passed to the directive.

```
1  <section ng-controller="TodoCtrl">
2    <!--Main input-->
3    <h1>todos</h1>
4    <form ng-submit="addTodo()">
5      <input placeholder="What needs to be done?" ng-model="newTodo"
       >
6    </form>
7    <!--List of tasks-->
8    <section ng-show="todos.length">
9      <ul>
10       <li ng-repeat="todo in todos" ng-class="{completed: todo.
       completed}">
11         <div>
12           <input type="checkbox" ng-model="todo.completed"
       ng-change="todoCompleted(todo)">
13           <label>{{todo.title}}</label>
14           <button class="destroy" ng-click="removeTodo(todo)"></
       button>
15         </div>
16       </li>
17     </ul>
18   </section>
19   ...
20 </secton>
```

Listing 2.8. Template of the *TodoMVC* application

The last directive that we can find in the template representing the main input of the application is the `ngModel` directive (line 5). This directive is used to set a two-way data binding between the `input` element and the variable pointed in the expression. Again, this variable is evaluated against the scope object created by the `ngController` directive and populated by the `TodoCrtl` controller. This type of binding guarantees that the value of the `input` element is always in sync with the one referenced by the `newTodo` variable. As is usual, this behavior is set during the template compilation by setting an event handler in the `input` element and constantly reviewing the value of the `newTodo` variable to maintain them in sync if a change is detected in any of them.

Since the application should only display the `section` element containing the list of tasks if this list has at least one element, the `ngShow` directive is used to indicate that the element should only be displayed if the property length of the array maintaining the tasks is greater than zero (e.g., the expression evaluates to true). Nested in the

`section` element we can find an unordered list (`ul` element) and then the template representing each task in the list (`li` element). In this element (line 10), we used the `ngRepeat` directive, which has the power of replicating the associated element by the number of items of the specified array. Additionally, this directive creates a new object scope for each item of the collection, which prototypically inherits from the parent scope.

As we can see, the value of the `ngRepeat` directive has the form "*item in collection*". Each scope created is attached with a new property whose name is the same used in the *item* position to individually reference each item of the collection. This behavior mimics the one found in the *for each* statement of mainstream programming languages. In the case of the sample application, we use the variable `todo` to reference each task in the list of tasks (`todos`).

In the template representing each task we have four additional directives. The `ngClass` directive is used to add the "*complete*" CSS class depending on the value of the `completed` property of each task. The `ngModel` directive is used to sync the state of the check box with the value of the aforementioned property. Moreover, the `ngChange` and `ngClick` directives are used to toggle the `completed` property of the specific task when the state of the check box is changed (function `todoCompleted`) and to delete the specific task when the destroy button is clicked (function `removeTodo`), respectively. Finally, in line 13, we have the expression that references the `title` property of each task in the `label` element. In the final DOM, this expression is replaced with the title of each task.

The directives examined in this section refer to a wide set of built-in directives useful for building any application. However, there is also a way to create custom directives. For example, it is possible to create HTML tags that will be replaced with a more complex structure or custom attributes to modify the DOM or add event handlers in a specific way. When several directives are created they can become a Domain Specific Language (DSL) to describe the UI of the application.

2.3.2.7 Digest Cycle

In Section 2.3.2.6, we mentioned that AngularJS has the ability to constantly maintain in sync the state of the application with the view presented to the final user. As an example, we can mention the action to remove a task from the list. When the destroy button is clicked, the `removeTodo` method is executed and the task is actually removed from the array representing the tasks. Moreover, without having to write additional code, the DOM is automatically updated to reflect this change. AngularJS provides

this synchronization by constantly observing the variables in the scope referenced by expressions in the template. When a change is detected in one of these variables, AngularJS adequately modifies the DOM to reflect the change.

Scope objects have a method called `$watch` to register expressions or functions whose returned values must be constantly observed. Listing 2.9 shows two valid uses of this method. The first parameter is a string containing an expression to be evaluated or a function to be executed, and the second parameter is a function (handler) that will be executed if a change is detected in the value returned by the parameter being watched. To simplify the analysis of this process we will refer to this action as the addition of a *watch*. It is important to note that if the parameter being watched is an expression it is evaluated against the scope adding the watch.

```
1 //Watching an expression
2 $scope.$watch("expression", function(){})
3
4 //Watching a function
5 $scope.$watch(function( $scope ){}, function(){})
```

Listing 2.9. Use of the `$watch` method

There is a "list" that maintains a copy of the current value of the registered watches. When an action performed by the application has the possibility of changing the value of these watches, AngularJS executes all the expressions and functions being watched and compares the returned values with the old ones. If a change is detected, the registered handler is executed. Since the execution of the handlers can also change the scope properties, this process is repeated until no more changes are detected. This process is known as the *digest cycle*.

Another two important methods of the scope objects are `$digest` and `$apply`. `$scope.$digest` is used to run a smaller digest cycle that verifies changes only in the current scope and its children. On the other hand, `$scope.$apply` is used to verify changes in all the scopes of the application. In other words, `$scope.$apply` internally calls `$rootScope.$digest`. Since all the scopes of the application are children of the root scope, all the registered watches are processed.

During the template compilation, AngularJS registers a watch for each property referenced by the expressions in the template. In this case, the registered handlers update the necessary parts of the DOM to reflect the changes detected in the watches. Additionally, the directives that can change the model of the application such as `ngClick` or `ngSubmit`, make a call to the `$scope.$apply` method to run the entire digest cycle.

Now, we can review in detail the process executed to remove a task. When the click event of the destroy button is triggered, the method `removeTodo` is used to handle

the event. Additionally, AngularJS triggers the digest cycle by using `$scope.$apply`. Since the collection representing the list of tasks is being watched because it appears in the expression of the `ngRepeat` directive, during the digest cycle the change in the array of tasks is detected (i.e., the remotion of the task). The handler attached to this watch deletes the DOM element that represents this task. Therefore, the synchronization between the model and the view is preserved.

2.4 Final Remarks

This chapter presented essential concepts for understanding this master dissertation. Initially, we discussed the main concepts of the Model-View-Controller (MVC) architectural pattern, including what are its principal components and how they interact with each other (Section 2.1). Next, we presented an overview about JavaScript MVC frameworks. We described what they are and why they are important in the development of modern Web applications (Section 2.2). Finally, we presented a description of the core features and concepts of AngularJS. To exemplify these features and concepts we used the implementation of a popular JavaScript application called *TodoMVC* (Section 2.3).

Chapter 3

AngularJS Survey

This chapter presents an empirical study about the relationship between the design decisions present in the architecture of the framework AngularJS and the advantages and disadvantages that they bring to the development of Web applications (primarily, Single Page applications). Section 3.1 describes the two phases that compose this study and how they were designed and executed. We discuss the results of the study in Section 3.2 and present possible threads to their validity in Section 3.3. Finally, as a conclusion to this chapter, we present some final remarks in Section 3.4.

3.1 Study Design

This study is composed by two phases. In the first phase, we performed a *Mapping Study* to collect and analyze information about work practices and perceptions of the developers with respect to the development of Web applications using AngularJS. This information was used to build a descriptive survey that was administered in a sample of AngularJS developers to better understand how they interact with the framework and what are the main benefits and problems that they experience during the development process. In this section we take a closer look at the design and execution of these two phases of the study.

3.1.1 Mapping Study

This part of the study was intended to gather as much information as possible about the use of AngularJS to develop Web applications. We were interested on identifying how the developers' work is affected by AngularJS features and design decisions. We focused on information about the design of the framework, its current state of develop-

ment, prevalent programming practices, developers perception, common problems with its adoption, and the effects that it has in the different phases of Web applications development. All this information was then used to build a survey, which represents the second part of the study.

We started by searching for previous studies made about AngularJS in recognized digital libraries of Computer Science. However, we only found one study, which is a tool to detect inconsistencies in JavaScript MVC applications [Ocariza et al., 2015]. Since this is a field of study that has been little explored, we decided to use a mapping study as a strategy to gather the desired information. This type of study is less stringent than a systematic review and therefore is very useful for fields of research with a broader scope [Wohlin et al., 2012]. A mapping study allows us to use a more flexible criteria to include or exclude sources of information to obtain the desired overview about AngularJS.

Since AngularJS is one of the most popular MVC frameworks, it is reasonable to assume that many information can be found in blogs, forums and sites of questions and answers (for developers) on the Internet. Therefore, we decided to use the Google search engine to perform the research. Firstly, we focused on finding perceptions and opinions about the benefits and disadvantages of AngularJS. Hence, we used search queries such as "*the best features of AngularJS*" or "*the bad parts of AngularJS*". Since all the information was important to get a picture about the current use of AngularJS, we carefully reviewed all the sites listed in the first positions of the results.

We also used another two strategies to reach more sources of information. The first one was to recursively access the references in the literature already reviewed, which is known as "snowballing" [Skoglund and Runeson, 2009]. The second one was to build additional search queries based on frequently mentioned topics in which we wanted to obtain a broader knowledge. As an example we can mention the search query "*transclusion directives*", which was built due to the frequency with which this topic appeared as an AngularJS concept difficult to master. In fact, there was a topic that was mentioned in so many sites, that we decided to study it separately. This study is about performance in AngularJS and it is presented in Chapter 4. It is also worth mentioning that to control the quality of the mapping study, we only maintained well-written posts and favored those in which the background of the author included experience in software development. This criteria was only applied to blog posts since we did not want to exclude Q&A posts in which common difficulties are reported by less experienced developers.

To simplify the analysis of the collected information, during the review of each source we highlighted the most relevant quotations using a Google Chrome extension.

Additionally, we created a document in which we logged these quotations. This document also contains important data about the source of information such as the access URL, the date in which it was accessed, the name of the author, and the professional information of the author (when this information was available). An identifier was assigned to each source of information and each quotation registered in the created document uses these identifiers to reference the source from which it was taken. A copy of this document is presented in Appendix A.1

The data registered in the created document was then further analyzed and classified by identifying topic trends. The result of this analysis was used as the basis for the survey design.

3.1.2 Survey Design

The goal of this survey is to generate knowledge about the effect of AngularJS in the development of Web applications by collecting experiences and opinions from developers who have used the framework. The questions in the survey are based on the information gathered in the mapping study. The questionnaire was built with the Survey Monkey[1] tool and contains 25 questions that can be answered in approximately 15 minutes. These questions are distributed as follows: three open-ended questions, nine rating questions, four Likert-type questions, and nine multiple choice questions (three of them with possible multiple answers). It is important to note that we used branching, which is a technique to redirect the participants to a specific set of questions depending on the response given to a previous question. At the end of the questionnaire we also included a space to collect relevant comments about the research and two spaces to ask the name and email of participants interested in receiving the results. Only the background questions were mandatory. To avoid forcing responses, for the rest of the questions we explicitly asked developers to skip the question in which they did not feel confident to provide an answer.

All the scales used in the survey use an even number of points to force participants to make a choice. This is a practice that have been followed in similar studies [Gousios et al., 2015]. For these questions we only labeled the options in the extreme of the scales to give the respondents an idea of the meaning of their responses. In some of the multiple choice questions where we considered that the respondent could provide a response not included in the predefined set of answers, we included and "Other" option to elicit the opinion of the developers.

[1] `http://www.surveymonkey.com/`

The survey is split into seven sections that reflect the emerging themes in the mapping study. The first section contains questions to assemble data about the background of the participants, including demographic information and experience with the technologies studied. The second section is focused on contrasting the value of the key characteristics of AngularJS. The questions in the third section are intended to evaluate perceptions and practices related with alleged problems in AngularJS templates. In the fourth section we have questions related with debugging and testing. The fifth section contains questions about diverse development practices, which are useful to complement the information extracted in other sections. The results presented in Section 3.2 reference theses questions as complementary questions. In the sixth section we evaluate complex concepts and features of AngularJS. Finally, the last section consists of two questions regarding the version 2.0 of AngularJS.

To find suitable participants for the survey, we used the Stack Exchange API[2] to look for AngularJS developers in the Stack Overflow community. First, we made a query to get all existing tags with the substring "angularjs". Then, we fetched all the questions (and their answers) in Stack Overflow that had been tagged with the tags recovered in the previous step. As a result, we got 120,696 questions, 153,689 answers, and their associated information. From this information we extracted data about three types of users: (1) users who own a question, (2) users who own an answer, or (3) users who have made an edition on a question or an answer. We got a total of 74,005 users that have had a relation with an AngularJS question or answer in Stack Overflow.

To find a group of users that we could contact for the survey, we designed an algorithm to look for their GitHub profiles. For this purpose, we used the search method of the GitHub API[3] to match the Stack Overflow users with the GitHub users. For each user, the search was made by utilizing his/her Stack Overflow nickname as a substring that should be present in either the name or the nickname of the GitHub user. We only preserved the users whose search resulted in a single match. From this process we obtained a total of 20,695 matched users (28%). However, only 8,084 of them (10.9%) had public contact information. Additionally, we used a filter to preserve users who met the following three conditions: (1) the user has a valid email, (2) the user has a Stack Overflow nickname that matches exactly the name or nickname of the associated profile in GitHub, and (3) the user has at least one relation with the collected questions that is not an edition. This filter resulted in 7,157 potential participants for the survey.

The final list of users was ranked by considering the number of AngularJS ques-

[2] `http://api.stackexchange.com/`

[3] `https://developer.github.com/v3/`

tions and answers that they have published in Stack Overflow. Each user was assigned with a score based on the following formula:

$$S = Q + 3A$$

The score S is determined by the sum of the number of questions Q and three times the number of answers A. We have given an additional weight to the number of answers since it is probable that users who have provided answers have a greater experience than users who are looking for them.

The survey was promoted by sending the questionnaire link by email. To identify possible improvements that could be made in the first version of the survey, we ran a pilot with 30 users selected from the middle of the rank whose scores were between 9 and 24. The feedback received from the pilot helped us to identify and correct typographical errors and ambiguities present in some questions. However, the most important change was in the structure of some questions. In the initial version of the survey we used ranking questions to evaluate some characteristics of AngularJS, but they were changed to rating questions because some participants complained about the time to take the survey and because we observed that most of them interrupted the survey when they faced these questions. The responses obtained during the pilot phase were discarded.

The final version of the survey was sent first to a group of 60 users to evaluate if new upgrades would be necessary. This time we received complete responses (the participants went through the entire questionnaire) and observed an enhanced response rate. Therefore, we decided to extend the invitation to the rest of the users by emailing groups of 700 users selected from the top of the ranking. A new group was emailed every day. At one point, we decided to stop because the score of the last emailed users was too low and we started to receive replies from people saying that their experience with the framework was very little. In total, we sent the survey to 3,060 users whose score was between 3 and 831. From these invitations we received 460 complete responses reaching a response rate of 15%. The survey was open during approximately one month (from early November to early December 2015).

Appendix B.1 includes a complete copy of the survey.

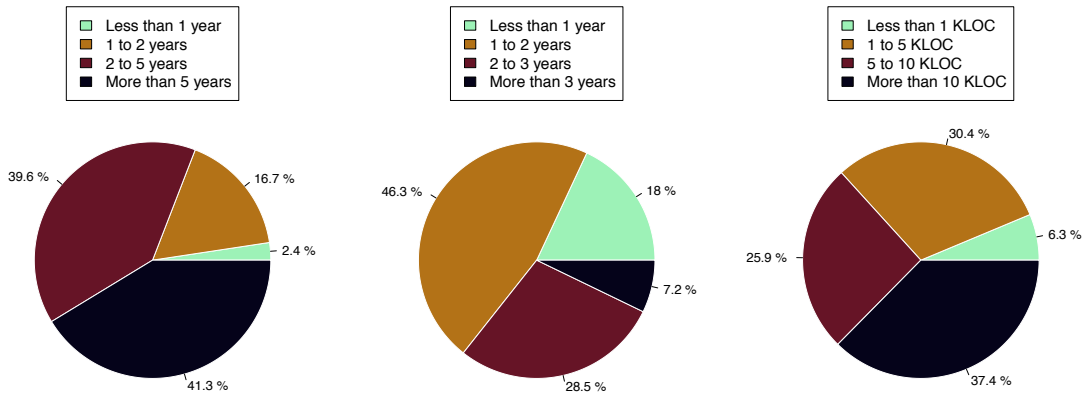
3.2 Results

In this section, we present the results of the survey and a brief discussion about each one of them. The results are examined in different subsections that reflect the thematic

structure of the questionnaire.

3.2.1 Background

The first four questions of the questionnaire give us information about the background of the respondents to contextualize the results. The distributions presented in Figure 3.1 shows how long the participants have interacted with the technologies involved in this research, and the largest application that they have developed with them. With respect to the experience in JavaScript we can say from Figure 3.1(a) that 97.6% of the respondents declared to have at least one year of experience with the language and 80.9% of them declared to have at least two years. It is also notable that more than 40% of them declared to have been working more than five years with the language. On the other hand, according to the data shown in Figure 3.1(b), there is a greater number of respondents with little experience in AngularJS. We can see that 18% of them have less than one year using AngularJS. However, 74.8% of the developers reported to have one to three years of experience with the framework. Only 7.2% of them reported having more than three years of experience. These results are reasonable since AngularJS is a relatively new technology.



(a) JavaScript experience (b) AngularJS experience (c) Largest application

Figure 3.1. Respondents background

Figure 3.1(c) shows how the responses for our question about the largest application written with AngularJS are distributed. It is possible to note that 37.4% of the developers reported that their largest application has more than 10 KLOC. If we look for the developers who have written any application with more than 5 KLOC using AngularJS, we can see that 63.3% of them have already done that. Finally, we can see

that 6.3% of the developers who took the survey have not yet developed an application with more than 1 KLOC.

We were also interested in finding out what release of AngularJS was being used by the participants for development, since depending on that they could be aware or not of some features added in the latest releases. Figure 3.2 shows this information. We can observe that most of the developers (60%) taking the survey was using the version 1.4.x for their current development, which was the latest release when this survey was taking place. The other releases that were used most frequently are the 1.3.x with 19.8% of the developers using it and the 1.2.x with 10%. Only 2.6% of the respondents declared to be using the version 1.5.x (which was in its beta version) for their current development. There were also 28 developers that answered with the option "other" to this question. From them, 20 reported that they were no longer using AngularJS; three, that they were using the version 2.0; three, that they use more than one version; and the last two, that they were not sure about the version that they were using.

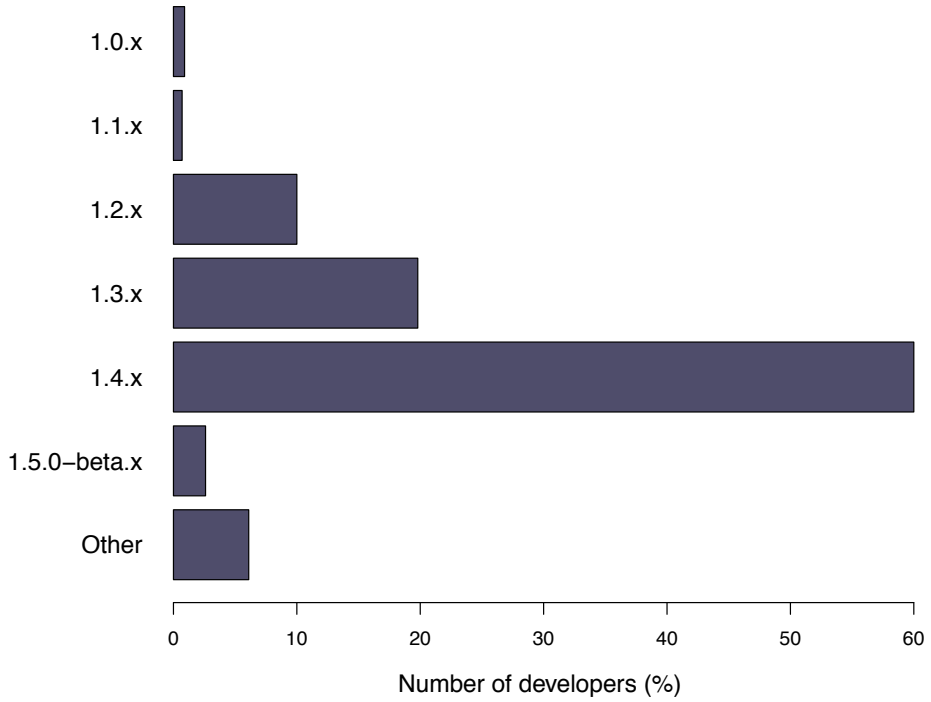


Figure 3.2. Current release used for development

3.2.2 Key Characteristics of AngularJS

Since AngularJS is currently the most popular framework, we were interested in identifying the most valuable characteristics of the framework from the developers point of view. For this purpose, we set a question (questions 6 to 15 in Appendix B.1) in which we asked the participants to rate a set of characteristics of the framework according to the value that they bring to them using a scale from *one* to *four* where *one* represents a characteristic with no value at all and *four* represents a very valuable characteristic. To construct the question we chose 10 characteristics that were frequently mentioned as important advantages of the framework in the sources considered by our mapping study. We focused mainly in technical aspects. Below we present a brief description for each one of this characteristics and how they can be considered advantageous.

1. *Pre-defined components for code organization:* This characteristic is about code structure. AngularJS has different abstractions to confine code depending on its role in the application. According to some blogs, these abstractions keep the code where it belongs and foster separation of concerns.

2. *Dependency Injection:* This is a design pattern implemented by AngularJS to handle the dependencies between the components that make up the application. The goal is to make testing and modularization easier in development.

3. *Application models with POJOs:* Due to the change detection mechanism (digest loop) implemented by AngularJS, the models can be implemented using Plain Old JavaScript Objects (POJOs) since there is no need to extend proprietary classes to provide the objects with accessors. This helps, for example, to deal directly with server information sent in JSON format.

4. *Templates directly in HTML:* AngularJS uses DOM-based templates instead of string-based templates. This type of templates allows the use of HTML syntax to write templates and maintains a reference to the created DOM elements, simplifying operations like data binding, event mapping, and updating of large components of the interface.

5. *Support to custom components:* In AngularJS it is possible to create custom directives that describe the behavior and format of single components of the UI. Together, these directives provides a DSL that uses HTML syntax to describe reusable components of the UI.

6. *Ease of writing tests:* One of the most promoted advantages of AngularJS is that it is easy to write tests due to some characteristics of the framework. Among these characteristics are the use of dependency injection, the different abstractions for separation of concerns, and the `ngMock` module that allows the mocking of operations related with timing, logging, animations, HTTP requests, etc.

7. *Two-way data binding:* This advantage refers to the ability of keeping synchronized the data between the view and the model components without the need of writing any complex code.

8. *HTML to declare UI behavior:* AngularJS allows developers to declare the behavior of the UI in the HTML document. It is possible to use HTML syntax to specify, for example, the functions called when some interactions are performed and the conditions in which an element has to appear, disappear or change its appearance.

9. *A solution to manage many common problems:* AngularJS is an opinionated framework [AngularJS Developer Guide, 2010]. This characteristic reduces or simplifies the decisions that have to be made about several concerns in the development of single-page applications. In the framework it is possible to find APIs to solve problems related with templates, code organization, DOM manipulation, data binding, filtering, RESTful communication, testing, animation, routing, among others.

10. *Supported by Google:* This is a non technical advantage. Nevertheless, some developers highlight this fact as an important advantage since it guarantees the stability, evolution and constant maintenance of the project.

Figure 3.3 shows the result of the ratings for each one of the characteristics mentioned above. For each item is presented the percentage of ratings located in the high part of the scale and in the low part. We can see the number of answers received for each item at the right part of the chart. These numbers reveal that none of the considered features was rated by 100% of the developers. The features with more participation (*custom components* and *dependency injection*) received 452 answers (98.3%) and the one with less participation (*easy of tests*) received 428 answers (93%).

Another observation that we can make from the results in Figure 3.3 is that all of the considered features were highly rated (options *three* or *four*) by a significant number of developers. The characteristic with the lowest percentage of developers in

the high part of the scale is the one regarding the Google support with 54.8% of the participants. This is also the characteristic with the greater number of developers (19.8%) rating it with option *one*.

Looking at the technical characteristics, we can see that *ease of testing* and *modeling with POJOs* are the ones with the greater number of answers in the low part of the scale (35.1% and 29.4%, respectively). It is specially unexpected that more than a third of the participants rated the *ease of testing* as a characteristic with limited or no value, when this is promoted as one of the major advantages of the framework by the AngularJS team.

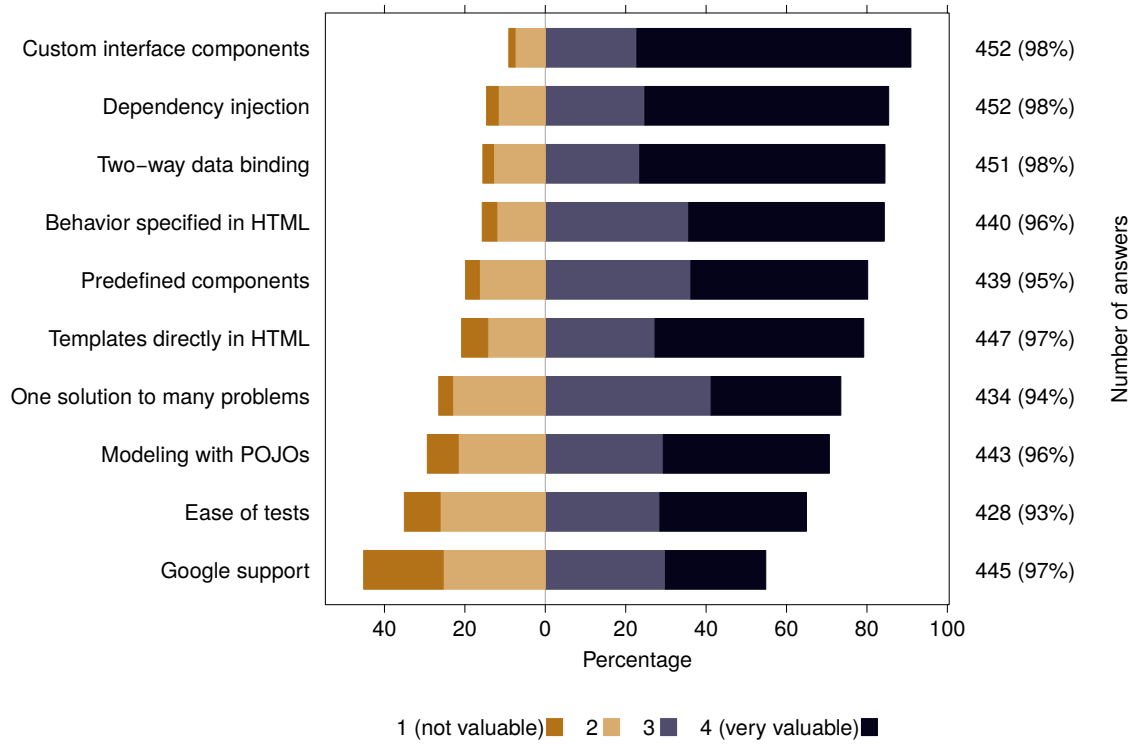


Figure 3.3. Key features and characteristics of AngularJS

On the other hand, the first three features (from top to bottom) are outstanding due to the great number of developers rating their value with the highest option in the scale (*four*) and the very small number of developers rating their value with the options in the low part of the scale (*one* or *two*). The ability of building customized UI components by using custom directives was rated as a very valuable feature by 68.1% of the participants and only seven of them (1.5%) rated as a feature without value. Furthermore, 22.8% of the participants qualified the value of this characteristic with the option *three*. This is a clear indication of the value that AngularJS users give to

the ability of building their own HTML elements and attributes to declare customized and reusable UI components.

The way in which AngularJS handles the internal dependencies of the application is also a valuable feature for the participants of this survey. From them, 85.4% rated this feature with the highest levels (options *three* or *four*) and only 2.9% with the option *one*. This is an interesting result because even when ease of testing is not one of the most valuable characteristics of AngularJS according to the participants, dependency injection, which is an enabling feature in this process, is one of them.

Finally, from Figure 3.3 is clear that another highly appreciated feature of AngularJS is two-way data binding. This feature was rated by 23.5% of the respondents with option *three* and by 61% with option *four*. Only 12 respondents (2.7%) rated this feature as a not valuable one. This result is probably due to the fact that the synchronization between the model and the view is a very common operation in SPAs. It can be simply done with AngularJS without writing boilerplate code.

3.2.3 Code in HTML templates

Most JavaScript MVC frameworks have some type of template system that is used to reflect the state of the application (maintained in JavaScript objects) into the DOM-based user interface. These template systems have different characteristics that impact the development process. For example, the framework can allow the use of complex logic in templates or no logic at all. It also can use a string or a DOM object as input for the template processor.

In the case of AngularJS, this a characteristic that represents a source of controversy. Since one of the main goals of AngularJS is to turn HTML into an appropriate language to declare dynamic views in web applications, application behavior can be declared using directives and expressions that accept code to define bindings, actions (function calls) and conditions. In AngularJS templates, it is possible to use logical operators such as the ternary (`?:`), OR (`||`), and AND (`&&`) operators. While many developers consider this as an advantage due to the expressive power given to the HTML, there are others that consider that putting code in HTML is a bad practice. In our mapping study, we found the following four problems about code in AngularJS templates:

1. *Silent failures:* The evaluation of expressions in AngularJS is forgiving to null and undefined values. It means that when undefined functions or objects are referenced in the template no exceptions are thrown [AngularJS Developer Guide,

2015a]. As a consequence, when a reference is misspelled in the template or it does not match the expected type, the application fails silently.

2. *Code hard to debug:* Since the code written in HTML templates is not JavaScript code, it is not possible to use a break point to inspect directly the value of the expressions. So alternative ways have to be used to do this work.

3. *Low readability of HTML:* The large amount of directives and expressions used to declare the bindings and behaviors of the application may be spread all over the HTML. These new elements and attributes may result in a verbose template decreasing the readability of the HTML document.

4. *Modularity and SOC loss:* The use of JavaScript code in HTML documents has been considered in some studies as a code smell because it violates the principle of separation of concerns (SOC) [Nguyen et al., 2012; Fard and Mesbah, 2013]. However, AngularJS expressions foster this practice.

To find out if the aforementioned problems have a real effect in the development of Web applications, we set a question (questions 16 to 19 in Appendix B.1) in which we asked participants to rate the frequency with which each item represents a problem in their daily development. For this purpose we provided them with a scale from *one* to *four* where *one* means that the item never was a problem and *four* means that the item is a very frequent problem. In Figure 3.4 we present how the responses for this question are distributed and how many answers we received for each item.

As a general observation we can see that none of the considered problems seems to have a strong effect in the application development of the interviewed population. All of the items have more than 60% of the answers in the low part of the scale (options *one* and *two*). The item with the largest number of answers in the high part of the scale is the one addressing the problem of silent failures with 39.2% of the respondents. Another problem that at least one third of the developers has faced is the difficult to debug or dump the variables used in the template. Figure 3.4 shows that 24.3% of the respondents rated the frequency of occurrence of this problem with *three* and 10.9% did it with *four*. This result is not surprising since these two problems are a recurrent topic in reviews about the bad parts of AngularJS [Koshelko, 2015; Horie, 2014] and in questions asked by the users of the framework [Stack Overflow, 2014].

On the other hand, lack of separation of concerns and low readability of HTML are considered as a problem of having code in templates by fewer developers. These statements were rated with the options in the low part of the scale by 71% and 78.3% of the developers respectively. The probable reason behind these results is that according

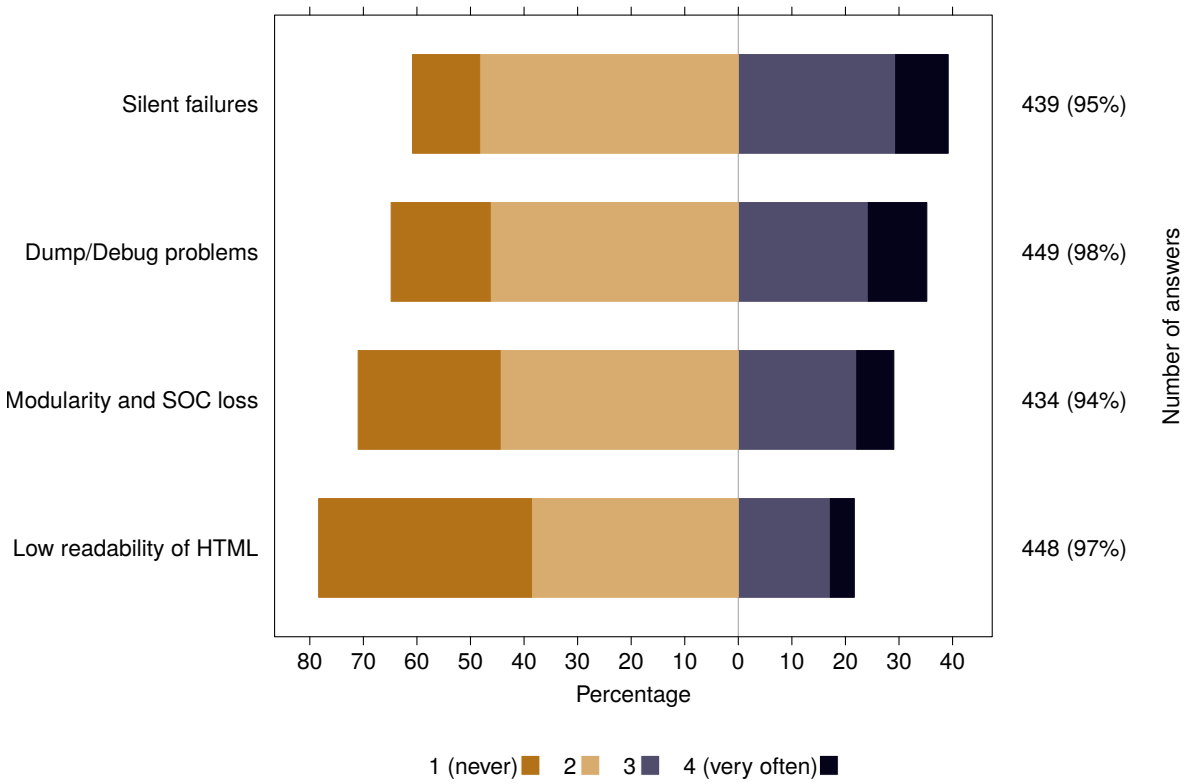


Figure 3.4. How often these items represent a real problem caused by code in AngularJS templates?

to good programing practices in AngularJS, complex expressions should not be used in templates [Gechev, 2013]. In other words, most of the code present in templates should be used just to reference objects and call functions.

Despite this recommended programing practice, we found evidences of AngularJS developers looking for a solution to the large amount of logic that they had placed in their templates [Stack Overflow, 2015b; Reddit, 2015]. Therefore, we set a question (question No 20 in Appendix B.1) to find out how often developers put large amounts of logic in HTML templates. Figure 3.5 presents the distribution of the answers to this question. We can see that at least one quarter of the population admitted that they had already placed large amounts of logic in HTML templates.

Since this is not a recommended practice in AngularJS development, we also were interested on finding out why some developers have followed it. Based on the cases that we found in the mapping study, we selected four possible reasons that could lead developers to use large amounts of logic in templates: (a) lack of experience in JavaScript, (b) lack of experience in architecting Web applications in general, (c) lack of experience in architecting applications using the abstractions provided by AngularJS

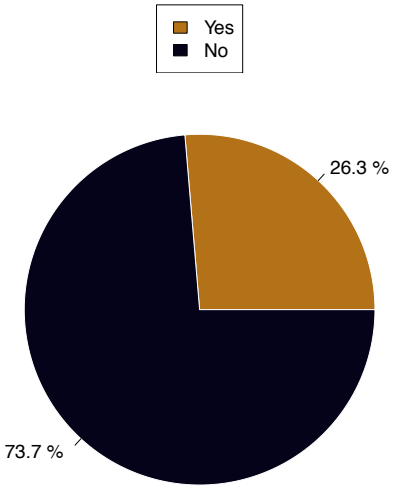


Figure 3.5. Have you ever placed large amounts of logic in HTML templates?

and (d) AngularJS design, which may foster this behavior. We asked developers who answered positively to the last question to select one or more of the aforementioned reasons to justify their behavior (question No 21 in Appendix B.1). We also included an "other" option to avoid forcing pre-defined responses for this question.

Figure 3.6 presents the results. We can see that the two most voted reasons were the design of AngularJS (which fosters this behavior) and the lack of experience to architect applications using AngularJS abstractions. They were selected by 54.8% and 43.5% of the developers respectively. The lack of experience in architecting applications in general was the third option with more votes (20.9%) excluding the "other" option. Only five developers selected the lack of experience in JavaScript as the reason that led them to put large amount of code in their templates. Another 27% (31 developers) expressed that they have had different motivations to follow this practice.

We manually analyzed and grouped these 31 answers to find out what were the most frequently mentioned reasons. Next we list the categories created from the one with more responses to the one with less responses.

- *Speed and deadlines:* Developers follow this practice because it is faster or because they had a tight deadline for the project.

- *Special cases:* In this case developers expressed that they were in a specific development where this practice was the best option. Additionally, they said that in that case the modularity or maintainability was not affected.

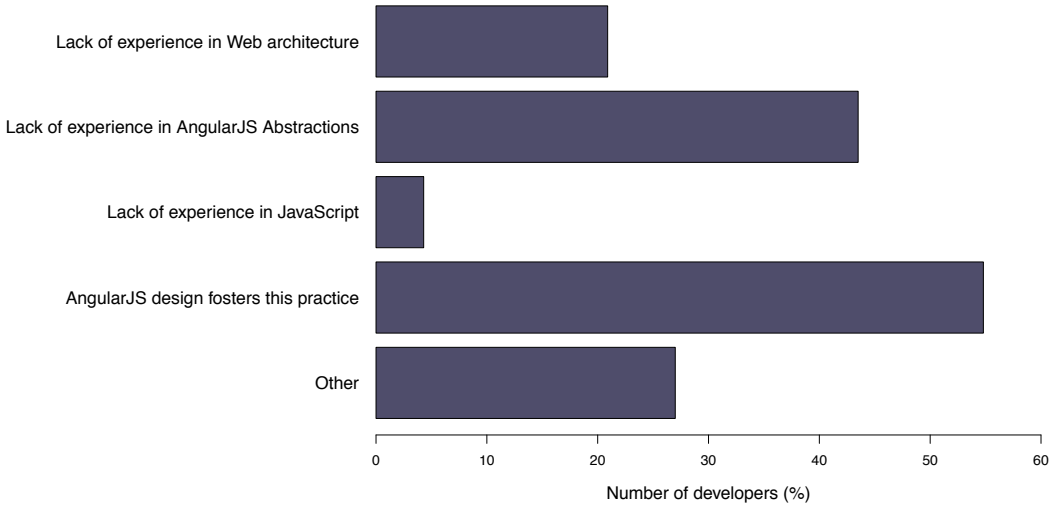


Figure 3.6. Reasons to place large amounts of code in HTML templates

- *Easiness and laziness:* They follow this practice by easiness or laziness. They did not want to implement an alternative solution such as creating a directive or a controller.

- *Earlier development or prototypes:* This practice is a convenient option in earlier stages of development or for prototyping purposes.

- *Maintenance or extension of old projects:* Some developers expressed that they had followed this practice because they had to maintain or extend code written by others.

- *View logic belongs to the template:* There was a group of developers that declared that this practice has no problems.

- *Hiding directives:* Some developers said that they have used large amounts of logic in directives related with hiding DOM elements (`ngShow`, `ngIf`, etc.)

3.2.4 Testing

Since one of the focus of AngularJS is testing, it is equipped with features such as dependency injection and mocking (which is offered through the `ngMock` module) to simplify this task. It also emphasizes the importance of placing the application code in the correct components to ensure separation of concerns and simplify the isolation of the code units under test [AngularJS Developer Guide, 2015e]. We were interested

in determining how often AngularJS developers make use of the features provided by the `ngMock` module and how complex it is to test each of the AngularJS components.

To achieve the first goal, we asked the participants to rate the frequency with which they use the `ngMock` module to test their applications (question No 22 in Appendix B.1). For this purpose we provided a scale in which *one* means that they never used the module and *four* means that they use the module very often. This question was answered by 441 developers (95.9%). As we can see in Figure 3.7, more than half of the developers (51.5%) never use the `ngMock` module. Additionally, 21.3% of them selected the option *two*, which results in 72.8% of the participants located in the low part of the scale. This result may have different explanations: low usefulness of the features provided by the `ngMock` module, unfamiliarity with the module, or a low rate of developers putting testing into practice.

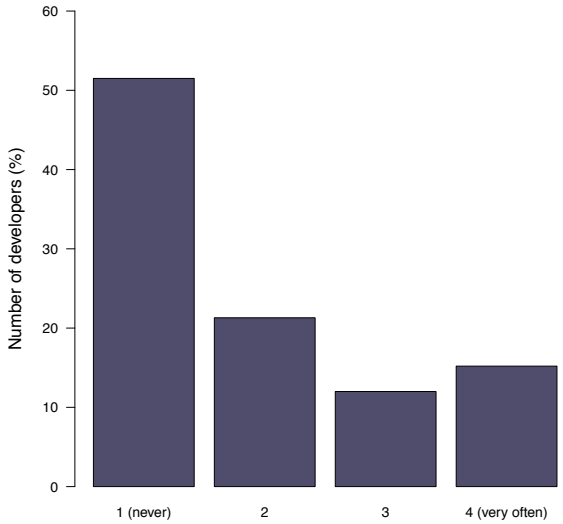


Figure 3.7. How often do you use ngMock module for testing?

To identify how complex it is to test each one of the AngularJS components, we listed them and asked the developers to rate from *one* to *four* how complex it is to test each one of them (questions 30 to 36 in Appendix B.1). In this case, the option *one* means that the evaluated component is very easy to test and the option *four* that it is very difficult. The functional roles of some of these components were described in Section 2.3.2. Figure 3.8 shows the number of participants that rated each component and how these ratings are distributed. As we can see, in this question we got a lower participation. The item with the highest number of answers references controllers and it was rated by 386 developers, which represent 83.9% of the sample.

Regarding the testability of each component, excluding the two first components,

all of them were rated as easy to test (options *one* and *two*) by more than 64% of the developers. From these components, it is special the case of the last three (services, filters, and controllers), which have a number of ratings in the option *one* considerably larger than the sum of the ratings in the options *three* and *four*. The reason behind these results can be that these components are very common and they are present even in the simplest applications. Additionally, most of the time these components do not include complex code and do not deal with complex APIs. As an example, we can see the case of filters. In general, since their unique responsibility should be formatting, most of their code only makes string transformations based on conditions and therefore it is easier to test.

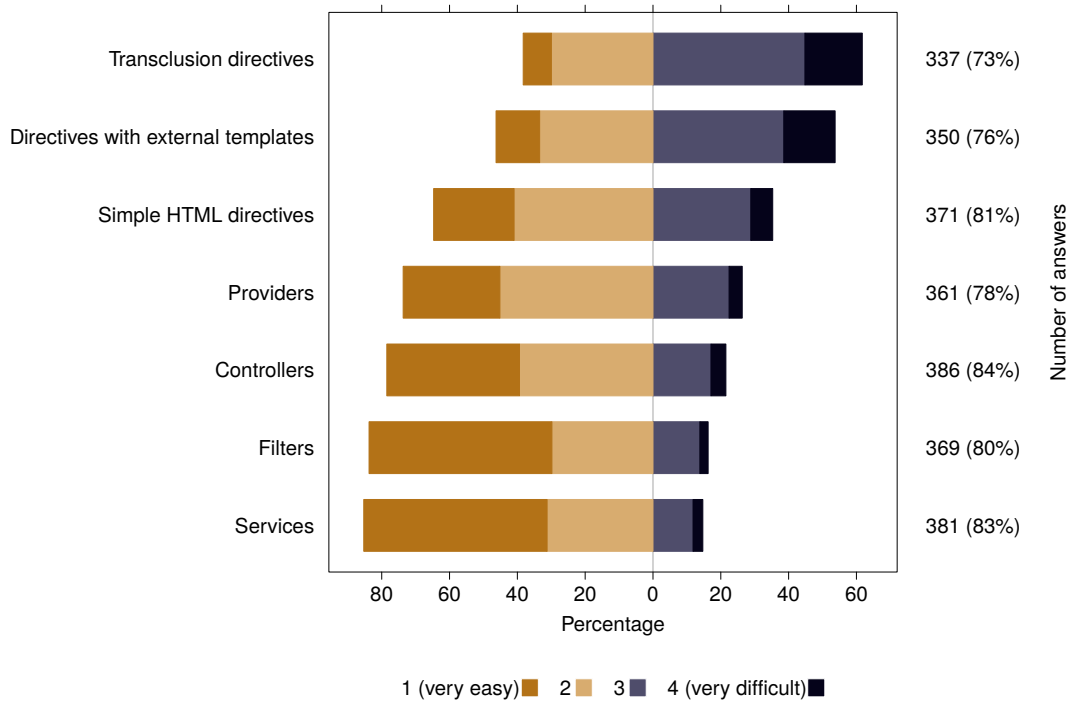


Figure 3.8. How difficult is it to test these AngularJS components?

On the other hand, the components that deal with DOM manipulation are considered by a larger number of developers as difficult to test (options *three* and *four*). This is the case of directives. When the template used to replace the markup of the directive is specified directly in the directive code, testing is an easier process. This type of directives, which we have called "simple HTML directives", was rated by more than a third of the developers (35.3%) with the options *three* or *four*.

Another approach to write custom directives is to place the template used to replace the markup of the directive in a separate HTML file. To avoid loading the

template over HTTP during test execution, it is necessary to pre-compile the template using a preprocessor named `karma-ng-html2js-preprocessor` and inject it into the test [AngularJS Developer Guide, 2015e; Hooker, 2014]. Figure 3.8 shows that 53.7% of the developers used options *three* or *four* to rate the difficulty of testing this type of directives. Finally, the components considered by the greatest number of developers as difficult to test are the transclusion directives. More than 60% of the participants rated this item using the highest options of the scale. This is not a surprising result since transclusion directives are often perceived as an obscure concept [Aston, 2014], or as providing a complex API [Parviainen, 2015].

In general, it is probable that developers find directives more difficult to test than other components because it is necessary to know many internal AngularJS concepts to build and test these components. For example, there are different types of transclusion, they can use different types of scopes, they have code that interacts with the DOM API, it is necessary to know the differences between the `link` and `controller` function, and it is fundamental to explicitly compile the directive and run the digest cycle during the test.

3.2.5 Mastering AngularJS

In this section we evaluate how difficult is to master some characteristics of AngularJS. The characteristics that we consider in this section were pointed in some blogs of the mapping study as being complex to understand or even as design errors. We selected nine of these characteristics and asked participants to rate them according to the complexity to comprehend them and use them (questions 41 to 49 in Appendix B.1). Below we list the characteristics considered and present a brief description of each one.

1. *Prototypical Inheritance of Scopes:* In AngularJS, expressions are evaluated against a context represented by an object called scope. Each directive used in the template can create a new scope. If two directives that create new scopes are nested in the DOM tree, the scope created by the inner directive prototypically inherits from the scope created by the external one. If prototypical inheritance is not understood, developers may experiment unexpected behaviors due to "global access" or overwriting of properties in the child scope.

2. *Directives scopes:* Customized directives can be created with three different scope options. A directive can use the scope of its parent, it can use a new scope that prototypically inherits from the scope of its parent, or it can use a new isolated scope. When an isolated scope is used, it is necessary to specify how the properties

of this scope are bound to the parent scope. There are four types of bindings that can be specified. Finally, if two directives are used in the same HTML element, depending on the type of scope that each one create, there can be compatibility problems.

3. *Five types of entities to register dependencies:* In AngularJS the dependencies of the application have to be registered so that later they can be injected in the different modules. For this purpose, AngularJS offers six different methods that differ in the type of data to be registered (constant, value, service, factory, decorator, and provider) and the access that other components have to them. Some blogs suggest that this is an unnecessary complexity, which could be replaced by a unique method [Eidnes, 2014].

4. *Compile, link and controller functions:* It is possible to implement directives that are able to manipulate the DOM, register event handlers, and interact with other directives. However, their declaration needs to specify properties such as `controller`, `compile` and `link`, whose value must be a function or an object containing it. In the mapping study we found evidence that it is not always clear what is the purpose of each function and in which cases they must be used. Additionally, the developer has to know very well some internal concepts of AngularJS such as pre-linking and pos-linking to declare these functions.

5. *Transclusion directives API:* The creation of transclusion directives is considered complex because in addition to all the configurations necessary to create a simple directive (some mentioned in the previous items), it is necessary to deal with new options and concepts that apply only to these directives, including three types of transclusion, transclusion functions, transclusion scopes, etc.

6. *Correct use of `$scope.$apply`:* `$scope.$apply` is a method available in every scope of the application that runs the digest cycle over `$rootScope`. This method can be explicitly called to detect changes made in the model. However, we found evidence of developers that find difficult to understand how this method works and when it has to be used [Stack Overflow, 2013b,c].

7. *Tackling of performance degradation details:* As will be presented in Section 4.2.4, it can be challenging to know all the technical aspects of writing applications in AngularJS to avoid performance problems.

8. *External component integration:* To introduce some third party plug-ins in the AngularJS world, most of the time it is necessary to create a new directive to

wrap its code and use it. Additionally, to update the changes made in the model by an external code, it is necessary to explicitly call the `$scope.$apply` method.

9. *Correct syntax to inject dependencies:* The simplest method that can be used in AngularJS to inject dependencies calls `toString` in the function defining the new module and extracts the names of the dependencies from its parameters and looks for them in the list of registered dependencies. The problem with this method is that it does not work anymore after minification of code. As an alternative, AngularJS provides another two methods in which the names of the dependencies are specified using strings. However, some developers complain about having three different methods to do the same task [Eidnes, 2014; Koshelko, 2015].

To rate the complexity of use of the aspects and features listed above, we provided participants with a scale in which *one* means that the evaluated item is very simple to use and *four* means that it is very complex. Figure 3.9 shows the distribution of the responses and the number of answers received for each item. For this question, the item with less answers was rated by 86.3% of the sample (397 developers) and the item with more answers was rated by 92.4% of the sample (425 developers). As a general observation, we can say that excluding the items that reference "the different methods to register dependencies" and "the integration with external components", all the considered aspects and features were rated as complex to comprehend and use (options *three* and *four*) by at least 43.5% of the participants.

The bar in the top of Figure 3.9, represents the most complex aspect of AngularJS according to the developers. Tackling all the details that can lead to performance degradation was rated by 32.4% of the participants as a very complex task. Additionally, 43.6% of them used the option *three* to rate this item, which results in 76% of votes in the high part of the scale. This result shows how frequent AngularJS developers have concerns about the performance of their applications and strengthens the relevance of the study presented in Chapter 4.

The next four items also have a distribution in which most developers used options *three* or *four* to assess their complexity of comprehension and use. The API to build transclusion directives and the different types of scope that can be used to build a directive were rated with these options by 61.9% and 58.5% of the developers respectively. The use and comprehension of the functions `compile`, `link` and `controller` used to declare directives were rated with the options in the high part of the scale by 55.4% of the participants. Finally, these options were used by 52.2% of the participants to rate the complexity of use and comprehension of the method `$scope.$apply`.

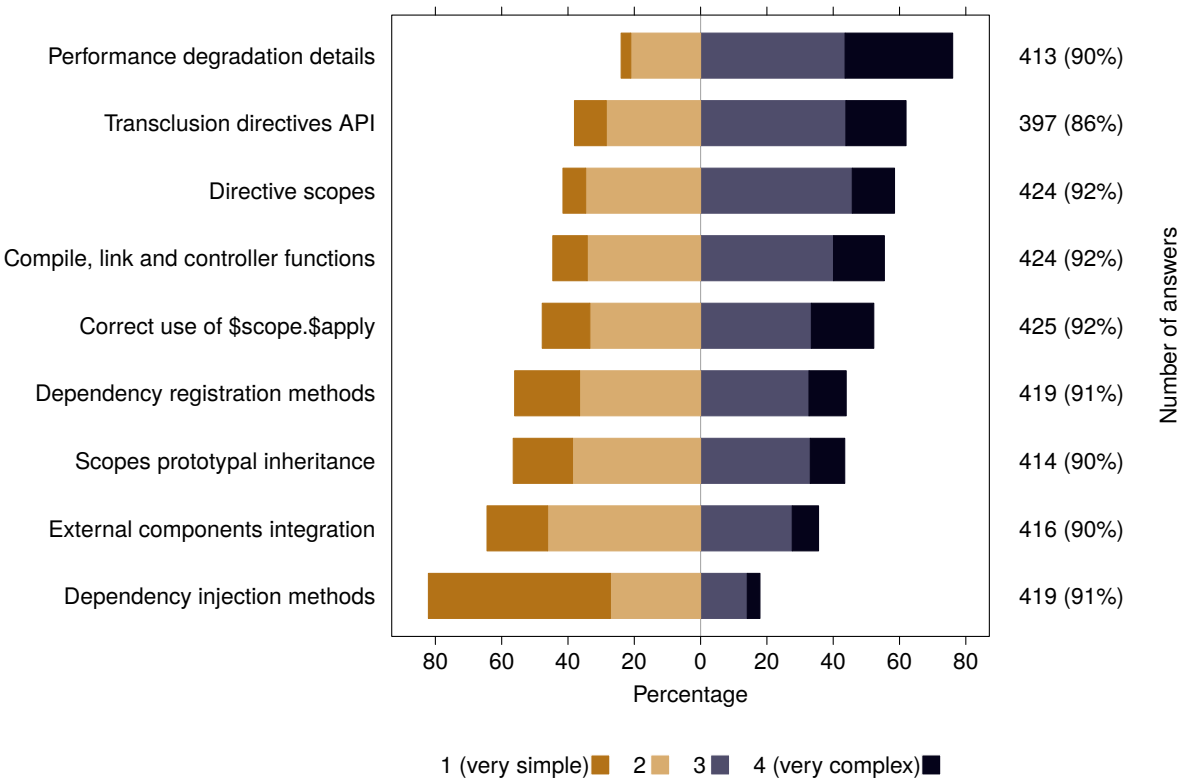


Figure 3.9. How complex is the comprehension and correct usage of the following AngularJS aspects and features?

From these results, it is worth noting that the first three items, after "performance degradation details", are related with the implementation of directives.

As an additional information, we want to introduce the answers obtained from a complementary question. In this question (question No 38 in Appendix B.1), we asked developers about the frequency with which they create their own directives. As usual, to answer this question they used a scale from *one* to *four* in which *one* means that they never create custom directives and *four* means that they create them very often. This question was answered by 451 developers, of which 57.6% selected the option *four* (very often) and 25.9% selected the option *three*. This result shows clearly that this is a common practice in AngularJS applications.

Since directives are the way to implement custom UI components, we can conclude by examining the results obtained in Section 3.2.2 and in the previous question, that although directives are highly appreciated and used by the AngularJS community, it is complex to understand and to use the API provided to implement them.

The remaining items in Figure 3.9 were rated by most developers with the options in the low part of the scale. From these items it is worth noting the case of the item

representing the integration with external components that was rated as a simple task (options *one* or *two*) by 64.5% of the developers; and the case of the item representing the use of the correct syntax to inject dependencies to avoid problems with minification, which was rated with the options *one* or *two* by 82.1% of the developers. For the last item is also notable that more than half of the developers (54.9%) used the option *one* to rate this aspect.

3.2.6 AngularJS 2.0

The AngularJS team is currently working in the version 2.0 of the framework. Important changes and new features are intended to improve the development experience of the version 1.0. However, as we found in different blogs reviewed in the mapping study, many developers are concerned with changes that break compatibility with version 1.X and with the absence of a clear migration path from versions 1.X to version 2.0 [Hamilton, 2014; Petersen, 2014]. As a consequence, developers would not only have to update their applications, but could also have to rewrite them. There are two solutions to solve this problem. The first is to adopt programing practices to write applications with AngularJS 1.X that allow an easy migration to version 2.0 [Takhteyev, 2015; Krivanov, 2015]. The other solution is to use the features `ngUpgrade` and `ngForward`, which were provided by the AngularJS team to allow developers to (1) use the two versions at the same time or (2) write AngularJS 1.X applications using AngularJS 2.0 syntax [Angular Blog, 2015a].

Since we were interested in determining how many developers are introducing changes in their applications to facilitate migration to version 2.0, we asked participants to rate the frequency with which they make design decisions in their current development having in mind compatibility with AngularJS 2.0 (question No 50 in Appendix B.1). For this purpose, we provided them with a scale in which option *one* means that they never make design decisions having in mind compatibility with version 2.0 and *four* means that they do it very often. This question was answered by 406 developers (88.3%). We can see from Figure 3.10 that only 10.6% of the developers answered with option *four* (very often) and in total only 23.2% of them chose one of the options in the high part of the scale. On the other hand, 55.4% of the respondents declared that they never make design decision having in mind compatibility with version 2.0 and 21.4% used the option *two* to rate the frequency with which they do it. These results suggest that there is a considerable number of developers that are not contemplating a migration to version 2.0 or that at least they are not following the advised recommendations to simplify this migration.

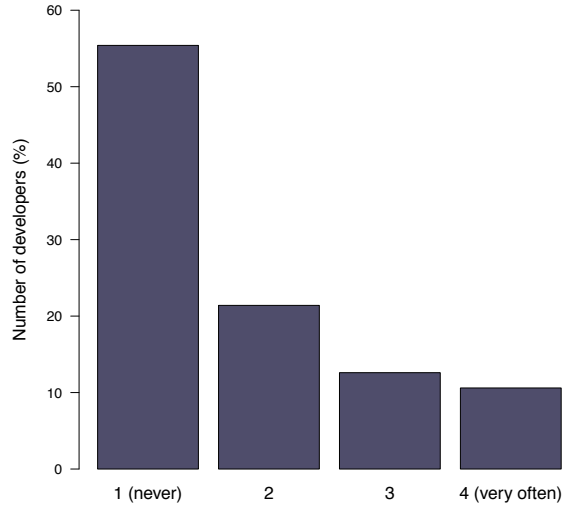


Figure 3.10. When developing your current applications, how often do you make design decisions having in mind compatibility with AngularJS 2.0 release?

This result is reinforced with the answers received for one of the complementary questions. Since the `$scope` reference will no longer exist in AngularJS 2.0, one of the recommendations to ease the migration is to avoid the use of this reference in applications written with versions 1.X. Therefore, we set a question to find out how often the participants avoid the use `$scope` in their applications (question No 37 in Appendix B.1). Again, they used a scale from *one* to *four* in which *one* means that they never avoid the use of `$scope` and *four* means that they avoid this reference very often. The question was answered by 453 developers (98.5%) and the answers show that 34.2% of the respondents selected the option *one* to rate the frequency of this practice in their development and 25.4% of them selected the option *two*. Maybe this result is not as overwhelming as the one obtained for the previous question, but it is important to keep in mind that migration to version 2.0 is not the only motivation to avoid the use of `$scope`.

To reveal the new features or improvements in AngularJS 2.0 that are most expected by developers, we selected 10 features extracted from the mapping study and asked developers to select the ones that they were most eager to use (question No 51 in Appendix B.1). Since these features were selected when AngularJS 2.0 was still in its earliest development phase, they are likely to change or they may even not be implemented in the final release. The construction of this question is also based on another question made in a survey built by the AngularJS team [Angular Blog, 2015b]. Next we list the characteristics considered for this question:

1. The new API to define the main UI building blocks (components and directives)
2. Zone.js (no more `$scope.$apply()`)
3. Server rendering
4. The new logging service called diary.js (an execution time measurement service)
5. New mobile features (e.g. support for touch event gestures)
6. The new template syntax ([properties] and (events))
7. The new change detection mechanism (immutable/observable)
8. The new API for injecting dependencies
9. Integration with ES6 modules
10. Component Router (which will enable the asynchronous loading of different components)

For this question we received a total of 378 answers (82.1%). Figure 3.11 shows how they are distributed. We can see that the feature selected by the greatest number of developers (68.3%) is the integration with ES6 modules. The integration with the module system defined by the new ECMAScript specification[4] allows to load the components of the application asynchronously and on demand. The second most voted feature was the new API to create UI components, which was selected by 60.1% of the developers. This is not a surprising result since, as we have seen in Section 3.2.5, the current API is perceived by a significant amount of developers as difficult to use and understand. The third feature most expected by AngularJS developers is the component router, which was selected by 56.1% of the developers. The advantage of this feature is that it allows the dynamic loading of UI components in the view. Finally, the new change detection mechanism was chosen by 48.7% of the developers. This is one of the most relevant design changes in the new version of AngularJS and it is expected to represent a significant improvement in the performance of the framework.

3.3 Threats to Validity

The first threat to validity for this study can be found in the execution of the mapping study. Due to the large amount of information available on the Internet about AngularJS, it is possible that literature addressing more specific topics about AngularJS or

[4] `http://www.ecma-international.org/ecma-262/6.0/`

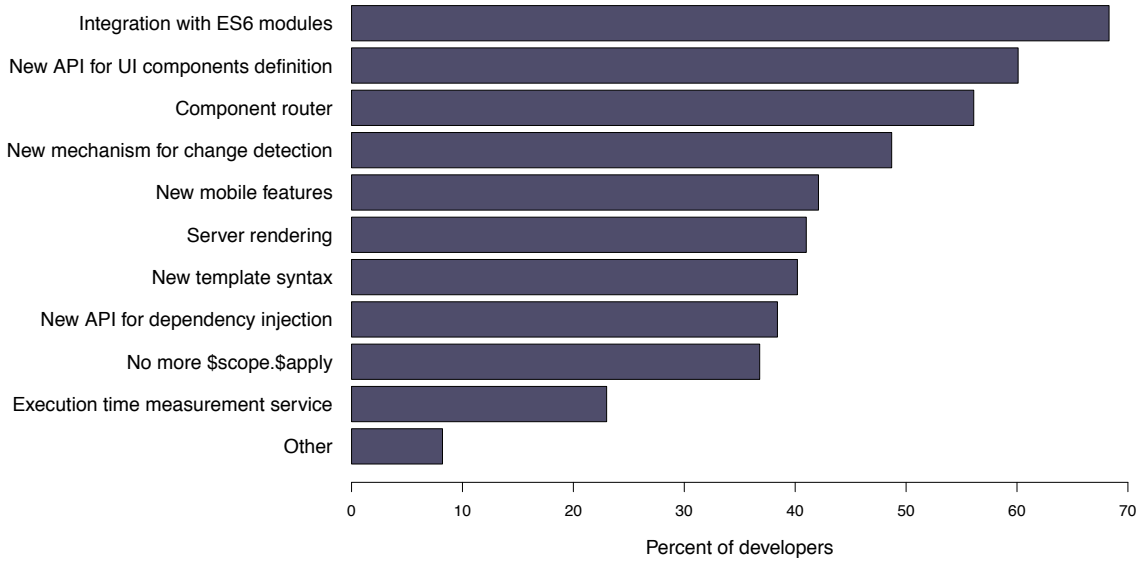


Figure 3.11. Most expected AngularJS 2.0 features

presenting different points of view has not been included in the analyzed sources. The incorporation of this information could have generated a different construction of the questionnaire (e.g, different options to answer a specific question) leading to different results.

The main threat to validity related with the construction of the questionnaire is the insertion of ambiguous and leading questions. We made our best to avoid this problem by constantly reviewing and improving the proposed questions. Additionally, we ran a pilot survey to identify and correct this type of questions. In some questions we also gave the participants the opportunity to respond with an answer different from the proposed ones by adding an "Other" option. Furthermore, with the exception of the background questions, no question was mandatory to avoid forcing responses.

There are also two threats to validity related to the method used to construct the sample of the survey. The first one is related with the match between the Stack Overflow profile and the GitHub profile of the participants. It is possible that a Stack Overflow user has been matched with a homonym user in GitHub. (i.e., users in Stack Overflow and GitHub who have the exactly same name, but who are different people). As a result, the questionnaire could be received by people with no experience in AngularJS. Additionally, it is possible that the heuristic used to assess and rank the experience of the selected developers does not reflect the reality. Consequently, we may have removed experienced users from the top of the rank or we may have included

novice users in the top of the rank.

There are also some aspects that limit the generalization of our results (external validity). First, the sample for the survey was selected only from the Stack Overflow community. It is possible that the findings in this study do not apply to a different population. Moreover, constant and rapid changes in the Web development environment, including new technologies and new versions of AngularJS, can lead to different results if the study is repeated in the future.

Finally, we have to mention the threats to validity related with human behavior. As an example we can mention the ordering of the questions since one may provide context for the next one. Another threat to validity is the attitude of the participants towards the topic of research. Their responses can introduce bias to make AngularJS appear in a positive or negative light. We can mention the case of one of the participants who used the comments space to declare that he is a contributer of the AngualrJS project.

3.4 Final Remarks

In this chapter we reported an empirical study about different aspects of AngularJS based on opinions and experiences of AngularJS developers. In Section 3.1 we described the two phases that compose the whole study and how they were executed. First, we described in details the mapping study that was used to systematically collect information about AngularJS by using different sources found in the Internet. Then, in Section 3.1.2, we presented a description of the design of the survey, including characteristics of the questionnaire and how we generated a sample of AngularJS developers from Stack Overflow. We also included information about the number of developers invited to the survey and the number of responses received (response rate).

In section 3.2 we presented the analysis of the responses provided by the 460 participants of the survey. We showed that the survey was answered by participants with different levels of experience in AngularJS, but that there is a predominant group that has one to three years of experience with the framework (74.8% of the sample). In Section 3.2.2 we found that one of the most valued features of AngularJS is the possibility of creating custom UI components by using directives. The value of this feature was rated with the highest option of the scale by 68.1% of the respondents. However, as we noticed in Section 3.2.5, three of the most difficult aspects of AngularJS are related with the API to declare directives. Furthermore, directives are also considered as the components most difficult to test by a considerable amount of developers

(Section 3.2.4). From Section 3.2.5 we also identified that the most complex aspect of AngularJS is to tackle all the details that can lead to performance degradation. This conclusion is supported by the number of developers that used the option *three* or *four* to rate the complexity of this aspect (76%). These results are just examples of the findings provided by this study, but we also investigated other themes such as perception about the AngularJS templates (Section 3.2.3) and most expected features of the version 2.0 of this framework (Section 3.2.6).

Finally, in Section 3.3 we discussed the threats to the validity of this study which are mainly related with the method used to generate the sample, the social behavior presented by the respondents, and the construction of the survey instrument.

Chapter 4

AngularJS Performance Survey

This chapter presents an empirical study about different aspects of AngularJS performance, which uses as its main source of information qualitative data collected through a descriptive survey. The study design is presented in Section 4.1 and describes in detail the *Mapping Study* and the *Survey Design* that represent the two phases in which the study was conducted. In Section 4.2 we present the results of the survey and a brief discussion about them. Threats to validity are presented in Section 4.3. Finally, we conclude this chapter with some final remarks presented in Section 4.4.

4.1 Study Design

This study was conducted in two complementary phases. The first part is an exploratory empirical research that aims at obtaining knowledge about the state-of-the-art on practices, problems, and solutions regarding AngularJS performance. The second part of the study consists of a survey research whose construction is supported by the results delivered by the first research. The following subsections describe the relation between these studies and how they were designed and implemented.

4.1.1 Mapping Study

The main goal of this part of the study was to gather as much information as possible about AngularJS performance, including but not limited to the frequent problems faced by developers who use the framework, the causes behind these problems and the possible solutions to them. The information collected in this study was used as a basis to build an exploratory survey that would help us to validate the information itself

and provide useful findings based on the experience of different developers using the framework.

To achieve this goal we performed a review of the existing literature addressing this topic. We made an effort to find related literature in recognized digital libraries of Computer Science, but we could not find many titles. As mentioned in Section 3.1, there are very few studies about JavaScript MVC frameworks such as AngularJS and no one focus on performance. For this reason we decided to perform a mapping study (or scoping study) which is a research method less stringent than a systematic literature review [Wohlin et al., 2012]. A mapping study is more suitable for fields of study that have been less explored. Our efforts were focused on the Internet where the Google search engine was used as the main tool in the investigation.

We considered any informal literature contributing with information about the causes of performance problems in AngularJS, how often these problems occur, what are the practices suggested by experienced developers to avoid them, and what are the most common mistakes made by the less experienced ones that lead to these problems. We started our research using search queries such as "*AngularJS performance*" or "*performance problems in AngularJS*". The results listed in the first positions were examined manually to verify that the topic was inside the ones that we were interested on. The literature checked included mainly blogs, but we also checked forums, community groups, and Q&A sites such as StackOverflow.

During the revision of blog posts we applied a criteria to select or discard a text. We only maintained those posts that were well-written and in which the described background of the author included some experience in software development.

To simplify the analysis of the collected information we systematically logged in a document the most relevant quotations found in the different web sources. This document also has important data about the source such as the access URL, the date in which it was accessed, the name of the author, and the professional information of the author (when this information was available). An identifier was assigned to each source of information and each quotation registered in the created document uses these identifiers to reference the source from which it was taken. A copy of the created document is presented in Appendix A.2

We also used other ways to reach more literature in the web regarding AngularJS performance. While we were going through the literature already found, we also visited those sites referenced in the text, resulting in the application of a similar strategy used in systematic literature reviews called "snowballing" [Skoglund and Runeson, 2009]. Furthermore, when a topic was frequently mentioned during the revisions, a new search query was used to find more information about that specific topic. As an example,

we can mention the search query "*ng-repeat performance*" that was built due to the frequency with which the `ng-repeat` directive was brought up in the sources reviewed. The data registered in the created document was then further analyzed to find some trends about problems, practices and opinions in the realm of AngularJS performance. Then common themes were identified and used as the basis for the construction of our survey.

4.1.2 Survey Design

As a result of the mapping study, we constructed a survey (built in Google Forms) with 31 questions that could be answered in approximately 10 minutes. The questionnaire is mainly made up of multiple choice questions and rating questions that use nominal or ordinal scales to measure the information provided by the respondents. Some of the multiple choice questions have only two possible options (yes or no). In a subgroup of these questions we used optional open-ended questions, which foster the respondents to extend the information provided by asking about a more specific point of the base question topic, so we could draw better conclusions in the analysis stage. When it was considered necessary, the multiple choice questions with more than two options had an additional option with statements such as "I am not sure" or "I have no idea" to avoid forcing a response when the respondents did not have enough knowledge to answer the question.

We used a five points ordered response scale for all of the rating questions. Labels were only put in the extremes of the scale to show the respondents the range of possible responses. For example, if we were asking for the frequency of a practice, the scale presented to the respondents included the labels "very rarely" and "very often" as options one and five, respectively.

At the end of the questionnaire we provided the respondents with a space in which they could leave an optional comment that they considered relevant for our research. We also gave the respondents the opportunity to specify their names and emails in case they wanted to receive the results of our research.

The questionnaire was divided into four sections that collected information about (1) the background of the users, (2) their development practices and perceptions, (3) general causes of AngularJS performance problems and (4) technical causes of these problems. The background section collected information about the familiarity of the developers with JavaScript and AngularJS, as well as the size of the largest project they had developed with these technologies. In the second section, we included questions related mostly to perceptions about AngularJS performance, and practices and common

solutions to deal with this aspect of an application. In the third and fourth parts of the questionnaire, the causes for applications with poor performance were addressed. The third part focused on general causes such as the experience of the developer or the architecture of the application, while the fourth part focused on technical causes such as incorrect use of `$scope.apply` or unnecessary use of two-way data binding. It is worth mentioning that due to the technical knowledge required to answer this last section, we explicitly asked the respondents not to answer the questions in which they felt that they did not have the knowledge required to provide an answer.

Since the survey targets developers familiar with AngularJS we used some well-established communities to look for the respondents. To promote the survey in these groups we created a standard post in which we explained the objective of the research and invited the members to participate in the survey. This post was then published in the selected communities.

After elaborating the first version of the questionnaire we ran a pilot in two national groups: a Google group[1] and a Google+ community[2]. We did it with the purpose of detecting possible improvements that could be made on the survey instrument. After the pilot round, we only found and corrected typographical errors.

The final version of the survey was promoted in other communities including the AngularJS community in Google+[3], the AngularJS Google group[4], and a community built in Reddit for AngularJS developers[5]. Furthermore, we also spread the survey link in Twitter accounts related to the AngularJS ecosystem. The survey was open during approximately three weeks (since early September 2015) to receive responses and in this period we obtained 87 responses. Since the unique corrections made to the first version of the questionnaire were of typographical type, we also considered the responses from the pilot phase to obtain a total of 95 responses.

Due to the nature of the methodology used to promote the survey, it is not possible to calculate accurately the response rate. Even when the number of members of each community is known, we can not determine how many of them visualized the survey post and how many of them are members of multiple communities.

Appendix B.2 includes a complete copy of the survey.

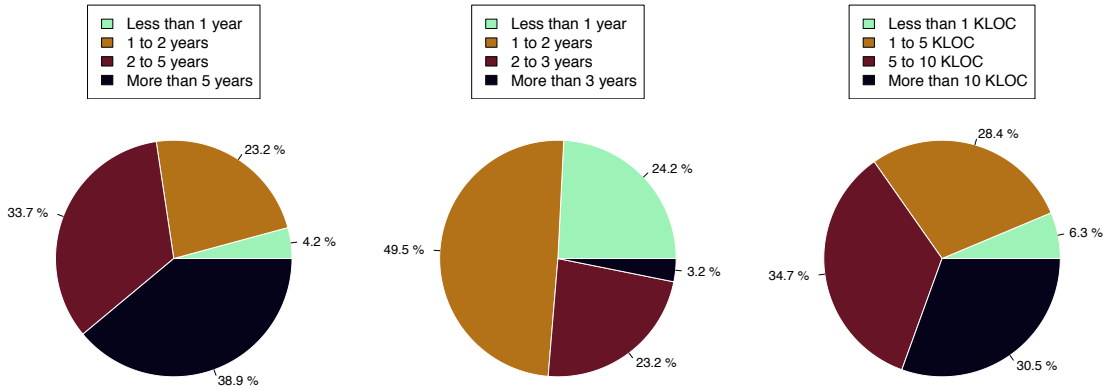
[1] `https://groups.google.com/forum/#!forum/angularjs-br`
[2] `https://plus.google.com/communities/110372004895086149874`
[3] `https://plus.google.com/communities/115368820700870330756`
[4] `https://groups.google.com/forum/#!forum/angular`
[5] `https://www.reddit.com/r/angularjs/`

4.2 Results

In this section, we present the results of the survey and the analysis of the responses. The questions are grouped by topic to facilitate their interpretation. The first group provides information about the respondents background and it is the base for making comparisons through the rest of the survey.

4.2.1 Background

Figure 4.1 presents the distribution of the background answers. These answers give us a picture about the experience of the participants in JavaScript and in AngularJS. As we can see (Figure 4.1(a)), 72.6% of the respondents reported an experience of at least two years in JavaScript and 38.9% reported to have been working more than five years with the language. Figure 4.1(b) shows that 24.2% of the developers have less than one year of experience in AngularJS. The other part of the developers (75.8%) is divided as follows: 72.7% declared to have an experience from one to three years in the framework and only 3.2% to have an experience for more than 3 years. This is probably due to the date of creation of AngularJS (2009) and the date in which it became popular (2013).



(a) JavaScript experience (b) AngularJS experience (c) Largest application

Figure 4.1. Respondents background

In order to complement the background information, we asked developers about the largest application they have developed using AngularJS. The results are shown in Figure 4.1(c). Only 6.3% of the developers reported that their largest application has less than 1 KLOC. On the other hand, 30.5% declared to have developed applications

with more than 10 KLOC. The other 63.1% said that their largest application has between 1 KLOC and 10 KLOC.

Therefore, most of the respondents have a considerable experience in JavaScript while a more distributed experience is observed in AngularJS. Nevertheless, half of the participants have an experience in the framework that ranges from one to two years. In relation to the size of the applications implemented by them, we have a similar proportion of developers writing small, medium, and large applications with AngularJS.

4.2.2 How Developers Improve AngularJS Performance

As a result of the mapping study we found some trends in the practices followed by developers to improve the performance of their applications. Some of these practices suggest that developers need to know implementation details of the framework, substitute or improve some parts of the framework with other components, or even modify the source code of the framework. As an example we can see the following descriptions of the components created for this purpose (Czerwinksi [2013]):

sly-repeat is our variant of ng-repeat, which hides extra DOM elements rather than destroying them. sly-evaluate-only-when prevents inner change watchers from executing unless... And sly-prevent-evaluation-when-hidden prevents the inner repeat clause from executing until the mouse moves over this line and the div is displayed.

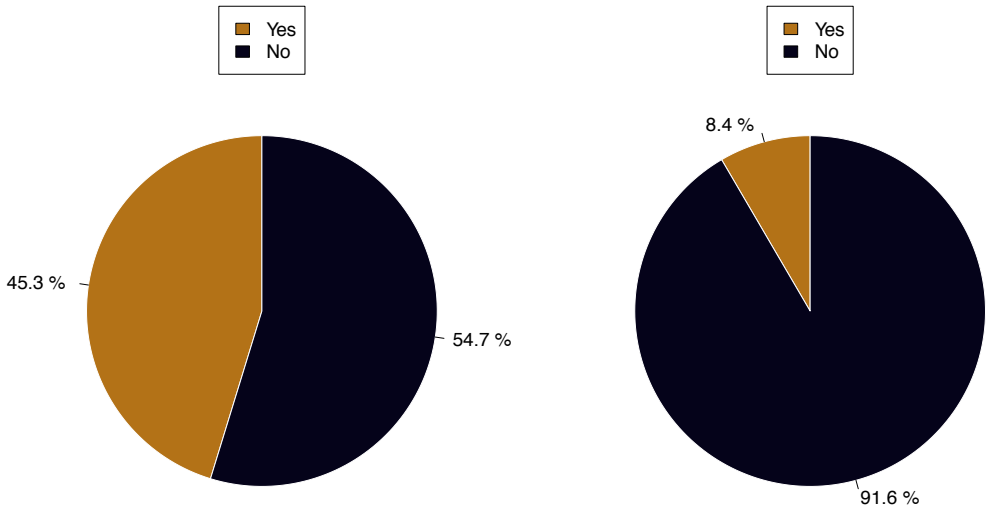
At the end of the post it is also said that:

We overrode the Scope's $watch method to intercept watcher registration, and then had to do some careful manipulation of Scope's instance variables to control which watchers are evaluated during a $digest.

To find how popular these practices are among AngularJS developers, we created a set of questions whose results are discussed below. These practices do not include actions such as small refactoring operations.

Figure 4.2 presents the distribution of the responses about interactions with the source code of AngularJS (questions seven and eight in Appendix B.2). We can see how often developers have inspected the AngularJS source code looking for performance improvements and how often they even modified the source code for this purpose. Figure 4.2(a) shows that almost half (45.3%) of the developers have at least once

inspected the AngularJS source code to figure out how to improve the performance of their applications. This is a clear evidence that developers seek to understand the internals of AngularJS when fixing performance problems. On the other hand, only 8.4% (8 developers) reported to have made changes in the AngularJS source code to deal with performance problems (Figure 4.2(b)). The reason is probably because this is a bad practice with serious implications in the maintenance and evolution of the software product.



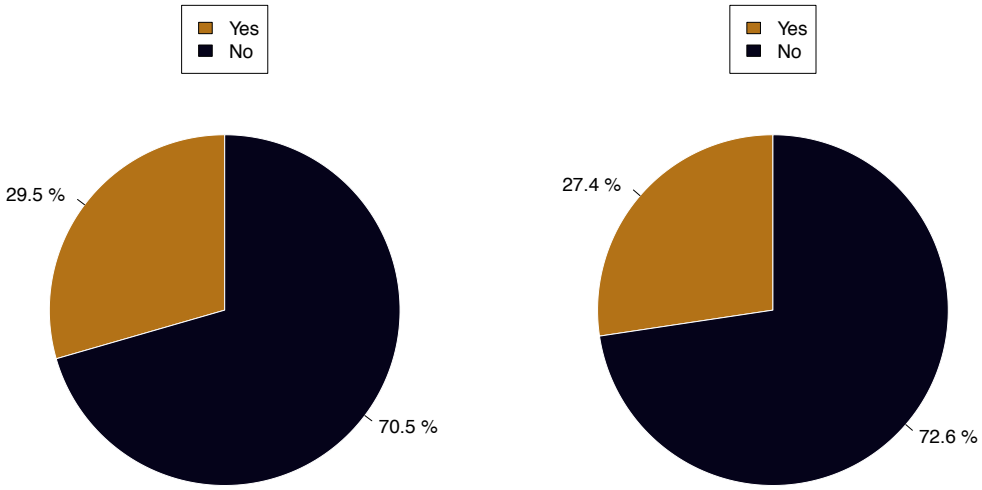
(a) Have you ever inspected AngularJS source code to improve performance?

(b) Have you ever changed AngularJS source code to improve performance?

Figure 4.2. Interactions with AngularJS source code

Another trend found about how developers improve the performance of their applications in AngularJS regards the substitution of features provided by the framework with custom components. These components are created by the project owner itself or by third parties. Figure 4.3 shows the percentage of developers who have created or used third-party components to improve the performance of their applications (questions 9 and 11 in Appendix B.2). As we can see 29.5% of the respondents reported that they have used third-party components to improve performance and 27.4% reported that they have created a custom component for this purpose.

To reveal the roles of these components, we set two open-ended questions asking (1) the name of the third-party components used and (2) a description of the custom components created (questions 10 and 12 in Appendix B.2). We analyzed these answers qualitatively. For the third-party components we used the names to search for



(a) Have you ever used a third-party component to improve the performance of an AngularJS application?

(b) Have you ever created a custom component to improve the performance of an AngularJS application?

Figure 4.3. Developers using and creating components to improve application performance

the purpose of their creation on the Internet. We found that the two most popular reasons for which these components have been used are: (1) to be able to bind once when this feature was not available in AngularJS (before version 1.3) and (2) to substitute or improve the standard `ng-repeat` directive. In the first category, the only component mentioned was `bind-once` while in the second category we found names such as `angular-virtual-scroll`, `infinite-scroll`, `ng-table` and `collection-repeat` from the Ionic Framework.

When analyzing the descriptions of the created custom components we found a clear trend in their purpose. Most of them deal with large amounts of data that need to be rendered as tables or long lists. This result is consistent with our mapping study, when we found a considerable number of developers facing problems with the `ng-repeat` directive when handling large collections of data, and developers and third-party modules providing solutions for this problem. As examples, we can mention some posts found in StackOverflow with titles like *"Angular performance in ng-repeat"* [Mustafa, 2014] and *"How to improve performance of ngRepeat over a huge dataset (angular.js)?"* [Stack Overflow, 2013a].

The last result that we present in this section has to do with performance in mobile devices. In some of the reviewed blogs, we found developers saying that perfor-

mance problems in AngularJS got worse in mobile development [Koshelko, 2015; Koch, 2015]. Therefore, we asked developers if they had ever made refactorings in their applications due to performance issues in mobile devices (question No 13 in Appendix B.2). Figure 4.4 shows that 34 developers (35.8%) answered positively to this question.

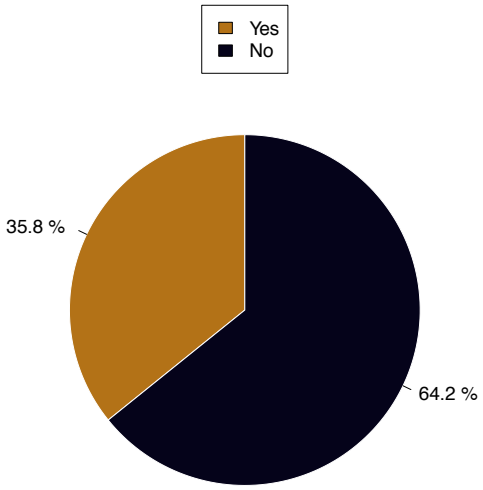


Figure 4.4. Developers who refactored their AngularJS applications specifically to improve performance on mobile devices

4.2.3 General Causes of Performance Problems in AngularJS

There are different factors that may lead to low performance in AngularJS. At the highest level, the causes of performance issues can be things like the experience of the developer or the architecture of the framework. We decided to include some general questions to find out what are the most probable causes based on the experience and opinions of the developers. We chose four possible reasons: (a) the lack of experience in the language (JavaScript), (b) the lack of experience in the framework (AngularJS), (c) applications following wrong architectures and (d) the internal architecture of AngularJS. Figure 4.5 presents the answers of the developers on how often these causes are responsible for applications with performance problems (questions 18 to 21 in Appendix B.2). To make this evaluation, they rated each cause in a scale from one (very rarely) to five (very often).

The first group of bars (from left to right) in Figure 4.5 represents the result for the first cause: lack of experience in JavaScript. The most common answer was the neutral option (*three*) with 28.4%. The number of answers for the options *two* and *four*

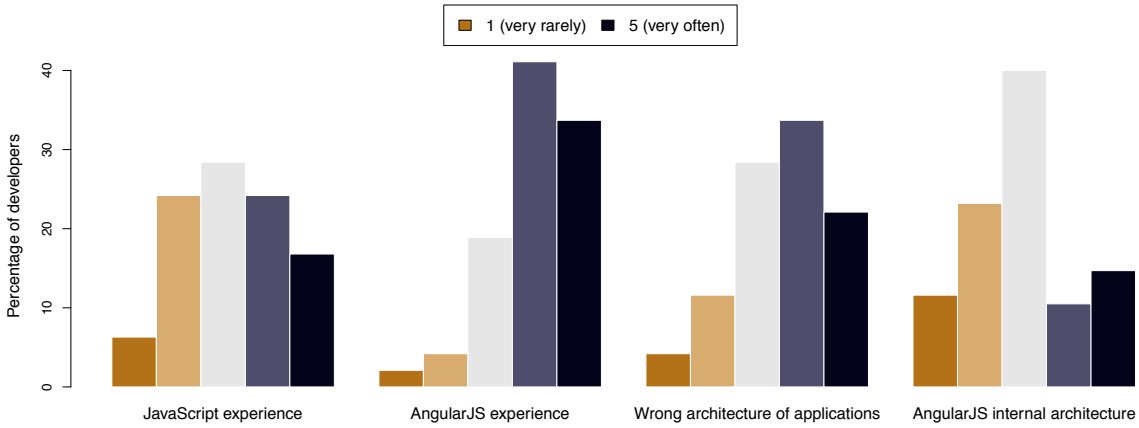


Figure 4.5. Evaluation of general causes of performance issues in AngularJS

was the same (23 answers), and the difference between the extreme options (10.5%) is not enough to draw a conclusion.

On the other hand, the second group of bars shows a more clear tendency in the responses. The answers of 74.8% of the respondents correspond to one of the highest options in the scale (*four* or *five*) and only 6.3% to the lowest option (*one* or *two*). Therefore, according to the respondents, the lack of experience in AngularJS is a frequent cause of developers building applications with performance problems when using this framework.

The third group of bars, representing the cause "applications following a wrong architecture", also shows a trend to the high part of the scale; however, this is less strong than in the previous case. We can see that 55.8% of the developers gave an answer of *four* or *five* and that 15.8% gave an answer of *one* or *two*. These results reveal that most developers also agree that inadequate architectures are frequently a cause of poor performance. In other words, the problem is not related with the framework but with the ability of the developers to define architectures that allow high performance in their applications.

In the last group of bars we present the opinion of the developers about how often the internal architecture of AngularJS can be blamed for poor performance. The first result worth noting is that this is the group in which the neutral option is the most popular answer with 40%. The second is that there is not a strong trend in either side of the scale (34.8% of the answers are in the low part of the scale and 25.2% in the high part). In summary, we can not conclude that the internal architecture of AngularJS is considered by the developers as a cause of performance problems.

The most clear conclusion that we can draw from this section is that many developers agree that the lack of experience in AngularJS is a very likely reason for which their peers build applications with poor performance. However, this can also be seen as an expected result. It is almost certain that developers who know better the tools with which they work will develop applications with better quality. But, how much do they need to know about AngularJS to avoid performance degradations in their applications? To answer this question in Section 4.2.4 we investigate the relation between poor performance and technical aspects of the framework.

As a final comment, we must state that the findings presented in this section are more based in opinions and not with the experience or practice of the developers. Therefore, bias may have a major effect in the results due to the population that was selected to answer the survey.

4.2.4 Technical Causes of Performance Problems

During the mapping study we also found several recommendations (based on technical knowledge) about practices that should be followed (or avoided) to deal with performance problems in AngularJS [Markiewicz, 2015; Nadel, 2013; Kras, 2015]. From this study, we extracted 13 practices than can lead to performance degradation. A brief description of each practice is presented below.

1. *Unnecessary use of two-way data binding:* Two-way data binding is the solution offered by AngularJS to keep in synchrony the data between the model and the view components. However, every time that a binding of this type is used a new `$watch` expression is added to be checked for changes during the digest cycle. Therefore, the higher the number of bindings of this type, the greater the cost of the digest cycle. The release 1.3 of AngularJS introduces a way to implement one-time data binding, which consists in expressions that are no longer checked for changes once a stable value is assigned to them. Therefore, this feature reduces the number of expressions being watched and makes the digest cycle faster. Unfortunately, sometimes two-way data binding is used when one-time data binding is enough leading to an unnecessary degradation of the application performance.

2. *Use of a wrong watching strategy:* During the digest cycle, the change detection process is performed. The current values of the expressions being watched are compared to the old ones. This comparison can be done in three different ways: by reference, by collection, or by value. If an object is being checked, for example,

a comparison by reference only is able to detect a change when the whole object is replaced by another. When a comparison is made by collection or by value changes made in the nested values of the object can be detected. However, a comparison of this type is expensive since 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 [AngularJS Developer Guide, 2015c]. If the developer misuses the most expensive watching strategies, the performance of the whole application can be negatively affected.

3. *Watching complex functions:* Each scope in an AngularJS application allows the developer to register a function whose returned value will be watched. This registration can be done through the `$watch` method. In each digest cycle the function is executed and the returned value is compared with the value returned by the function in its last execution. If the function is highly complex, its execution will take longer, as well as the digest cycle. If many complex functions are watched, the performance of the application can drop considerably.

4. *Use of filters in templates instead of in JavaScript code:* In AngularJS, filters are functions applied to the value of expressions to format data displayed to the user. They can be used directly in templates with an especial syntax (`{expression | filter}`) or they can be used in controllers, services and directives using JavaScript code. However, when a filter is used in a view template, it is evaluated on every digest cycle, which can be very expensive specially for large data structures. On the other hand, when a filter is used in a controller, it is possible to call the filter only when needed avoiding its execution on each digest cycle.

5. *Use of mouse events directives:* AngularJS provides developers with directives such as `ngClick`, `ngMouseenter`, `ngMouseleave`, etc., that allow the tracking of mouse events. Since the functions that handle these events can change the model data, each time that a user interaction associated with the directive is performed, the digest cycle is triggered for change detection.

6. *Use of a wrong conditional-display strategy:* There are two different types of directives that can be used to display DOM elements depending on a condition. The first type of directive (e.g., `ngHide` and `ngShow`) hides and shows elements by using CSS classes, without removing them. The second type of directive (e.g, `ngIf` and `ngSwitch`) actually creates or destroys DOM elements depending on a condition. The main advantage of the last strategy is that along with the DOM

element are also destroyed all the event handlers and bindings (and watches) associated to it. However, creating and destroying elements are expensive operations. Therefore, the type of directive used should be carefully selected depending on the situation.

7. *Inappropriate use of `$scope.$apply()`:* `$scope.$apply` is the method used by AngularJS to trigger the full digest cycle. It is implicitly called by the framework when there is a possibility of mutation in the model data. However, the operations responsible for these mutations can also occur in places where AngularJS has no control (e.g., in a third party library). For this reason, AngularJS allows the developers to explicitly call the `$scope.$apply` method. An abuse of this method may result in multiple triggers of the digest cycle.

8. *Runtime configurations and flags:* By default AngularJS attaches information (i.e., CSS classes) about bindings and scopes to the DOM elements so that debugging or testing tools can work. However, when the application is in production, it is recommended to disable this feature.

9. *Not cleaning up the application:* This problem arises when some parts of the application use computational resources even when they are no longer needed. This can occur, for example, when there are handlers that will never be triggered because the event emitter has been destroyed. Therefore, it is important to unbind handlers, remove unnecessary watches, and stop asynchronous operations, etc., when these actions are not done automatically. Most components of an AngularJS application have a destroy event that can be listened to destroy other components that depend on them, which can avoid losses in performance.

10. *Use of `ngRepeat` with long lists:* The `ngRepeat` directive is used to create DOM elements for each item in a collection. It is used to render lists and maintain the view updated when their items change. A new scope and a binding is created for each item to track the possible changes that can occur in the collection. When the collection is too large, these bindings can affect the digest cycle and therefore the performance of the application.

11. *Use of `ngRepeat` without "track by":* As mentioned before, the creation and deletion of DOM elements are expensive operations. When the `ngRepeat` directive is used AngularJS assigns a unique hash key to each item of the collection to make an association with its DOM element and avoid those expensive operations. However, when the collection is refreshed from the server this assignment is lost and

all the elements must be re-rendered in the view. To solve this problem, the *track by* expression can be used with the `ngRepeat` directive to specify a unique key to each item to make the identification. If, for example, the identifiers of the objects loaded from the server are used for this purpose, even when the collection is reloaded it is possible to recognize the items that have been rendered before.

12. *Nested directives in `ngRepeat`:* When the items of the collection used in the `ngRepeat` directive are represented by templates that use more directives creating bindings (e.g., `ngShow`), it is easy to create a lot of watches in the digest cycle.

13. *Unconscious triggering of the full digest cycle:* To guarantee the synchronization between the view and the model, AngularJS automatically triggers the digest cycle when operations that can change the data in the model are executed. Some methods that trigger the full digest cycle are `$http`, `$resource`, `$timeout`, `$interval`, `$scope.$apply`, etc. The lack of knowledge of this fact can lead to architect the application in a way in which the digest cycle is triggered multiple times.

To find out the relevance of the aforementioned situations, we asked the developers to rate how often they are responsible for negative effects in the performance of an AngularJS application (questions 22 to 24 in Appendix B.2). For this purpose, the participants were given a scale for each item in which the lowest option (*one*) was labeled with "very rarely" and the highest option (*five*) was labeled with "very often".

Figure 4.6 presents the distribution of the responses for each item. The first observation is that there is no bar reaching 100% of the population. This observation may be due to the request that we made to the participants to avoid answering questions in which they did not have the necessary knowledge to provide an answer. As we can see, the cause with the lowest number of responses was the one addressing the configuration of the application for production mode (70.5% of the population gave an answer). On the other hand, the causes that received the largest number of responses were the ones about watching complex functions and using unnecessary two-way data binding, in which 87.4% of the population participated.

In Figure 4.6, the last three bars show the clearest trend towards the right end of the scale, which means that they affect the performance of AngularJS applications with more frequency. The unnecessary use of two-way data binding was signaled by 31.3% of the developers who gave an answer as a very frequent cause of performance degradation. Another 36.1% rated this cause with *four* in the scale, which represents 67.4% of the responses in the highest part of the scale. Furthermore, only 15.6% of the

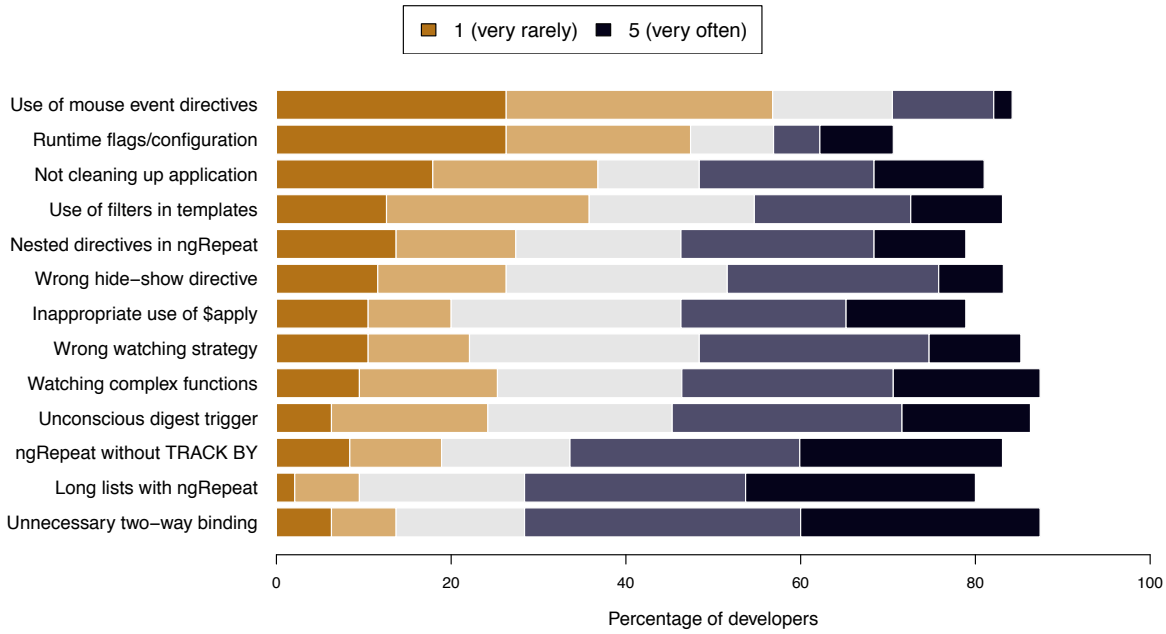


Figure 4.6. Evaluation of technical causes of performance issues in AngularJS

responses are located in the lowest options of the scale for this cause (7.2% in option *one* and 8.4% in option *two*).

This result may be related to two questions from the first section of the questionnaire. In the first one we asked the participants to select the version of AngularJS with the major performance improvements. Removing the participants who answered "I am not sure" for this question (40% of the responses), 63.2% indicated version 1.3. A possible explanation for this result could be that in that version the option to use one-time data binding was introduced in AngularJS. The second question was about refactoring. We asked the developers if they had ever made a refactoring to replace two-way with one-time data binding to improve its performance. From the whole sample, 72.6% of the participants declared to have made at least once a refactoring of this type. This is another clear indicative of the high impact that this design decision has in the performance of an application.

The other two bars with a considerable amount of responses in the high part of the scale are associated to the use of the `ngRepeat` directive. Regarding the use of long lists with the `ngRepeat` directive, 31.6% of the developers marked the option *four* and 32.9% marked the option *five* in the scale. In total, 64.5% of the responses are in the highest part of the scale for this cause. Additionally, the responses in the lowest part of the scale (options *one* or *two*) represent only 11.8% of the total. These results

make evident that most developers think that performance can be deteriorated by rendering long collections with the `ngRepeat` directive. This is not a surprising result given that, as was mentioned in Section 4.2.2, this is a consistent topic addressed in the literature found in our mapping study and it is one of the problems that the AngularJS community has been trying to solve through third-party components. The problem with this directive is that it can be easily used to create complex DOM elements for each item in a collection creating at the same time several bindings (watches) that slow down the completion of the digest cycle. For this reason, most of the solutions proposed to solve this problem include the reduction of the displayed list by using strategies such as pagination or infinite scroll [Arora, 2015; Fröstl, 2013].

Another practice, related with the rendering of long lists, that was also highly rated as a frequent cause of performance degradation is the use of the `ngRepeat` directive without the *track by* expression. The ratings for this cause are shown in the third bar of Figure 4.6. We can see that more than half of the developers selected an option from the high part of the scale for this cause. To be more specific, 31.6% selected the option *four* and 27.8% selected the option *five*. The neutral option was chosen by 17.7% of the developers who gave an answer and the lower options were selected by 22.8% of the participants. This reveals how much attention must be put in the architecture of AngularJS applications when working with large amounts of data representing items that need to be rendered and kept updated during the application execution. We can take as an example the use of a global identifier that allows the direct association of items in the server with the elements rendered in the client. In this case, expensive operations such as creation and deletion of DOM elements can be avoided by using the *track by* expression with the `ngRepeat` directive.

Another point worth mentioning about the results presented in Figure 4.6 is the classification given to the "use of mouse events directives" and the "use of wrong configurations for production" as unusual causes of performance degradation. For both scenarios more than 67% of the developers used options *one* or *two* to rate the frequency with which they were the problem behind the poor performance of an AngularJS application. In the case of mouse-events directives the reason for these results may be that using these directives by themselves does not affect the performance of the application. The problem appears only when they are spread all over the application creating multiple potential triggers of the entire digest cycle.

Disabling debug information is also an action that is not seen by the respondents as an important performance booster. The addition of information (scopes and CSS classes) to the DOM elements is mostly perceived while the application is being loaded for the first time. This optimization has little or no effect in the digest cycle and for

this reason can be considered by the respondents as having a minor improvement in the performance. During our mapping study we also found developers saying that no improvements in performance are perceived after disabling debug data in production [Stack Overflow, 2015a].

Finally, from Figure 4.6 we can declare that almost all of the technical causes considered for the construction of this survey are recognized by the developers as relevant in the deterioration of performance. With the exception of the two causes represented by the two bars in the top, at least 34.2% of the developers rated all the proposed causes with *four* or *five*. Maybe this is because all of them are related with three important programming practices that have to be kept into account to avoid problems of performance: (1) keep the digest cycle as light as possible by avoiding unnecessary and complex watches, (2) avoid multiple triggers of the digest cycle and (3) reduce as much as possible the number of expensive interactions with the DOM API.

4.3 Threats to Validity

The first threat to validity is related to our mapping study since it is not possible to guarantee that the sources reviewed include all of the available information about performance in AngularJS. In other words, information not collected during our mapping study could modify the construction of our survey leading to different results.

In the construction of a survey there is always the risk of having ambiguous or leading questions. However, we made our best to avoid this type of questions by using multiple iterations of small reviews and corrections of the proposed questions. We also used the pilot phase to refine the wording of the questions. Additionally, it is important to note that we did not use mandatory questions whenever we thought that some kind of special knowledge was necessary to avoid forcing responses.

There are also some aspects that limit the generalization of our results (external validity). First, the subject population was selected from specif on-line communities and it is not possible to make sure that the findings also apply to populations not participating in these communities. Additionally, due to the constant and rapid change in the Web development environment, it is possible to obtain different results if the study is repeated in the future. As an example, we can mention that while this study was being conducted a new release of the framework (1.5) was being developed and also a new version (2.0) for which the results could no longer apply.

Finally, we have to mention the threats to validity related with social aspects. For example, the ordering of the questions could affect the answers of the respondents.

The responses can also be affected, either positively or negatively, by the attitudes of the developers to the research topic or their tendency to look better when they are interrogated. However, this last threat may be alleviated by the anonymous character of the responses.

4.4 Final Remarks

This chapter reported an empirical study about AngularJS performance supported by a survey. We started by describing the design of the study, which was divided in two phases. The first phase was a *Mapping Study* (Section 4.1.1) in which we gathered information about the state-of-the-art of AngularJS performance by reviewing informal literature that was found mainly on the Internet. The second phase was the design and execution of a survey that we described in details in Section 4.1.2. More specifically, the topics addressed in the survey were (a) background of the users, (b) development practices and perceptions regarding AngularJS performance, (c) general causes of poor performance in AngularJS applications and (d) technical causes of this problem.

The results of the survey were presented in Section 4.2 where we discovered, for example, that many developers (45.3%) reported to have inspected the code of the framework to fix performance issues. We also concluded that the most frequent causes (from a general point of view) of poor performance are the lack of knowledge in AngularJS and applications following a wrong architecture. In Section 4.2.4, we analyzed the technical errors made by developers that can lead to performance degradation. Based on the results, the most frequent causes of performance degradation in AngularJS applications are related with the unnecessary use of two-way data binding and the use of the `ngRepeat` directive with long lists or without the *track by* expression. On the other hand, we also concluded that using mouse-events directives or not using the adequate configurations for production are not considered as major threats to performance by the developers. Finally, in Section 4.3 we discussed the threats to validity of this study which are mainly concerned with the sample selected, the social behavior presented by the respondents, and the construction of the survey instrument.

Chapter 5

Conclusion

AngularJS is one of the most popular JavaScript frameworks that implement a variation of the MVC pattern for the construction of Web applications. It is highly used to deal with the challenges presented mainly in the construction of Single Page Applications (SPAs). However, there is not a clear understanding about how this framework affects the development process. More specifically, what are the most relevant characteristics of the framework and what are the main problems faced by the developers that use it. To address this shortcoming, we made an empirical study by using two surveys built on information collected from the review of blogs and Q&A sites in the Internet. The first survey collects developer's perceptions about different characteristics of the framework, including usefulness of its features, testability, template system, complexity of use, among others. The second survey, on the other hand, focuses on the performance of the framework.

We received 460 responses from the first survey and 95 from the second one. After the analysis of these results (Sections 3.2 and 4.2), we reported several findings about the development experience provided by AngularJS. The main ones are listed in Section 5.1. Additionally, we discuss how this research may be complemented by presenting some future work in Section 5.3.

5.1 Main Findings

The following items represent the main findings of this research.

- There are three characteristics of AngularJS that excel by the value that developers give to them: the ability to create custom UI components by means of

custom directives, the use of dependency injection, and the ease to set two-way data binding avoiding boilerplate code.

- The two problems that arise more frequently regarding the use of code in AngularJS templates are the emergence of silent failures and the difficult to dump/debug the variables referenced in the HTML.

- The two main reasons for which developers place large amounts of logic in templates is the lack of experience in using AngularJS to architect Web applications and the design of AngularJS, which fosters this practice. Additionally, many developers declared that they had followed this practice for other reasons such as speed, easiness, prototyping or special cases.

- The components more difficult to test in an AngularJS application are the directives. Specially, the ones using transclusion. The remaining components (i.e., controllers, services, filters, etc.) are considered easy to test for at least 60% of the developers.

- Although the creation of custom directives is one of the most appreciated characteristics of AngularJS, the API that supports this feature is perceived as complex to use and understand by the developers.

- Tackling all the details to avoid performance degradation is considered as one of the most complex characteristics of the framework.

- Many developers (45.3%) have inspected the source code of AngularJS to fix performance problems.

- The most frequent causes (from a general point of view) of poor performance in AngularJS applications are the lack of knowledge in AngularJS and applications following a wrong architecture.

- From a technical point of view, the most frequent causes of applications with poor performance are related to the unnecessary use of two-way data binding and the use of the `ngRepeat` directive with long lists or without the *track by* expression.

5.2 Contributions

The findings reported in this work are useful to understand how the development experience of Web applications is affected by the architectural characteristics of AngularJS.

This knowledge represents an important contribution for two different fields: the development of client-side MVC frameworks and the academic research concerned with Web applications development.

From the perspective of developers of client-side MVC frameworks, it is possible to find in this study different lessons provided by users of the most popular framework for the construction of better ones. These new frameworks can improve characteristics such as performance and usability considering the value and the problems that some features and architectural decisions of AngularJS have brought to the development experience.

During the construction of client-side MVC frameworks, there are many important decisions that have to be made about the architecture of the framework. These decisions are hard to make because they have advantages and disadvantages in the development of applications that are hard to change once that they are implemented. The findings reported in this study can help in this decision-making process since it is possible to prioritize the most valued characteristics and give up the least valued ones to improve the framework as a whole. As an example of valuable characteristics that can be improved in AngularJS, we can mention the ability to build customized and reusable UI components (directives) and the constant synchronization between the model and the view (two-way data binding). New frameworks could embrace these characteristics and improve them by offering a more simple API to build reusable UI components (i.e., without exposing the final users to many internal concepts and decisions), and by using a better mechanism to detect changes in the model to maintain the synchrony between the model and the view (i.e., a mechanism that reduces the number of details that have to be considered to avoid performance problems).

Additionally, common challenges faced by AngularJS developers can be identified and used as inspiration to build new development tools and implement new features that help to deal with these challenges.

From an academic perspective, the knowledge generated in this work is an important base to begin more specific research about different aspects that constitute the development of single-page applications. There are many architectural decisions that have to be made in the development of single-page applications, and all of them can be individually and further investigated in the Computer Science area. Topics such as template systems, data binding, testing, modularization, dependency management, and maintainability can be investigated by using our findings in a greater or lesser extent as a starting point.

Additionally, today there is a large amount of emerging technologies for the construction of Web applications (including client-side MVC frameworks) and it is harder

and harder for Web developers to make decisions about what technology to use. The problem with this excess of frameworks is that many of them propose very similar solutions for the same problems adding complexity to the decision process and with little or no value to the current state of single-page applications development. Our research can contribute in the search of convergent architectures that balance the trade-offs between the different design decisions. These new architectures can become standard technologies for the development of Web applications. As an example, we can mention the rise of technologies such as Web Components, which propose reusable user interface widgets that belong to the browser and do not need external libraries to be used [Mozilla Developer Network, 2016]. The evidence presented in this study shows, for example, that the standardization of this technology (which is still experimental) would be highly appreciated by the community developing Web applications and that it is worth investing efforts to improve it.

5.3 Future Work

This research may be complemented with the following future work:

- *Interviews:* Interviews with a set of AngularJS developers could help us to better understand the findings of this research. For example, they could provide additional information about how the characteristics of the framework are better (or worst) with determined types of applications. This examination in detail could help us gather information that is hard to capture using a questionnaire.

- *Surveys with other frameworks:* Similar studies with other highly-used client-side MVC frameworks such as Ember.js or Backbone.js could help to understand how the development of Web applications is affected by characteristics that AngularJS does not have. Moreover, it would allow the comparison between the framework features and the trade-off of implementing one or another.

- *Analysis of Stack Overflow data:* An analysis of the questions and answers posted in Stack Overflow by AngularJS developers could help us compare the findings reported in this research, which is mainly based on opinions and perceptions, with a source of data that contains quantifiable evidence of the main problems faced in development. Again, this approach could be extended to other client-side MVC frameworks for comparison purposes.

- *Analysis considering experience:* All the findings reported in this study are based in the analysis of the responses provided by the whole population. However, com-

plementary conclusions could be drawn by classifying and analyzing the responses according to the experience of the respondents.

- *Analysis of AngularJS code:* Since the findings reported in this study are based mainly on the perceptions of the respondents, it would be interesting to analyze AngularJS code to compare the findings reported in this study to the real practices of AngularJS developers. For example, we could check if common practices reported in this study (such as the frequent use of customized directives or the difficulty to test certain parts of applications) are in fact followed in real projects. This type of verifications could be also performed by doing historic analysis of the code to check the evolution of AngularJS products.

- *Performance study:* The findings reported in this study regarding AngularJS performance are useful to verify how often AngularJS developers face performance problems, how they deal with these problems, and how much the architecture of the framework have to be considered to avoid this type of problems (technical causes). However, it is not possible to measure the real performance of the framework by means of a survey. Therefore, an interesting future work could be to set up a controlled experiment to study quantitatively the performance of the framework. This study would even allow the performance comparison between different frameworks and their architectures.

- *Framework construction:* The findings made in this research could be used to build a new framework focused on keeping the most appreciated features of AngularJS and improving the ones that represent a problem in Web application development.

Bibliography

Angular Blog (2015a). Angular 2 Beta. http://angularjs.blogspot.com.br/2015/12/angular-2-beta.html. [Online; accessed December-2015].

Angular Blog (2015b). Angular 2 Survey Results. http://angularjs.blogspot.com.br/2015/09/angular-2-survey-results.html. [Online; accessed October-2015].

AngularJS Developer Guide (2010). What is Angular? https://docs.angularjs.org/guide/introduction. [Online; accessed January-2016].

AngularJS Developer Guide (2015a). Angular Expressions. https://docs.angularjs.org/guide/expression. [Online; accessed July-2015].

AngularJS Developer Guide (2015b). Modules. https://docs.angularjs.org/guide/module. [Online; accessed January-2016].

AngularJS Developer Guide (2015c). Scope $watch Performance Considerations. https://docs.angularjs.org/guide/scope. [Online; accessed July-2015].

AngularJS Developer Guide (2015d). Templates. https://docs.angularjs.org/guide/templates. [Online; accessed January-2016].

AngularJS Developer Guide (2015e). Unit Testing. https://docs.angularjs.org/guide/unit-testing. [Online; accessed July-2015].

Arora, C. (2015). AngularJS Performance. https://www.packtpub.com/books/content/angularjs-performance. [Online; accessed July-2015].

Aston, B. (2014). An Intervention: Why AngularJS is Worse Than a New ASP.NET WebForms. https://medium.com/@benastontweet/an-intervention-4535d835e836#.w8wfz78sm. [Online; accessed June-2015].

Bijlsma, D. and Ghijsen, M. (2014). AngularJS: Keeping up with the forefront of web applications. https://www.sig.eu/files/8014/1872/3508/27-Technology_Risk_Review_AngularJS.pdf. [Online; accessed July-2015].

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). {Pattern-oriented Software Architecture}. 1:125–143.

Crockford, D. (2008). *JavaScript: The Good Parts*. "O'Reilly Media, Inc.".

Czerwinksi, S. (2013). Optimizing AngularJS: 1200ms to 35ms. `http://blog.scalyr.com/2013/10/angularjs-1200ms-to-35ms/`. [Online; accessed June-2015].

Eidnes, L. (2014). AngularJS: The Bad Parts. `http://larseidnes.com/2014/11/05/angularjs-the-bad-parts/`. [Online; accessed June-2015].

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 116--125. IEEE.

Flanagan, D. (2006). *JavaScript: the definitive guide*. "O'Reilly Media, Inc.".

Fowler, M. (2001). Separating user interface code. *Software, IEEE*, 18(2):96--97.

Fröstl, S. (2013). AngularJS Performance Tuning for Long Lists. `http://tech.small-improvements.com/2013/09/10/angularjs-performance-with-large-lists/`. [Online; accessed June-2015].

Garrett, J. J. (2005). Ajax: A new approach to web applications.

Gechev, M. (2013). AngularJS Style Guide. `https://mgechev.github.io/angularjs-style-guide/#templates`. [Online; accessed July-2015].

Gousios, G., Zaidman, A., Storey, M.-A., and Deursen, A. V. (2015). Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358--368. IEEE Press.

Hall, W. and Tiropanis, T. (2012). Web evolution and web science. *Computer Networks*, 56(18):3859--3865.

Hamilton, C. (2014). Angular 2.0 Announcement Backfires. `https://jaxenter.com/angular-2-0-announcement-backfires-112127.html`. [Online; accessed June-2015].

Hendrickson, E. and Fowler, M. (2002). The software engineering of internet software. *IEEE software*, 19(2):23.

Hooker, C. (2014). Unit Testing AngularJS Directives With External Templates. http://www.bluesphereinc.com/blog/unit-testing-angularjs-directives. [Online; accessed July-2015].

Horie, L. (2014). Lessons Learned From AngularJS. http://lhorie.github.io/mithril-blog/lessons-learned-from-angular.html. [Online; accessed June-2015].

Kienle, H. M. (2010). It's about time to take javascript (more) seriously. *Software, IEEE*, 27(3):60--62.

Knight, A. and Dai, N. (2002). Objects and the web. *IEEE software*, 19(2):51.

Koch, P. (2015). The problem with Angular. http://www.quirksmode.org/blog/archives/2015/01/the_problem_wit.html. [Online; accessed June-2015].

Koshelko, E. (2015). Why You should not use AngularJS. https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99. [Online; accessed June-2015].

Kransner, G. E. and Pope, S. T. (1988). Cookbook for using the model-view-controller user interface paradigm. *Journal of Object Oriented Programming*, pages 26--49.

Kras, A. (2015). 11 Tips to Improve AngularJS Performance. http://www.alexkras.com/11-tips-to-improve-angularjs-performance/. [Online; accessed July-2015].

Krivanov, I. (2015). Creating Angular 2 Style Components Using Angular 1. http://blog.rangle.io/angular2-components/. [Online; accessed October-2015].

Leff, A. and Rayfield, J. T. (2001). Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118--127. IEEE.

Markiewicz, R. (2015). Angular non-trivial performance hints. http://codetunes.com/2015/angular-non-trivial-performance-hints/. [Online; accessed July-2015].

Mesbah, A. and Deursen, A. V. (2007a). An architectural style for ajax. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*, pages 9--9. IEEE.

Mesbah, A. and Deursen, A. V. (2007b). Migrating multi-page web applications to single-page ajax interfaces. In *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, pages 181--190. IEEE.

Mozilla Developer Network (2016). Web Components. `https://developer.mozilla.org/en-US/docs/Web/Web_Components`. [Online; accessed February-2016].

Mustafa, K. (2014). Angular performance in ng-repeat. `http://stackoverflow.com/q/22937504/5244036`. [Online; accessed July-2015].

Nadel, B. (2013). How Often Do Filters Execute In AngularJS. `http://www.bennadel.com/blog/2489-how-often-do-filters-execute-in-angularjs.htm`. [Online; accessed June-2015].

Nederlof, A., Mesbah, A., and van Deursen, A. (2014). Software engineering for the web: the state of the practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 4--13. ACM.

Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., Nguyen, A. T., and Nguyen, T. N. (2012). Detection of embedded code smells in dynamic web applications. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 282--285. IEEE.

Ocariza, F., Bajaj, K., Pattabiraman, K., and Mesbah, A. (2013). An empirical study of client-side javascript bugs. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 55--64. IEEE.

Ocariza, F., Pattabiraman, K., and Mesbah, A. (2015). Detecting inconsistencies in javascript mvc applications. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 325--335. IEEE Press.

Osmani, A. (2012). *Learning JavaScript design patterns*. "O'Reilly Media, Inc.".

Osmani, A., Hartig, P., and Sorhus, S. (2011). TodoMVC Project. `http://todomvc.com/`. [Online; accessed January-2016].

Parviainen, T. (2015). A Guide To Transclusion in AngularJS. `http://teropa.info/blog/2015/06/09/transclusion.html`. [Online; accessed July-2015].

Petersen, J. (2014). How Google Broke the OSS Compact With Angular 2.0. `http://codebetter.com/johnvpetersen/2014/10/27/how-google-broke-the-oss-compact-with-angular-2-0/`. [Online; accessed June-2015].

Potel, M. (1996). Mvp: Model-view-presenter the taligent programming model for c++ and java. *Taligent Inc*, page 20.

Reddit (2015). How to not Let Angular Spoil All Your HTML With Logic Code? `https://www.reddit.com/r/angularjs/comments/2b15dv/how_to_not_let_angular_spoil_all_your_html_with/`. [Online; accessed July-2015].

Skoglund, M. and Runeson, P. (2009). Reference-based search strategies in systematic reviews. In *The Proceedings of the 13th International Conference on Evaluation and Assessment in Software Engineering, Durham, England*.

Smith, J. (2009). Patterns-wpf apps with the model-view-viewmodel design pattern. *MSDN magazine*, page 72.

Stack Overflow (2013a). How to improve performance of ngRepeat over a huge dataset (angular.js)? `http://stackoverflow.com/q/17348058/5244036`. [Online; accessed July-2015].

Stack Overflow (2013b). Using scope.*watchandscope*.apply in AngularJS. `http://stackoverflow.com/q/15112584`. [Online; accessed July-2015].

Stack Overflow (2013c). What does *scope*.apply() do? `http://stackoverflow.com/q/18710478`. [Online; accessed July-2015].

Stack Overflow (2014). Why don't AngularJS errors in the html show up in the console? `http://stackoverflow.com/q/26227596/5244036`. [Online; accessed July-2015].

Stack Overflow (2015a). Angularjs disabling debug data in production. `http://stackoverflow.com/q/32967438`. [Online; accessed July-2015].

Stack Overflow (2015b). AngularJS: Reducing Logic in HTML Templates. `http://stackoverflow.com/q/27217397/5244036`. [Online; accessed July-2015].

Takhteyev, Y. (2015). Write Your Angular 1.x Applications in an Angular 2.0 Style. `http://blog.rangle.io/write-angular-2-style-code-now-typescript-decorators-components-and-flux-for-angular-1-x-applications/`. [Online; accessed July-2015].

W3C (2005). Document Object Model (DOM). `https://www.w3.org/DOM/`. [Online; accessed February-2016].

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Appendix A

Mapping Study Documents

In this appendix, we present the documents generated as a result of the two mapping studies (Section 3.1.1 and Section 4.1.1). Each document contains a list of the most relevant quotations found in the different sources of information. The listed quotations use an identifier to reference the source from which they were taken. In the first sections, we present the considered sources of information with their respective identifiers. These identifiers use the prefix *"BLOG"* to indicate that the source is a blog, *"Q&A"* for communities of questions and answers, and *"DOC"* for AngularJS official documentation.

A.1 Document for Survey About General Aspects of AngularJS

A.1.1 Sources of Information

1.

Identifier: DOC1
Title: Developer Guide: Unit Testing
Access date: September-2015
URL: `https://docs.angularjs.org/guide/unit-testing`

2.

Identifier: DOC2
Title: API Reference: `$compile`
Access date: September-2015
URL: `https://docs.angularjs.org/api/ng/service/$compile`

3.

Identifier: BLOG1
Title: Choosing a JavaScript MVC Framework
Access date: June-2015
URL: `http://www.funnyant.com/choosing-javascript-mvc-framework/`
Author: Craig McKeachie
Author information: Microsoft Certified Solutions Developer.

4.

Identifier: BLOG2
Title: AngularJS vs. Backbone.js vs. Ember.js
Access date: August-2015
URL: `https://www.airpair.com/js/javascript-framework-comparison`
Author: Uri Shaked
Author information: The author works for WatchDox, lectures at the Israel Institute of Technology, and organizes the Tel Aviv Google Developers Group.

5.

Identifier: BLOG3
Title: Unit Testing AngularJS Directives With External Templates
Access date: September-2015
URL: `http://www.bluesphereinc.com/blog/unit-testing-angularjs-directives`
Author: Chris Hooker
Author information: The author is a graduate of MIT, and has over 15 years of software development experience.

6.

Identifier: BLOG4
Title: What's wrong with Angular
Access date: September-2015
URL: `http://www.quirksmode.org/blog/archives/2015/01/the_problem_wit.html`
Author: Daniel Steigerwald
Author information: JavaScript Developer and Consultant. Member of the Google Developers Expert Program.

7.

Identifier: BLOG5
Title: Optimizing AngularJS: 1200ms to 35ms
Access date: August-2015
URL: `http://blog.scalyr.com/2013/10/angularjs-1200ms-to-35ms/`
Author: Steven Czerwinksi
Author information: Staff Software Engineer at Scalyr.

8.

Identifier: BLOG6
Title: Speeding Up AngularJS Apps With Simple Optimizations
Access date: August-2015
URL: `http://www.binpress.com/tutorial/speeding-up-angular-js-with- simple-optimizations/135`
Author: Todd Motto
Author information: Director of Web Development at Mozio. He works with JavaScript and AngularJS.

9.

Identifier: BLOG7
Title: AngularJS: My Solution to the `ng-repeat` Performance Problem
Access date: July-2015
URL: `http://www.williambrownstreet.net/blog/2013/07/angularjs-my-solution-to-the-ng-repeat-performance-problem/`
Author: Thierry Nicola
Author information: Computer Scientist, working with Grails, AngularJS, BackboneJS, and ChaplinJS.

10.

Identifier: BLOG8
Title: AngularJS Performance in Large Applications
Access date: August-2015
URL: `https://www.airpair.com/angularjs/posts/angularjs-performance-large-applications`
Author: Abraham Polishchuk
Author information: NodeJS/AngularJS engineer at Lanetix.

11.

Identifier: BLOG9
Title: Why You Should Not Use Angularjs
Access date: June-2015
URL: `https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs- 1df5ddf6fc99`
Author: Egor Koshelko
Author information: Fullstack NodeJS Developer.

12.

Identifier: BLOG10
Title: AngularJS Critique
Access date: October-2015
URL: `http://tutorials.jenkov.com/angularjs/critique.html`
Author: Jakob Jenkov
Author information: Founder and CEO at Jenkov Aps.

13.

Identifier: BLOG11
Title: Debugging AngularJS
Access date: October-2015
URL: `https://www.ng-book.com/p/Debugging-AngularJS/`
Author: Ari Lerner
Author information: Developer with more than 20 years of experience, and co-founder of Fullstack.io. The author has been using AngularJS for a long time and is a recognized expert in the field.

14.

Identifier: BLOG12
Title: A Guide To Transclusion in AngularJS
Access date: October-2015
URL: `http://teropa.info/blog/2015/06/09/transclusion.html`
Author: Tero Parviainen

15.

Identifier: BLOG13
Title: An Intervention: Why AngularJS is Worse Than a New ASP.NET WebForms
Access date: October-2015

URL: https://medium.com/@benastontweet/an-intervention-4535d835e836
Author: Ben Aston
Author information: Consultant developer specializing in web applications.

16.

Identifier: BLOG14
Title: Scope Creep, a Deep Dive Into Angular's Scope
Access date: October-2015
URL: http://jonathancreamer.com/working-with-all-the-different-kinds-of-scopes-in-angular/
Author: Jonathan Creamer
Author information: JavaScript, Ruby, C#, Node, Rails, .NET MVC, MS MVP, Telerik Developer Expert, and IEuserAgent.

17.

Identifier: BLOG15
Title: Adding Clarity to Scope Inheritance in Angular
Access date: October-2015
URL: http://jonathancreamer.com/adding-clarity-to-scope-inheritance-in-angular/
Author: Jonathan Creamer
Author information: JavaScript, Ruby, C#, Node, Rails, .NET MVC. MS MVP, Telerik Developer Expert, and IEuserAgent.

18.

Identifier: BLOG16
Title: Mastering AngularJS directives
Access date: October-2015
URL: http://code.tutsplus.com/tutorials/mastering-angularjs-directives--cms-22511
Author: Hüseyin Babal
Author information: NodeJS, PHP, Java, Elasticsearch, WordPress Plug-in/Widget Development, MySQL, MongoDB, SEO, Agile Software Development, Cloud Integration, SCRUM.CSM.

19.

Identifier: BLOG17
Title: Creating Custom AngularJS Directives Part I - The Fundamentals
Access date: September-2015
URL: `http://weblogs.asp.net/dwahlin/creating-custom-angularjs-directives-part-i-the-fundamentals`
Author: Dan Wahlin
Author information: JavaScript/ES6, Node.js, Angular, HTML5, jQuery, Node.js, ASP.NET MVC, C#.

20.

Identifier: BLOG18
Title: Directive Controller And Link Timing In AngularJS
Access date: September-2015
URL: `http://www.bennadel.com/blog/2603-directive-controller-and-link-timing-in-angularjs.htm`
Author: Ben Nadel
Author information: Adobe Community Expert as well as an Adobe Certified Professional in Advanced ColdFusion.

21.

Identifier: Q&A1
Title: Template Type Conversion Errors are Difficult to Debug
Access date: September-2015
URL: `https://github.com/angular/angular.js/issues/1974`

22.

Identifier: Q&A2
Title: Why Don't AngularJS Errors in the HTML Show Up in the Console?
Access date: September-2015
URL: `http://stackoverflow.com/q/26227596`

23.

Identifier: Q&A3
Title: Separating JS Logic From Angular Templates
Access date: October-2015
URL: `https://groups.google.com/forum/#!topic/angular/pHSMrphDJhM`

24.

Identifier: Q&A4
Title: AngularJS Use of Inline JavaScript in HTML Attributes is not Bad Practice?
Access date: October-2015
URL: `http://stackoverflow.com/q/14793692`

25.

Identifier: Q&A5
Title: AngularJS: Is `ngClick` a Good Practice Why is There no `ng-event` in AngularJS?
Access date: September-2015
URL: `http://stackoverflow.com/q/14346073`

26.

Identifier: Q&A6
Title: How to not Let Angular Spoil All Your HTML With Logic Code?
Access date: October-2015
URL: `http://www.reddit.com/r/angularjs/comments/2b15dv/how_to_not_let_angular_spoil_all_your_html_with/`

27.

Identifier: Q&A7
Title: Should AngularJS Logic Be Placed in HTML File?
Access date: September-2015
URL: `http://stackoverflow.com/a/24845815`

28.

Identifier: Q&A8
Title: Using `scope.$watch` and `scope.$apply` in AngularJS
Access date: September-2015
URL: `http://stackoverflow.com/q/15112584`

29.

Identifier: Q&A9
Title: What Does `$scope.$apply()` Do?
Access date: September-2015
URL: `http://stackoverflow.com/q/18710478`
Author: Dan Prince

30.

Identifier: Q&A10
Title: Correct Way to Integrate jQuery Plugins in AngularJS
Access date: October-2015
URL: `http://stackoverflow.com/q/16935095`

31.

Identifier: Q&A11
Title: Difference Between the 'controller', 'link' and 'compile' Functions When Defining a Directive
Access date: October-2015
URL: `http://stackoverflow.com/q/12546945`

A.1.2 Relevant Quotations

- *DOC1:* One of the most useful parts of `ngMock` is `$httpBackend`, which lets us mock XHR requests in tests, and return sample data instead.

- *DOC1:* Angular is written with testability in mind, but it still requires that you do the right thing. We tried to make the right thing easy, but if you ignore these guidelines you may end up with an untestable application.

- *DOC1:* Angular also provides the `ngMock` module, which provides mocking for your tests. This is used to inject and mock Angular services within unit tests. In addition, it is able to extend other modules so they are synchronous. Having tests synchronous keeps them much cleaner and easier to work with.

- *DOC2:* There are many different options for a directive.

- *DOC2:* The 'isolate' scope object hash defines a set of local scope properties derived from attributes on the directive's element. These local properties are useful for aliasing values for templates. The keys in the object hash map to the name of the property on the isolate scope; the values define how the property is bound to the parent scope, via matching attributes on the directive's element

- *BLOG1:* With Angular and Ember you usually have to live with the choices made by the authors of the frameworks, which may or may not suit your project needs and personal style.

- *BLOG1:* It's important to understand how big a download of each of these frameworks is and what you are getting for that extra weight in your application.
- *BLOG2:* Angular helps you categorize your application building blocks into several types: Controllers, Directives, Factories, Filters, Services and Views (templates).
- *BLOG2:* The Digest Cycle of angular, which takes care of the "Magical" dirty checking, has the tendency to surprise developers. It is easy to forget to call `$digest()` when running in non-Angular context.
- *BLOG2:* Putting logic inside the templates makes it harder to test, as it becomes impossible to test it in isolation.
- *BLOG2:* Mistakes such as misspelling a directive name or calling an undefined scope function are silently ignored and can be challenging to find.
- *BLOG2:* Failing to understand scope inheritance causes many cases of frustrated developers.
- *BLOG2:* Wrapping your head around all the concepts such as compiling function, pre/post linking functions, the different scope kinds (transclusion / isolate / child scope) and all the other configuration settings for directives takes some time to master.
- *BLOG2:* Promises play a main role in the Angular cast.
- *BLOG3:* I hate including HTML inline as a string in Javascript, so I definitely wanted to use `templateUrl`. This was all fine and good, and I got my directive up and running. Then I decided it was time to unit test it. That's where the real fun began.
- *BLOG3:* I discovered that while getting some simple unit tests running wasn't too hard using just Jasmine, unit testing my `templateUrl` directive was no simple matter.
- *BLOG3:* The problem with using `templateUrl` is that Angular uses an HTTP request to go get the file. However, in a unit-testing environment, you don't have the full web server environment and can't actually make the HTTP request. So, you'll get an error when you try to test the directive.

- *BLOG3:* You want to pre-process your HTML template and convert in into Javascript, which can be testing without any need for HTTP requests.
- *BLOG4:* Dirty checking, accessors (Ember and Backbone), `Object.observe` and all that stuff. Wrong! It's slow and brittle and it will consume mobile battery like hungry dog, for no reason.
- *BLOG4:* Angular is HTML parser. I really don't want to debug any string based parser instead of my code.
- *BLOG5:* This shows the power of AngularJS for encapsulation and separation of concerns.
- *BLOG6:* You'll get an error thrown from Angular if you're calling `$scope.$apply` in the "wrong" place, usually too high up the call stack.
- *BLOG7:* I started with BackboneJS as frontend MVC, but soon switch to AngularJS because of the concept AngularJS adds to HTML.
- *BLOG7:* AngularJS offers just the right structure for code organization. After some effort to learn to use directives and services, AngularJS forces me to get my code clean.
- *BLOG8:* Isolate Scope and Transclusion are some of the most exciting things about Angular. They allow the building of reusable, encapsulated components, they are syntactically and conceptually elegant and a core part of what makes Angular Angular.
- *BLOG9:* Errors in bindings don't fire at all.
- *BLOG9:* You can't put a breakpoint inside {{ this expression }}.
- *BLOG9:* When you write in AngularJS you put your logic into your HTML (`ng-repeat`, `ng-show`, `ng-class`, `ng-model`, `ng-init`, `ng-click`, `ng-switch`, `ng-if`). Existence of such logic is not as bad as the fact that it is impossible to test this logic with unit tests, this logic can't be debugged and errors don't fire from markup (but this code contains very important logic).
- *BLOG9:* Errors that occurred in JavaScript are caught by the internal angular interceptor, and interpreted by browser as caught errors (everything that happens in AngularJS, stays in AngularJS).

- *BLOG9 (comment):* The null reference exceptions (in templates (HTML)) are swallowed in order to not break the first renders working properly.
- *BLOG9:* It's unclear why it was necessary to introduce several ways to do the same thing.
- *BLOG9:* It is without a doubt the most common error that absolutely every AngularJS developer faces (Scope inheritance).
- *BLOG9:* There is no logical reason to separate logic for 3 methods (`compile`, `link`, `controller`), all this can be easily implemented in a single method.
- *BLOG9:* Even in order to integrate some code in the angular world, for example some jQuery plugin, you need to wrap it in a directive.
- *BLOG9:* Since server side rendering adds logic into your HTML and AngularJS writes logic in HTML, there is no clear separation of concerns and as a result you get spaghetti code.
- *BLOG9:* The only good thing that AngularJS has is that it forces developers to break their logic into modules, and code becomes more granulated.
- *BLOG10:* Before AngularJS it was "best practice" to keep function calls out of the HTML. For instance, you should not use the `onclick` event attributes on HTML elements, but rather attach event listeners via JavaScript. Somehow that was forgotten with AngularJS, and now we are back to embedding JavaScript function calls in the HTML.
- *BLOG10:* In order to "teach HTML new tricks" you end up with HTML full of non-HTML elements and attributes.
- *BLOG10:* We are back to embedding JavaScript function calls in the HTML.
- *BLOG11:* Angular Batarang is a Chrome extension developed by the Angular team at Google that integrates very nicely as a debugging tool for Angular apps.
- *BLOG12:* Based on what I've heard, I'm not alone in this. Transclusion is one of those things people often mention when they talk about their difficulties with Angular.
- *BLOG12:* I think the bigger problems (with transclusion) are tangential:
  - The API is tricky, with its higher-order functions and "magic arguments".

  - Parts of the API are deprecated, and it can be hard to figure out which parts.
  - The documentation uses lots of big words ("a transclude linking function pre-bound to the correct transclusion scope").
  - There are actually two separate features provided by the same API: Regular transclusion and element transclusion.

- *BLOG13:* AngularJS brings with it a domain specific language: transclusion, scope, directives, modules, factories and services.
- *BLOG13:* You could probably sum up AngularJS with this single word: it makes the inclusion of DOM fragments in your page sound novel and complicated.
- *BLOG14:* Getting used to the concept of scope in Angular is among the more difficult concepts to fully grok when first being introduced to the magical land of Angular.
- *BLOG14:* Because of the fact that the scopes do inherit from other scopes, if you create a primitive value (string, number, boolean) on a parent scope, the child scope will have an be able to manipulate the value.
- *BLOG15:* The fact is prototypical inheritance in JavaScript presents confusion to a lot of folks out there. In terms of Angular.js that may explain why the concept of `$scope` is difficult to grok.
- *BLOG16:* Directives are one of the most powerful components of AngularJS, helping you extend basic HTML elements/attributes and create reusable and testable code.
- *BLOG16:* So, what is the link function there? Simply, the link function is the function that you can use to perform directive-specific operations. The directive is not only rendering some HTML code by providing some inputs. You can also bind functions to the directive element, call a service and update the directive value, get directive attributes if it is an `E` type directive, etc.
- *BLOG16:* Every directive has its own scope, but you need to be careful about the data binding with the directive declaration.
- *BLOG16:* The main advantage of the directive is that it's a reusable component that can be used easily.

- *BLOG16:* When you use directives inside the template, what you see on the page is the compiled version of the directive. Sometimes, you want to see the actual directive usage for debugging purposes. In order to see the uncompiled version of the current section, you can use `ng-non-bindable`.

- *BLOG17:* AngularJS directives can be a bit intimidating the first time you see them. They offer many different options, have a few cryptic features (and cryptic is my politically correct term for "what were they thinking here?"), and are generally challenging at first.

- *BLOG17:* In addition to performing data binding operations with templates, directives can also be used to manipulate the DOM. This is done using the link function shown earlier.

- *BLOG18:* I've talked about the timing of directives in AngularJS a few times before. But, it's a rather complicated topic, so I don't mind digging a bit deeper.

- *BLOG18:* As the DOM (Document Object Model) is compiled by AngularJS, the directive controllers and link functions execute at different parts of the compile lifecycle.

- *BLOG18:* This is an important difference. While you can only access the DOM tree in the bottom-up linking phase, the directive controller can provide a hook into the top-down lifecycle. This can be critical if you have to handle DOM events based on when elements of the DOM tree came into existence. The linking phase can never give you that because it's executed in reverse.

- *Q&A1:* Template type conversion errors are difficult to debug.

- *Q&A1:* FYI the only line in the stack trace that references my actual document (index.html, line 500), line 500 is past the end of the source document, so it's referencing some angular script injected into the header, thus does not seem to be useful for debugging.

- *Q&A2:* Expressions are meant to handle very simple logic, and are optimized for simplicity and not breaking your app.

- *Q&A8:* I don't understand how to use `$scope.$watch` and `$scope.$apply`. The official documentation isn't helpful.

- *Q&A3:* It seems as though the developers built the entire framework and wrote the documentation promoting the idea that it's A-OK to frankenstein JavaScript expressions into your HTML.

- *Q&A3:* Architecturally, the way jQuery solves problems is not the best. Clayton pointed out that it's difficult to test, but it also leaves intent nebulous. Directives extend DOM functionality (and we can test them within and without the view) and so belong within the DOM. When you use jQuery to programmatically insert on-click behavior, whoever is reading the view (designer or developer) doesn't know what something does. With AngularJS, that becomes quite clear.

- *Q&A4:* As I read through the Angular tutorials, I really like a lot of it, but isn't "`ng-click`" the equivalent of an inline `onClick`? My understanding was that the JavaScript community had determined inline JavaScript event handlers in your HTML was "bad practice".

- *Q&A5:* I understand that `ng-click` is technically not the same as `onclick`, but both are placed in the markup. I thought that would be a "bad practice"? Why is this one of the core concepts of AngularJS, if most people say this is "bad"? I thought it would be better to select the DOM element from JavaScript and not to place logic in the markup.

- *Q&A6:* I'm relatively new to angular, and the more I use it the more I insert ng-this and ng-that into my HTML and soon I have a mess like this.

- *Q&A7:* I am very new to AngularJS. Now when I saw the code, I was very curious about all the logic that is placed in the HTML code.

- *Q&A7:* Now about adding logic to the views, if we are talking about business logic then it's a big no no. Use a method on your controller that will evaluate stuff using the service.

- *Q&A7:* If we are talking about `ng-if`/`ng-show` conditions then only if they are small and "readable" conditions I would add them to the view. When it's more than that, I move them to the controller for debugging issues and since I believe the HTML should be readable.

- *Q&A9:* I've been using `$scope.$apply()` to update the bindings for my models when I receive data through websockets in my Angular apps and it works. But what does it actually do and why does it need to be called to achieve the update?

- *Q&A10:* I was wondering what is the correct way to integrate jQuery plugins into my angular app. I've found several tutorials and screen-casts but they seem catered to a specific plugin.
- *Q&A10:* If you are using a jQuery plugin, do not put the code in the controller. Instead create a directive and put the code that you would normally have inside the link function of the directive.
- *Q&A11:* Some places seem to use the controller function for directive logic and other use link. The tabs example on the angular homepage uses controller for one and link for other directive. What is the difference between two?

A.2 Document for Survey About Performance of AngularJS

A.2.1 Sources of Information

1. **Identifier:** DOC1
   Title: Developer Guide: Scopes
   Access date: June-2015
   URL: `https://docs.angularjs.org/guide/scope`

2. **Identifier:** DOC2
   Title: API Reference: `$rootScope.Scope`
   Access date: June-2015
   URL: `https://docs.angularjs.org/api/ng/type/$rootScope.Scope`

3. **Identifier:** DOC3
   Title: API Reference: `ngRepeat`
   Access date: June-2015
   URL: `https://docs.angularjs.org/api/ng/directive/ngRepeat`

4. **Identifier:** BLOG1
   Title: Why You Should Not Use Angularjs

Access date: June-2015
URL: https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99
Author: Egor Koshelko
Author information: Senior Frontend Developer.

5.

Identifier: BLOG2
Title: AngularJS vs. Backbone.js vs. Ember.js
Access date: June-2015
URL: https://www.airpair.com/js/javascript-framework-comparison
Author: Uri Shaked
Author information: The author works for WatchDox, lectures at the Israel Institute of Technology, and organizes the Tel Aviv Google Developers Group.

6.

Identifier: BLOG3
Title: The Problem With Angular
Access date: July-2015
URL: http://www.quirksmode.org/blog/archives/2015/01/the_problem_wit.html
Author: Peter-Paul Koch
Author information: Mobile platform strategist, consultant, and trainer.

7.

Identifier: BLOG4
Title: What's wrong with Angular
Access date: July-2015
URL: http://www.quirksmode.org/blog/archives/2015/01/the_problem_wit.html
Author: Daniel Steigerwald
Author information: JavaScript Developer and Consultant. Member of the Google Developers Expert Program.

8.

Identifier: BLOG5
Title: Lessons learned from AngularJS
Access date: July-2015

URL: http://lhorie.github.io/mithril-blog/lessons-learned-from-angular.html
Author information: Developer in the project Mithril.

9.

Identifier: BLOG6
Title: AngularJS: The Bad Parts
Access date: July-2015
URL: http://larseidnes.com/2014/11/05/angularjs-the-bad-parts/
Author: Lars Eidnes
Author information: Developer from Trondheim, Norway.

10.

Identifier: BLOG7
Title: Angular Bad Parts, Part 2: Performance
Access date: July-2015
URL: http://www.fse.guru/angular-bad-parts-part-2
Author: Alexey Migutsky
Author information: Passionate software engineer, working with Scala, NodeJS and frontend technologies like Angular, ReactJS, HTML5, CSS.

11.

Identifier: BLOG8
Title: Optimizing AngularJS: 1200ms to 35ms
Access date: June-2015
URL: http://blog.scalyr.com/2013/10/angularjs-1200ms-to-35ms/
Author: Steven Czerwinksi
Author information: Staff Software Engineer at Scalyr.

12.

Identifier: BLOG9
Title: Speeding Up AngularJS Apps With Simple Optimizations
Access date: July-2015
URL: http://www.binpress.com/tutorial/speeding-up-angular-js-with-simple-optimizations/135
Author: Todd Motto
Author information: Director of Web Development at Mozio. He works with JavaScript and AngularJS.

13.

Identifier: BLOG10
Title: 11 Tips to Improve AngularJS Performance
Access date: July-2015
URL: `http://www.alexkras.com/11-tips-to-improve-angularjs-performance/`
Author: Alex Kras
Author information: Software Engineer at Support.com.

14.

Identifier: BLOG11
Title: AngularJS: My Solution to the `ng-repeat` Performance Problem
Access date: July-2015
URL: `http://www.williambrownstreet.net/blog/2013/07/angularjs-my-solution-to-the-ng-repeat-performance-problem/`
Author: Thierry Nicola
Author information: Computer Scientist, working with Grails, AngularJS, BackboneJS, and ChaplinJS.

15.

Identifier: BLOG12
Title: AngularJS Performance in Large Applications
Access date: July-2015
URL: `https://www.airpair.com/angularjs/posts/angularjs-performance-large-applications`
Author: Abraham Polishchuk
Author information: NodeJS/AngularJS engineer at Lanetix.

16.

Identifier: BLOG13
Title: Why The World Needed Another AngularJS Grid
Access date: July-2015
URL: `http://www.angulargrid.com/why-the-world-needed-another-angularjs-grid/`
Author: Niall Crosby
Author information: The author has been writing software for 20 years, lately focusing on full stack Java/Javascript/AngularJS.

17.

Identifier: BLOG14
Title: Angular Non-trivial Performance Hints
Access date: July-2015
URL: `http://codetunes.com/2015/angular-non-trivial-performance-hints/`
Author: Radek Markiewicz

18.

Identifier: Q&A1
Title: How Does Data Binding Work in AngularJS?
Access date: July-2015
URL: `http://stackoverflow.com/questions/9682092/how-does-data-binding-work-in-angularjs/9693933#9693933`
Author: Misko Hevery
Author information: Contributor to AngularJS.

19.

Identifier: Q&A2
Title: How Do I Measure the Performance of my AngularJS App's Digest Cycle?
Access date: June-2015
URL: `http://stackoverflow.com/q/23066422/5244036`
Author: Gil Birman

20.

Identifier: Q&A3
Title: AngularJS Scaling & Performance
Access date: June-2015
URL: `http://stackoverflow.com/q/17656397/5244036`

21.

Identifier: Q&A4
Title: How To Improve Performance Of `ngRepeat` Over A Huge Dataset (Angular.js)?
Access date: July-2015
URL: `http://stackoverflow.com/q/17348058/5244036`

22.

Identifier: Q&A5
Title: How To Analyze Performance Benchmark In AngularJS Components?
Access date: July-2015
URL: `http://stackoverflow.com/a/27400474/5244036`

23.

Identifier: Q&A6
Title: Ways To Improve AngularJS Performance Even With Large Number Of Dom Elements
Access date: July-2015
URL: `http://stackoverflow.com/a/16128418/5244036`

24.

Identifier: Q&A7
Title: How To Speed Up An AngularJS Application?
Access date: July-2015
URL: `http://stackoverflow.com/q/15643467/5244036`

25.

Identifier: Q&A8
Title: Angular.js Performance Issues
Access date: July-2015
URL: `http://stackoverflow.com/a/14486570/5244036`
Author: Ben Lesh

A.2.2 Relevant Quotations

- *DOC1:* If a `$watch` changes the value of the model, it will force additional digest cycle.

- *DOC1:* Dirty checking the scope for property changes is a common operation in Angular and for this reason the dirty checking function must be efficient. Care should be taken that the dirty checking function does not do any DOM access, as DOM access is orders of magnitude slower than property access on JavaScript object.

- *DOC1:* Dirty checking can be done with three strategies: By reference, by collection contents, and by value.

- *DOC2:* `$destroy()` must be called on a scope when it is desired for the scope and its child scopes to be permanently detached from the parent and thus stop participating in model change detection and listener notification.

- *DOC2:* Usually, you don't call `$digest()` directly in controllers or in directives. Instead, you should call `$apply()` (typically from within a directive), which will force a `$digest()`.

- *DOC3:* The `ngRepeat` directive instantiates a template once per item from a collection.

- *DOC3:* Special properties are exposed on the local scope of each template instance: `$index`, `$first`, `$middle`, `$last`, `$even` and `$odd`. Creating aliases for these properties is possible with ngInit. This may be useful when, for instance, nesting ngRepeats.

- *DOC3:* `ngRepeat` (as well as other ng directives) supports extending the range of the repeater by defining explicit start and end points by using `ng-repeat-start` and `ng-repeat-end`.

- *BLOG1:* It's slow, and it is especially all turns bad on mobile platforms and when you write something complex.

- *BLOG1:* Angular even imposes restrictions on how rich UI you can write and this is not some ephemeral or distant limit you will never experience. It's only 2000 watchers, and if you develop more or less large applications, you will undoubtedly run into this limitation.

- *BLOG1:* When the user loads the page he may see {{expressions in brackets}}. For these purposes AngularJS introduces new directives: ngCloack, ngBind.

- *BLOG1 (comment):* That number is made up. Not true. Originally shared by Misko on StackOverflow. I'll get him to correct it. Complexity of an expression is often more important than the pure number of expressions. But in general when it comes to pure count we are talking about orders of magnitude bigger numbers than 2k (based on our benchmarks).

- *BLOG1 (comment):* That may be the reason why one way data-binding is now on 1.3. Fortunately, AngularJS evolves.

- *BLOG2:* In general, for pages with a lot of interactive elements, Angular becomes really slow. A good rule of thumb is not to have more than 2,000 active bindings on the same page.

- *BLOG2:* Two-way data binding saves a lot of boilerplate code.

- *BLOG2:* The automatic Dirty Checking means that you don't have to access your model data with getters and setters — you can modify any property of an arbitrary scope object and angular will automatically detect the change and notify all the watchers for that property.

- *BLOG3:* All code snippets in are Angular instructions. The problem is that there is no way for Angular to discover these instructions except by parsing the entire DOM — a very expensive process, especially on mobile.

- *BLOG3:* What worries me is that this non-performant mode is Angular's default (talking about `ng-repeat` directive).

- *BLOG4:* Angular is slow. And with dirty checking and HTML parsing always will be. Do I have to tell something about consequences for mobile web apps?

- *BLOG5:* Another problem related to Angular's re-implementation of scope is the steep performance degradation when things like grids grow past a modest size. Often things look ok with test data, but with production-level data volume, you are suddenly forced to discover about Angular internal concepts like watchers, dirty checking, apply cycles, etc, or you need to do massively time consuming audits of the code base in order to figure out the feasibility of adding the `ng-bindOnce` plugin, or maybe you need to completely rewrite a top-level `ng-repeat` to use the `ng-grid` plugin instead.

- *BLOG5:* It's extremely difficult to reason about Angular performance, and standard tools don't help very much.

- *BLOG6:* The digest loop: It scans through everything that has such a binding, and sees if it has changed by comparing its value to a stored copy of its value. It then scans through everything looking for changes again. This keeps going until no more changes are detected.

- *BLOG6:* Changing anything in the application becomes an operation that triggers hundreds or thousands of functions looking for changes.

- *BLOG6:* This is a fundamental part of what Angular is, and it puts a hard limit on the size of the UI you can build in Angular while remaining performant.
- *BLOG6:* It's not hard to end up with more than 2000 bindings.
- *BLOG6:* On mobiles the performance was dreadful.
- *BLOG6:* There are ways to make bindings happen only once, and with Angular version 1.3 these are included by default. It nevertheless requires ditching what is perhaps the most fundamental abstraction in Angular.
- *BLOG7:* Do you know that each time you call `$scope.$apply()` you actually call `$rootScope.$apply()` and this call updates all scopes and run all your watches?.
- *BLOG7:* Moreover, `$rootScope.$apply()` is called each time when:
  - `$timeout` handler is invoked (almost all debounce services are broken by design)
  - `$http` receives a response (yeah, if you have a polling implemented on `$http`...)
  - Any DOM handler is called (have you throttled your `ng-mouseovers`? They actually invoke ALL your `$watches`, and built-in digest phasing does not really help)
- *BLOG8:* In our early tests, we found that advancing to the next log page could take several agonizing seconds of JavaScript execution. Worse, unrelated actions (such as clicking on a navigation dropdown) now had noticeable lag.
- *BLOG8:* We created a directive that "hides" the change watchers of its children. To accomplish this, we had to slightly break the AngularJS abstraction layer.
- *BLOG8:* A straightforward AngularJS implementation of the log view took 1.2 seconds to advance to the next page, but with some careful optimizations we were able to reduce that to 35 milliseconds. We had to break few rules to implement them.
- *BLOG8 (comment):* AngularJS 1.3 introduces one-time binding to reduce watching and digest loops which is relevant to this.
- *BLOG8 (comment):* Yes, with the current implementation's approach, if the AngularJS team changed the `$scope.$watch` implementation to use a different

non-public variable that we depend on, our change would break and we would have to fix it..

- *BLOG9:* AngularJS dropped a really interesting feature recently in the beta version of 1.3.0: the ability to render data once and let it persist without being affected by future Model updates.
- *BLOG9:* When `$scope.$apply()` is called it runs `$rootScope.$digest()`. This is what actually kicks off the internal digest cycle.
- *BLOG9:* Instead of `$scope.$apply`, we could turn to `$scope.$digest`, which runs the exact same digest loop, but is executed from the current scope downwards through its children. The only caveat to this approach is that if you're dependent on two-way binding between Objects from the parent scope, the parent scope won't be updated until the next `$rootScope` full digest cycle.
- *BLOG9:* Avoid `ng-repeat` where possible. The `ng-repeat` directive is most likely the worst offender for performance concerns.
- *BLOG9:* Instead of rendering a global navigation using `ng-repeat`, we could create our own navigation using the `$interpolate` provider to render our template against an Object and convert it into DOM nodes.
- *BLOG9:* Another offender that will increase `$$watcher` counts are the core Angular directives such as `ng-show` and `ng-hide`. Although these might not immediately increase watcher counts dramatically, they can easily stack up in the hundreds inside an `ng-repeat`.
- *BLOG9:* Angular also provides us with Directives such as `ng-mouseenter`, these can be more costly too as they're not only binding an event listener, they become a part of the digest cycle adding to the application weight.
- *BLOG9:* Limit DOM filters. Angular includes a `$filter` provider, which you can use to run filters in your JavaScript before parsing into the DOM.
- *BLOG10:* In Angular it is fairly easy to add so many watchers that your app will slow down to a crawl.
- *BLOG10:* Angular 1.3 added `::` notation to allow one time binding. Angular will wait for a value to stabilize after its first series of digest cycles; after that, Angular will remove the watcher forgetting about that binding. If you are on pre 1.3 version of Angular you can use this library to achieve similar results.

- *BLOG10:* Use `$watchCollection` instead of mcode$watch (with a 3rd parameter).
- *BLOG10:* If you know there is going to be a lot of changes coming from an `ng-model`, you can de-bounce the input. For example if you have a search input like Google, you can de-bounce it by setting the following `ng-model` option: `ng-model-options="{ debounce: 250 }"`. This will ensure that the digest cycle due to the changes in this input model will get triggered no more than once per 250ms.
- *BLOG10:* Use `ng-if` instead of `ng-show` (but confirm that `ng-if` is actually better for your use case).
- *BLOG10:* Console.time is a great API, and I found it particularly helpful when debugging issues with Angular performance.
- *BLOG10:* Use native JavaScript or Lodash for slow functions: In my tests I got a significant performance boost by simply re-writing some of the basic logic with lodash, instead of relying on built-in Angular methods (which have to account for much more generic use cases).
- *BLOG10:* Use Chrome Timeline and Profiler to identify performance bottlenecks: I like to think of myself as a Chrome Dev Tools power user. But it's not often that I get a to use the Timeline and Profiler views. In this project, both were extremely helpful.
- *BLOG10 (comment):* I just found yesterday a less known easy way to speed up angular : turn off angular's debug features!
- *BLOG11:* After some testing, the response times were very satisfying, however the application was blocking for 500-700ms after each refresh of the view. Quickly I stumble upon the problem: `ng-repeat`.
- *BLOG11:* I was expecting this bad performance due to the internal working of AngularJS.
- *BLOG11:* For `ng-repeat` you often want the items to be rendered and then forget about them as they are not updated from the client side. No two way binding required.
- *BLOG11:* Bindonce (plugin or third-party module) allows to create a template to render that uses jQuery to render the HTML, so no two-way binding. This

solved the problem of the slow rendering, but created another one namely that scrolling now had bad performance (on mobile scrolling was not controllable).

- *BLOG11:* Virtual Scrolling just renders the visible elements and removes invisible elements from the DOM.

- *BLOG11:* Final solution: Upgrade to AngularJS 1.1.5 and use limitTo together with Infinite scrolling.

- *BLOG11:* I slightly adapted the Infinite Scroll directive to make scrolling within a container possible that does not have height 100% of window.

- *BLOG11:* The limitTo with Infinite Scrolling plays very neat together with the Track By feature for `ng-repeat`.

- *BLOG12:* It is important to understand what causes an AngularJS application to slow down.

- *BLOG12:* A fantastic tool for benchmarking one's code is jsPerf. The Chrome Dev Tools have a fantastic JavaScript profiler.

- *BLOG12:* There are several things to be said about performant Javascript that are not necessarily limited to Angular.

- *BLOG12:* Now that we have discussed JavaScript performance, it is important to understand a few key Angular concepts that are somewhat "under the hood".

- *BLOG12:* Of particular importance to writing Angular JS applications in general and performance in particular is the digest cycle. Effectively, every scope stores an array of functions `$$watchers`.

- *BLOG12:* When any value in scope changes, all watchers in the `$$watchers` array will fire, and if any of them modify a watched value, they will all fire again.

- *BLOG12:* We should think through our data model and attempt to limit the complexity of our objects. Use a custom serializer to only return the subset of keys that your Angular application absolutely must have.

- *BLOG12:* Never watch a function result directly. This function will run on every digest cycle.

- *BLOG12:* Every time `$watch` is called on a scope value, or a value is bound from the DOM with interpolation, an `ng-repeat`, an `ng-switch`, and `ng-if`, or any other DOM attribute/element, a function gets added to the `$$watchers` array of the innermost scope.

- *BLOG12:* If non-Angular code is run through `$scope.$apply()`, this will immediately kickstart the digest cycle.

- *BLOG12:* Design with Angular in mind.

- *BLOG12:* Angular provides the ability to watch entire objects by passing a third, optional true parameter to `$scope.$watch`. This is a terrible idea.

- *BLOG12:* `ng-repeat` does some pretty heavy DOM manipulation (not to mention polluting `$$watchers`). Keep any lists of rendered data small whether through pagination or infinite scroll.

- *BLOG12:* Avoid using filters if at all possible. They are run twice per digest cycle and do not actually remove any part of the collection from memory.

- *BLOG12:* It is also important to avoid a global list refresh when using `ng-repeat`. Doing something like `$scope.listBoundToNgRepeat = serverFetch()` will cause a complete recalculation of the entire list, causing transcludes to run and watchers to fire for every individual element. This is a very expensive proposition. There are two ways around this. One is to maintain two collections and `ng-repeat` over the filtered set. The other is to use *track by* to specify your own key.

- *BLOG12:* Track by approach only works when a field on the repeated object can be guaranteed to be unique in the set.

- *BLOG12:* `ng-hide` and `ng-show` simply toggle the CSS display property: Any scopes will exist, all `$$watchers` will fire, etc. `ng-if` and `ng-switch` actually remove or add the DOM completely. Unfortunately this results in a case by case judgment call.

- *BLOG12:* A common source of slow Angular applications is incorrect use of `ng-hide` and `ng-show` over `ng-if` or `ng-switch`. The distinction is nontrivial, and the importance can not be overstated in the context of performance.

- *BLOG12:* The problem arises in the fact that `$scope.$apply` starts at `$rootScope` and walks the entire scope chain causing every scope to fire every watcher.
- *BLOG12:* In general, `$scope.$watch` is indicative of bad architecture.
- *BLOG12:* `$on`, `$broadcast` , and `$emit`, like `$watch`, are slow as events (potentially) have to walk your entire scope hierarchy.
- *BLOG12:* `evalAsync` can greatly improve your page's performance.
- *BLOG12:* By creating a new scope with Isolate Scope or Transclusion, we are creating a new object to track, adding new watchers, and therefore slowing down our application.
- *BLOG12:* Angular provides many pre-rolled DOM event directives. `ng-click`, `ng-mouseenter`, `ng-mouseleave`, etc. All of these call `$scope.$apply()` every time the event occurs. A much more efficient approach is to bind directly with addEventListener, and then use `$scope.$digest` as necessary.
- *BLOG13:* You see, Angular JS is great for form based applications, where you don't have hundreds of bound values laid out in a grid and scrolling.
- *BLOG13:* The extra logic that AngularJS puts in behind the scenes doesn't work in your favor when building something like a complex grid.
- *BLOG14:* Avoid complicated watcher's evaluation.
- *BLOG14:* There is no way to think about the higher level of Angular performance without understanding a few things:
  - How digest cycle works in general
  - Difference between `$scope.$apply()` and `$scope.$digest()`
  - List of Angular core features triggering digest cycle
- *Q&A1:* You can't really show more than about 2000 pieces of information to a human on a single page. Anything more than that is really bad UI, and humans can't process this anyway.
- *Q&A1:* Unfortunately it is way too easy to add a slow comparison into AngularJS, so it is easy to build slow applications when you don't know what you are doing.

- *Q&A2:* What is a simple way to measure the duration of the angularjs digest cycle?
- *Q&A3:* You need to create custom directives in order to curb the performance issues with angular. Unlike ember angular comes with all the bells and whistles turned on and it's up to you to tone it down.
- *Q&A4:* Most straightforward approach (fetch data, put it into `$scope`, let `ng-repeat=“”` do its job) works fine, but it freezes the browser for about half of a minute when it starts inserting nodes into DOM.
- *Q&A5:* The `$apply` function itself is relatively light (you can inspect it in the angular source). It is the process of evaluating watchers and comparing values (dirty-checking) during a `$digest` that can get expensive.
- *Q&A6:* I think that the best way to solve performance issues is to avoid using high level abstractions (AngularJS `ng-repeat` with all corresponding background magic) in such situations. AngularJS is not a silver bullet and it's perfectly working with low level libraries.
- *Q&A7:* The thing you can do that will speed up your Angular app the most is to reduce those bindings where you can. One way to do this would be to create a directive that built out the table for you with DOM manipulation rather than using `ng-repeat`s.
- *Q&A8:* Every `ngRepeat` you set up, sets up a watch. Every `{{binding}}` or `ngModel` you do inside of that repeat, sets up another watch, and so on. Each of those creates function and object instances, and also needs to be processed on every `$digest`. So, if your running into performance issues, you may need to implement a custom directive that writes out your data without setting up those superfluous watches, so you're a little more performant.

Appendix B

Questionnaires

In this appendix, we present the questionnaires used to collect the responses of the empirical studies presented in Chapter 3 and Chapter 4.

B.1 General AngularJS Questionnaire

AngularJS

This survey is part of a MSc dissertation about MVC frameworks in JavaScript. It can be answered in around 15 minutes and its goals are strictly academic.

Background

*** 1. In what country do you currently reside?**

*** 2. How many years of experience do you have in JS?**

- Less than 1 year
- 1 to 2 years
- 2 to 5 years
- More than 5 years

*** 3. How many years of experience do you have in AngularJS 1.x?**

- Less than 1 year
- 1 to 2 years
- 2 to 3 years
- More than 3 years

*** 4. What is the size of the largest application you have developed in AngularJS?**

- less than 1 KLOC
- 1 KLOC to 5 KLOC
- 5 KLOC to 10 KLOC
- More than 10 KLOC

*** 5. Which release of AngularJS are you using in your current development?**

- 1.0.x

- ○ 1.1.x
- ○ 1.2.x
- ○ 1.3.x
- ○ 1.4.x
- ○ 1.5.0-beta.x
- ○ Other (please specify)

From now on, please skip questions when you are not confident to provide an answer.

11%

Next

AngularJS

Key Characteristics of AngularJS

Please rate how much value do you give to the following AngularJS features and characteristics.

6. Predefined components for organizing my code

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

7. Dependency injection

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

8. I can implement the models of my applications using plain old JS objects (POJO)

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

9. The ability to specify templates directly in HTML documents

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

10. Support to custom components (extending HTML syntax through directives)

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

11. Ease of writing tests

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

12. Almost no code is required to implement two-way data binding

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

13. The ability to use HTML to declare User Interface (UI) behavior

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

14. AngularJS provides a solution to manage many common problems in one place

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

15. It is supported by Google

1 (not valuable)	2	3	4 (very valuable)
○	○	○	○

22%

Prev Next

AngularJS

Code in HTML Templates

Please rate how often the following items represent a real problem (in your development) caused by code in AngularJS templates

16. Silent failures (i.e., failures that do not trigger error messages)

For an example, please check this StackOverflow question

1 (never)	2	3	4 (very often)
○	○	○	○

17. Code hard to debug or variables hard to dump

1 (never)	2	3	4 (very often)
○	○	○	○

18. Low readability of HTML templates

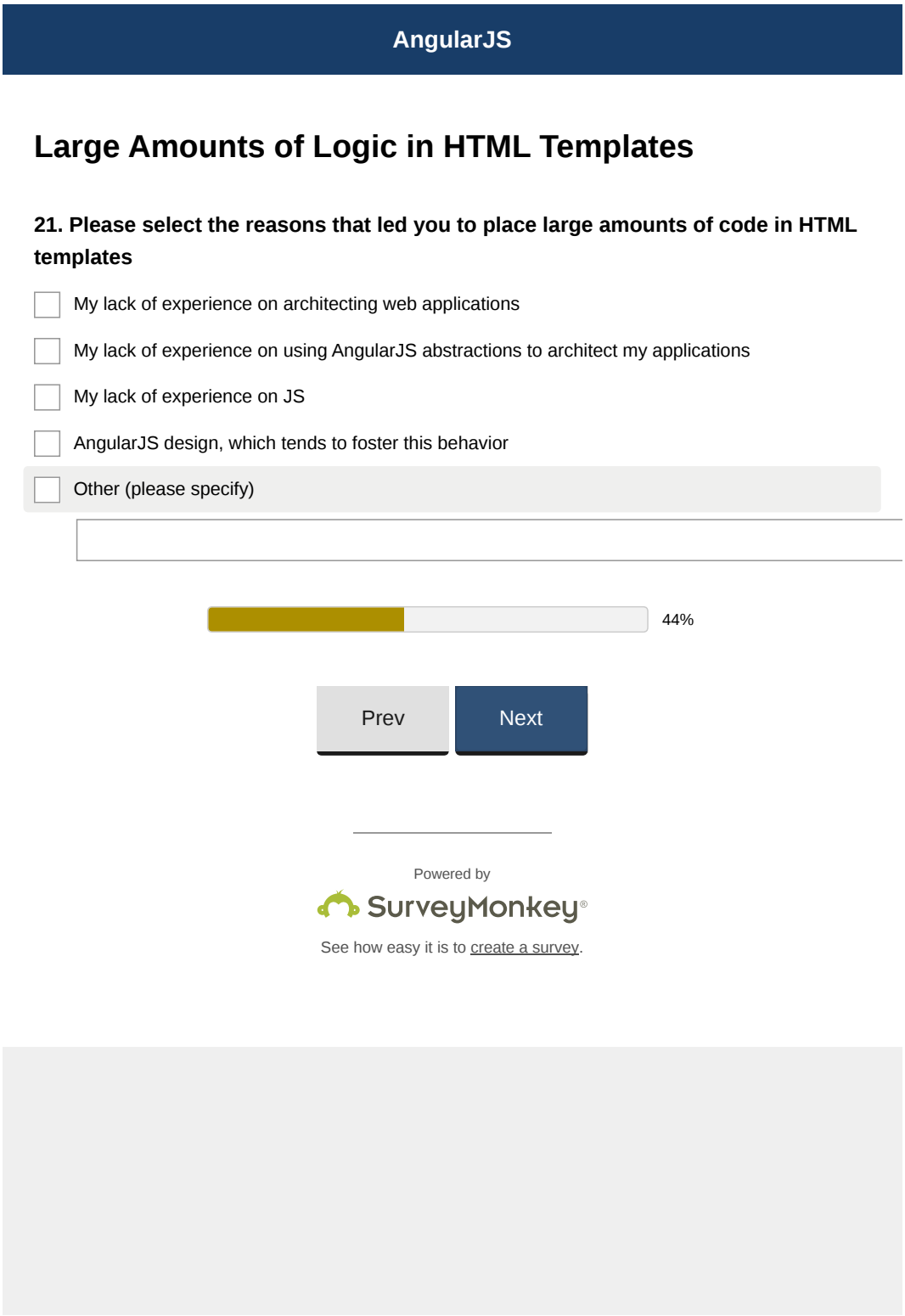
1 (never)	2	3	4 (very often)
○	○	○	○

19. Code in templates hampers modularity and separation of concerns

1 (never)	2	3	4 (very often)
○	○	○	○

20. Have you ever placed large amounts of logic in HTML templates?

- ○ Yes
- ○ No



AngularJS

Large Amounts of Logic in HTML Templates

21. Please select the reasons that led you to place large amounts of code in HTML templates

- [ ] My lack of experience on architecting web applications
- [ ] My lack of experience on using AngularJS abstractions to architect my applications
- [ ] My lack of experience on JS
- [ ] AngularJS design, which tends to foster this behavior
- [ ] Other (please specify)

44%

Prev Next

Powered by SurveyMonkey

See how easy it is to create a survey.

AngularJS

Debugging

22. How often do you use ngMock module for testing?

1 (never)	2	3	4 (very often)
◯	◯	◯	◯

23. How useful is the default error stack-trace provided by AngularJS?

1 (useless)	2	3	4 (very useful)
◯	◯	◯	◯

24. Have you ever overrided or decorated the $exceptionHandler service?

◯ Yes

◯ No

Template Debugging

Please rank how often do you use the following strategies to debug binding expressions (like {{message}})

25. I use specialized tools (e.g., Batarang, ng-inspector)

1 (never)	2	3	4 (very often)
◯	◯	◯	◯

26. I decorate the $interpolate service

1 (never)	2	3	4 (very often)
◯	◯	◯	◯

27. I implement custom AngularJS filters

For an example, please check this StackOverflow question

1 (never)	2	3	4 (very often)
◯	◯	◯	◯

28. I fetch DOM elements and then I use the API provided by angular.element() to check their properties

1 (never)	2	3	4 (very often)
◯	◯	◯	◯

29. If you often use another strategy, please describe it here:

Testing

Please rate how difficult is it to test the following items in AngularJS.

30. Controllers

1 (very easy)	2	3	4 (very difficult)
◯	◯	◯	◯

31. Filters

1 (very easy)	2	3	4 (very difficult)
◯	◯	◯	◯

32. Services

1 (very easy)	2	3	4 (very difficult)
◯	◯	◯	◯

33. Providers

1 (very easy)	2	3	4 (very difficult)
◯	◯	◯	◯

34. Simple HTML element directives

1 (very easy)	2	3	4 (very difficult)
○	○	○	○

35. Transclusion directives

1 (very easy)	2	3	4 (very difficult)
○	○	○	○

36. Directives with external templates

1 (very easy)	2	3	4 (very difficult)
○	○	○	○

56%

Prev Next

AngularJS

AngularJS Development Practices

37. How often do you *avoid* the use of $scope in your applications?

1 (never)	2	3	4 (very often)
○	○	○	○

38. How often do you create your own directives when writing AngularJS applications?

1 (never)	2	3	4 (very often)
○	○	○	○

39. Which Router do you use in your AngularJS applications?

- ○ AngularJS Route Module (angular-route.js)
- ○ AngularUI Router
- ○ I do not use routing in my applications
- ○ Other (please specify)

40. Please mention the main AngularJS plugins that you use to build your applications

67%

Prev Next

AngularJS

Mastering AngularJS

Please rate how complex is the comprehension and correct usage of the following AngularJS aspects and features.

41. The three different types of scope that can be introduced by a directive

1 (very simple)	2	3	4 (very complex)
○	○	○	○

42. Scopes using prototypal inheritance

1 (very simple)	2	3	4 (very complex)
○	○	○	○

43. Defining which entity (provider, service, factory, value or constant) to use when declaring dependencies

1 (very simple)	2	3	4 (very complex)
○	○	○	○

44. Using the compile, link and controller functions when building directives

1 (very simple)	2	3	4 (very complex)
○	○	○	○

45. The API provided to build directives using transclusion

1 (very simple)	2	3	4 (very complex)
○	○	○	○

46. Using correctly $scope.apply (including when to use it)

1 (very simple)	2	3	4 (very complex)
○	○	○	○

47. Integration with external components

1 (very simple)	2	3	4 (very complex)
○	○	○	○

48. Tackling all the details that may lead to performance degradation

1 (very simple)	2	3	4 (very complex)
○	○	○	○

49. Using the correct syntax to inject dependencies in order to avoid problems with minification

1 (very simple)	2	3	4 (very complex)
○	○	○	○

78%

Prev Next

AngularJS

AngularJS 2.0

50. When developing your current applications, how often do you make design decisions having in mind compatibility with AngularJS 2.0 release?

1 (never)	2	3	4 (very often)
○	○	○	○

51. Please select the AngularJS 2.0 features that you are most eager to use

- [ ] The new API to define the main UI building blocks (components and directives)
- [ ] Zone.js (no more $scope.$apply())
- [ ] Server rendering
- [ ] The new logging service called diary.js (an execution time measurement service)
- [ ] New mobile features (e.g. support for touch event gestures)
- [ ] The new template syntax ([properties] and (events))
- [ ] The new change detection mechanism (immutable/observable)
- [ ] The new API for injecting dependencies
- [ ] Integration with ES6 modules
- [ ] Component Router (which will enable the asychronous loading of different components)
- [ ] Other (please specify)

89%

Prev Next

AngularJS

52. Please leave any comment that you consider relevant to our research

Just in case that you want to receive a message with our results

53. Name

54. Email

100%

Prev Done

B.2 AngularJS Performance Questionnaire

Performance in AngularJS

This survey is part of a MSc dissertation about the performance of AngularJS applications. It can be answered in less than 10 minutes and its goals are strictly academic.

* Required

1. **How many years of experience do you have in JavaScript?** *
   Mark only one oval.
   - ◯ Less than 1 year
   - ◯ 1 to 2 years
   - ◯ 2 to 5 years
   - ◯ More than 5 years

2. **How many years of experience do you have in AngularJS 1.x?** *
   Mark only one oval.
   - ◯ Less than 1 year
   - ◯ 1 to 2 years
   - ◯ 2 to 3 years
   - ◯ More than 3 years

3. **What is the size of the largest application you have developed in AngularJS?** *
   Mark only one oval.
   - ◯ less than 1 KLOC
   - ◯ 1 KLOC to 5 KLOC
   - ◯ 5 KLOC to 10 KLOC
   - ◯ More than 10 KLOC

4. **What percentage of the development time do you usually spend improving the performance of AngularJS applications?** *
   Mark only one oval.
   - ◯ 0%
   - ◯ 1% - 5%
   - ◯ 5% - 10%
   - ◯ 10% - 20%
   - ◯ More than 20%

5. **How difficult is it to reason about the causes of performance problems in AngularJS applications?** *
Mark only one oval.

	1	2	3	4	5	
Very easy	◯	◯	◯	◯	◯	Very hard

6. **Which AngularJS releases do you think include major performance improvements?** *
Mark only one oval.

- ◯ Angular 1.1
- ◯ Angular 1.2
- ◯ Angular 1.3
- ◯ Angular 1.4
- ◯ I am not sure

7. **Have you ever inspected AngularJS source code to improve performance?** *
Mark only one oval.

- ◯ Yes
- ◯ No

8. **Have you ever changed AngularJS source code to improve performance?** *
Mark only one oval.

- ◯ Yes
- ◯ No

9. **Have you ever used a third-party component to improve the performance of an AngularJS application?** *
Mark only one oval.

- ◯ Yes
- ◯ No

10. **If yes, which component?**

.....

11. **Have you ever created a custom component to improve the performance of an AngularJS application?** *
Example: a custom implementation of ngRepeat
Mark only one oval.

- ◯ Yes
- ◯ No

12. **If yes, please describe this component.**

13. **Have you ever refactored an AngularJS application to improve its performance on mobile devices?** *
Mark only one oval.

- ◯ Yes
- ◯ No

14. **Have you ever refactored a AngularJS application by replacing two-way data binding with one-time data binding to improve its performance?** *
(including when you used a plug-in supporting one-time data binding)
Mark only one oval.

- ◯ Yes
- ◯ No

15. **Which tool do you use to reason about the performance of AngularJS applications?** *
Check all that apply.

- ☐ I have never used any tool
- ☐ Chrome developer tools
- ☐ Batarang
- ☐ Other:

16. **If you mentioned a tool, which is the main purpose of using it?**

17. **Based on your experience, what is the maximum number of active bindings an AngularJS application can handle without presenting performance problems?** *
Mark only one oval.

- ◯ 1500
- ◯ 2000
- ◯ 2500
- ◯ 3000
- ◯ 3500
- ◯ 4000
- ◯ I have no idea

Performance in AngularJS

General causes of performance problems in AngularJS

Please rank how often these items are the main causes of performance problems in AngularJS applications

18. **Application developers lack of experience in JavaScript** *
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

19. **Application developers lack of experience in AngularJS** *
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

20. **Application following a wrong architecture** *
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

21. **AngularJS internal architecture** *
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

Technical causes of performance problems (please skip questions when you are not confident to provide an answer)

22. **Unnecessary use of two-way data binding when one-time data binding is enough**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

23. **$watching by value when $watching by reference or by collection is enough**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

24. **$watching the result of complex functions**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

25. **Use of filters in view templates instead of in JavaScript code**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

26. **Use of event directives such as ng-click, ng-mouseenter, ng-mouseleave, etc**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

27. **Use of ng-hide and ng-show over ng-if or ng-switch**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

28. **Inappropriate use of $scope.$apply()**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

29. **Runtime configurations and flags (when moving the code to production)**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

30. **Not cleaning up the application (e.g., unbind events or cancel asynchronous operations) after destroying components**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

31. **Use of ng-repeat directive without using the TRACK BY feature**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

32. **Use of directives such as ng-show inside of ng-repeat directive**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

33. **Use of long lists with the ng-repeat directive**
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

34. **Unconscious use of features that trigger the full digest cycle**
Examples: $http, $timeout, $location
Mark only one oval.

	1	2	3	4	5	
very rarely	◯	◯	◯	◯	◯	very often

Final Comments

35. **Please leave any comment that you consider relevant to our research**

Contact Information

Just in case that you want to receive a message with our results.

36. **Name**

37. **Email**