PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

JAVASCRIPT MV* FRAMEWORKS FROM A PERFORMANCE POINT OF VIEW

SEBASTIÁN VICENCIO RODRÍGUEZ

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JAIME NAVÓN COHEN

Santiago de Chile, January 2014

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

JAVASCRIPT MV* FRAMEWORKS FROM A PERFORMANCE POINT OF VIEW

SEBASTIÁN VICENCIO RODRÍGUEZ

Members of the Committee:
JAIME NAVÓN COHEN
ROSA ALARCÓN CHOQUE
LIUBOV DOMBROVSKAIA
MIGUEL RÍOS OJEDA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, January 2014

To all people who have always
supported me

ACKNOWLEDGEMENTS

In general terms, I would like to thank all the people that always supported me, and believed that I would be able to finish this research in time. It is a good feeling knowing that you are truly supported.

Firstly, I would like to thank my advisor, Jaime Navón. Ever since I had the idea to enter this Master's program, he has guided me and let me look for a research subject that I felt passionate about. It was not an easy task, and it took a really long time, but it finally came out as a very interesting and inspiring subject. From this point onwards, he continued to guide me throughtout the whole research process and I am very grateful for that.

Secondly, I would like to thank my family and my girlfriend, for always supporting me, right from the beginning of this research project, and understanding when I had to spend my free time reading and studying. Without their support, I could not have finished this.

Finally, I would like to thank my friend Martín Concha, who helped me in one way or another, giving me advise in the most critical moments. I know there are also other people that helped me, without me even noticing, and I appreciate that too.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The architecture of Web applications has changed dramatically in the last few years. From a server taking a protagonistic role and a client limited to supply the view component, to an interactive client that contains most of the application code.

The rapid development of the mobile Web has enhanced this tendency with the rise of what is known as Single Page Application (SPA). This change towards the client side brings together a substantial increase in the size of the JavaScript code, which not only is involved in user interaction, but also in other tasks such as routing and data management. To manage this more complex reality, several frameworks that implement variants of the MVC pattern (known as MV*) have been proposed and built. Each one of these frameworks has its own relative merits in terms of how well they support software development and maintenance. However, very little information exists about the effects that these complex pieces of software have in the final performance of the application. In addition, there is little information about how these frameworks compare between them in terms of performance. In this work we put a standard Web application, implemented using the most popular frameworks, through a series of performance tests in order to get answers to some of these questions. The tests were first conducted using the jQuery library, and then using the most popular MV* frameworks. The results show that there are differences between the frameworks in terms of performance, but their use does not introduce heavy costs in terms of performance to the Web application itself.

Keywords: JavaScript, Single page applications, JavaScript MV* frameworks, Performance, Frontend, jQuery, Backbone.js, Ember.js, AngularJS, KnockoutJS, TodoMVC

RESUMEN

En los últimos años, la arquitectura de una aplicación Web ha cambiado en forma importante. Desde un servidor que tomaba un rol protagónico y un cliente que se limitaba a proporcionar el componente visual, hacia un cliente interactivo que contiene la mayor parte del código de la aplicación.

El avance acelerado de la Web móvil ha acentuado esta tendencia dando origen a las llamadas aplicaciones de una sola página (SPA, por su nombre en inglés). Este cambio de énfasis hacia el lado del cliente trae consigo un aumento en la cantidad de código JavaScript de la aplicación, el cual se encarga no sólo de la interacción con el usuario, sino que también de tareas de enrutamiento, manejo de datos, etc. Para manejar adecuadamente esta nueva realidad, han surgido numerosos frameworks que implementan variaciones del patrón MVC en el lado del cliente (y que se conocen como MV*). Cada uno de estos frameworks tiene su mérito relativo en términos de qué tan bien facilitan la tarea de desarrollo y mantención del código de la aplicación. Sin embargo, no existe mucha información sobre los efectos que estas piezas complejas de software tienen en términos de desempeño. Además también hay poca información acerca de cómo estos framworks se comparan entre ellos en términos de desempeño. En este trabajo se presentan pruebas de desempeño a las que fue sometida una misma aplicación Web estándar, implementada primero usando sólo la librería jQuery, y luego bajo los frameworks MV* más populares. Los resultados obtenidos muestran que, aunque hay diferencias entre los diversos frameworks, la utilización de ellos no introduce una penalización significativa en términos de desempeño de la aplicación Web.

Palabras Claves: JavaScript, Single page applications, JavaScript MV* frameworks, Performance, Frontend, jQuery, Backbone.js, Ember.js, AngularJS, KnockoutJS, TodoMVC

1. CHAPTER 1: INTRODUCTION

1.1. What is the Web?

The *Web* is a word that we used to hearing in our everyday's lives. It's inserted in everything: news, business, entrepreneurship, school and even entertainment. Phenomenoma like social networking (with Facebook and Twitter, among others) are just proof that everything works around the Web. Yet many large-scale systems (enterprise, banks) are built upon Web technologies, so it is clear that the Web is a very important ecosystem and is here to stay for a long time.

But what exactly is the Web? Its long name is World Wide Web (WWW), and it was invented by Tim Berners-Lee in 1989 (Berners-Lee, 1989). To put it in simple words: it's a system of interlinked documents accessed via the Internet. In order for the Web to work, three main components are required:

(i) HTML: **H**yper**T**ext **M**arkup **L**anguage, it is the language of the documents used in the Web. HTML is composed by elements (HTML tags), which contains attributes (some optional and used to define extra information) (Docs, 2014). These elements are used to surround or *markup* the different pieces of content of the document, and each one has a different use (depending on semantics). An HTML document has a hierarchical tree structure, which means one root element (usually `<html>`), and the rest as children elements. A typical HTML structure includes a `<head>` for several configurations (not visible) and a `<body>` wherein resides the visible content. The example 1.1 shows a basic HTML document.

Example 1.1. *Basic HTML document structure*

```
<html>
    <head>
        <title>A page title</title>
    </head>
```

```
    <body>
      <h1>Visual title header</h1>
      <div>Some content</div>
    </body>
</html>
```

(ii) URI: **U**niform **R**esource **I**dentifier, it is a unique text identifier of a Web document or Web resource. Hyperlinks inside an HTML document reference a target document using URIs. The basic syntax of an URI is:

`scheme_name://hierarchical_part?query_part`

The scheme name, in the Web context, indicates which protocol to use (commonly `http`). The hierarchical part, on the other hand, refers to the unique resource, in a hierarchical way (general to particular). The query part, at the end, is usually optional, and is used to request extra information (for example, filtering or sorting results).

(iii) HTTP: **H**yper**T**ext **T**ransfer **P**rotocol, it is the protocol used to establish a communication (using the Internet) between a machine that needs a Web resource or document, and the one that has it. The two sides communicate using a request-response paradigm in a synchronous way, meaning that one side requests a resource and then it has to wait until the other side sends a response. HTTP needs a URI to work, but also defines HTTP methods to indicate which action has to be performed on the resource. The most common methods are: `GET` (retrieve resource), `POST` (create resource), `PUT` (modify the whole resource) and `DELETE` (delete resource). The combination of a URI and an HTTP method makes possible the navigation through the Web.

1.2. How the Web works

1.2.1. Client and Server architecture

The Web resources or documents of which the Berners-Lee (1989) proposal talks about have to reside somewhere in order to be able to be fetched. That place is called a *server*. It is a machine that stores information, and that information can be retrieved thanks to a program running inside of it, called Web server. The Web server understands HTTP, meaning that it can receive an HTTP request for a specific document or resource, and then send back an HTTP response with the document itself. Which document is sent back as a response is not the responsibility of the Web server, but it is from another program (software) running inside the server that can interpret a URI and return the corresponding document or resource.

On the other side of the HTTP communication, there is the entity that requests a document. This entity is called *client*, and it also understands HTTP, so after requesting a document, it can receive an HTTP response from the server, which includes the document or resource inside its content. The interaction is shown en Figure 1.1.

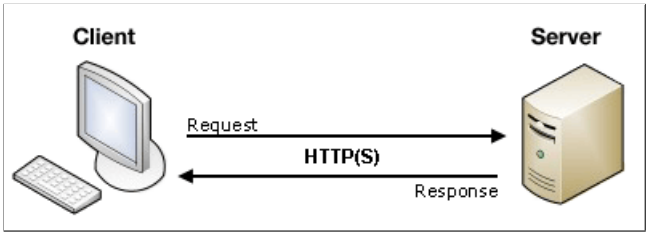

FIGURE 1.1. Client and server interaction through HTTP. Source: `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms/Sending_and_retrieving_form_data`

1.2.2. Browsers

One implementation of a client is a *Web browser*, a software application used for retrieving Web resources (using HTTP) and then presenting them to the user. A browser can generate an HTTP request, and then receive an HTTP response from a server, which includes an HTML document. One of the main tasks of this software application is being

able to parse the HTML document (building a tree called *Document Object Model* or *DOM*, which contains an abstraction of every HTML element), and then interpret every element inside of it, in order to visually render the content of the document to the user.

The HTML document not only includes content, but also references to style sheets (in a language called *CSS*) used for adding style to the content of the document, and it usually includes also scripts in a language called *JavaScript*, used for adding behavior to the document (for example, showing a message when the user clicks a button). The CSS files and JavaScript scripts are also parsed by the browser in order to add these features to the document.

In short, a Web browser is an application used to *browse the Web*, entering URIs and rendering the corresponding result to the user.

1.2.3. Web applications

A browser can render HTML documents, as the result of a response from a server. Usually this server provides related resources, which together form a set of Web documents. When this set includes only static resources (which may not change by user interaction), it is refered to as a *Website*.

However, when this set of documents includes dynamic resources, advanced user interactions and access to advanced browser capabilities, we are talking of a more complex piece of software called Web application (Borodescu, 2013). As Owen (2010) points out: "a Web application is an application that uses the Web and a simple browser to interact with the user". The difference between a Web application and a desktop application is that in a Web application, most of the logic resides in a remote server, which executes code depending on the request made by a client (browser). This approach implies that every time a request is made, the browser has to wait until the response is received, giving the user the perception of an interruption. As it is explained in the next sections, there has been a recent trend where some of this application logic has been moved from the server to the browser, in the form of JavaScript (the language that a browser can execute).

1.3. Traditional Web applications

The first Web applications of the 90s were completely different to a modern Web application. As was mentioned above, for years the main part of the content of a Web application was processed and executed in a remote server, which is usually called *backend*, because from a user perspective, the application runs in a back or hidden place.

These Web applications only presented static views to the user, and the only way to change these views was through an explicit action of the user (for example, clicking a link or filling and submiting a form). These actions generated a request to the server in a synchronous way (because of the HTTP nature), which means that the user had to wait for the whole roundtrip to the server before getting a response to the given request. What the user saw in the meantime was a blank screen until the new resource was loaded. This approach lasted for more than a decade, and it was considered the standard way to access a Web application.

1.4. Ajax appearance

However, no one considered at that time that a request to the server could be generated not only explicitly by a user action, but also through code executed by the browser, JavaScript code to be more precise.

Microsoft introduced in Internet Explorer 5.0 an object called *XMLHttpRequest* as an ActiveX control (MSDN, 2013), which enabled the browser to make asynchronous HTTP requests, and receive an XML document as a response. In time, this object was adopted by all major browser vendors (Mozilla, Apple, Google), implemented as a JavaScript object, and even standardized in the W3C (W3C, 2012). The name (abbreviated XHR) is merely inherited and it is not restricted to XML responses, but it can also handle JSON, HTML and even plain text responses. The XHR object provides an easy way to get data from a server without doing a full page refresh (MDN, 2014).

If the XHR object is used in combination with some manipulation of the elements in the DOM of a Web page, it is possible to update the content of a Web application in

the background, giving the user the perception that it was done automatically, without interruptions (no more blank screens). This approach is known as Ajax, a term that was introduced by Garrett (2005). The difference between a traditional or classic Web application and an Ajax application is shown in Fig. 1.2.

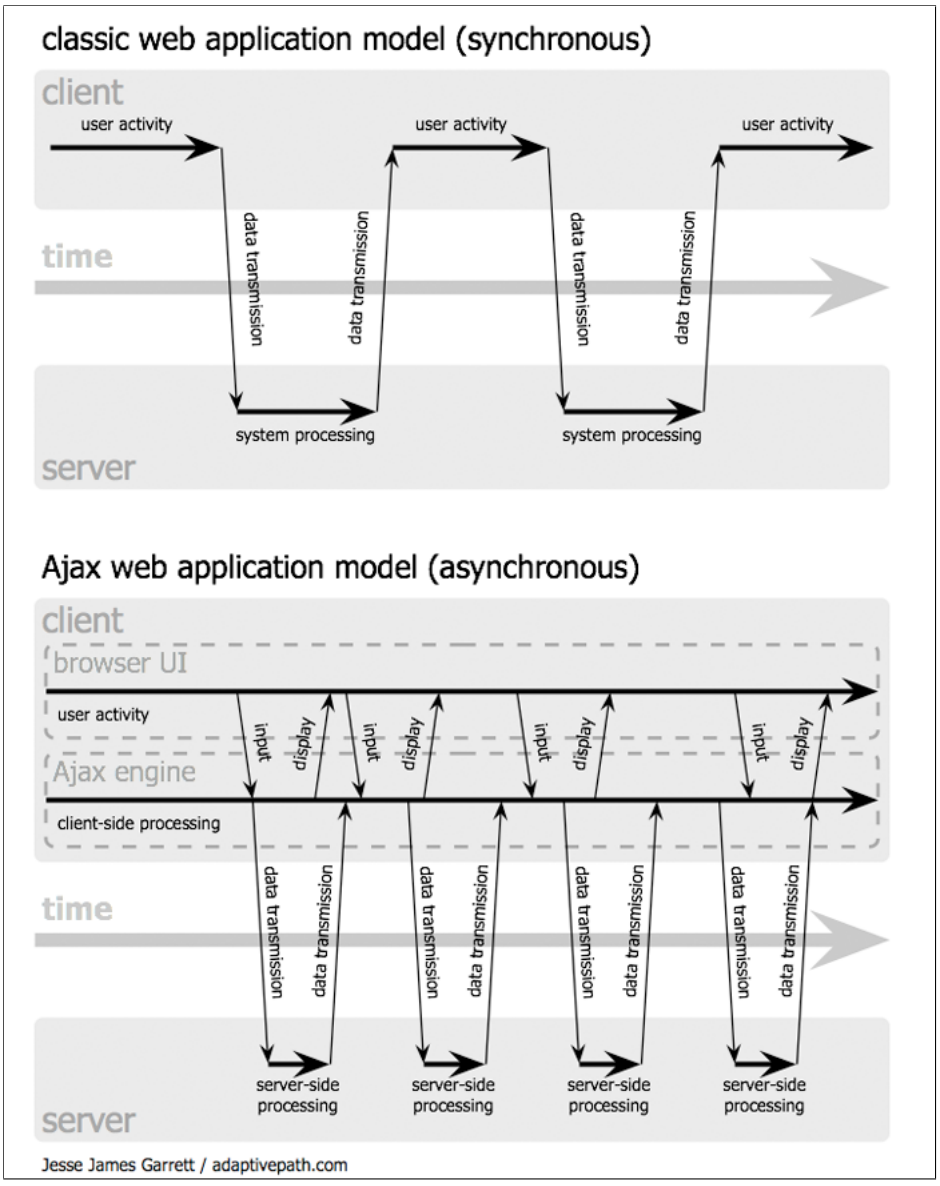

FIGURE 1.2. The synchronous interaction pattern of a traditional web application (top) compared with the asynchronous pattern of an Ajax application (bottom). Source: Garrett (2005)

The introduction of the Ajax approach completely changed the way in which Web development was done. From that moment on the client side, known as application *frontend*, began to have much more responsibility and importance.

1.5. JavaScript frameworks

Since the introduction of Ajax, the JavaScript code size of a Web application started to grow at a very fast rate. However, handling an Ajax request with pure XHR JavaScript objects is a bit verbose and a repetitive task if an application makes a lot of use of this feature (a symptom known as boilerplate code).

The Web community addressed this issue, and some solutions were developed in order to reduce code writing and facilitate the development of a Web application. These solutions grouped the main funcionalities for which JavaScript is used for in a JavaScript library: DOM handling (with the ability to select any HTML element), capture events originated by the user and Ajax requests. The library exposes a set of functions to access all those features. There are several JavaScript libraries like *jQuery*, *Prototype* and *YUI*, among others. jQuery is the most popular of them, with more than 50% of all websites using it, and with a JavaScript library market share of 90.2% (Gelbmann, 2013).

These solutions are part of the first JavaScript Web frameworks. But what is a Web framework? According to Shan and Hua (2006), it is "a reusable, skeletal, semi-complete modular platform that can be specialized to produce custom web applications". So a JavaScript (JS) Web Framework acts as a frontend framework, which means that, independently of the backend side, this kind of framework adds behavior to the Web application after it has been loaded in the browser.

1.6. Single-page applications (SPA)

If we take the Ajax approach and apply it throughout the whole Web application, what we get is an application that, besides the first load, never reloads a page. HTTP requests are done in the background and only for fetching data, not HTML documents. Such Web

application is known as Single-page application (SPA). Mesbah and van Deursen (2007) give a simple definition of what an SPA is: “The single-page web interface is composed of individual components which can be updated/replaced independently so that the entire page does not need to be reloaded on each user action”.

Single-page apps have the ability to redraw any part of the visible content (User interface) without requiring a server roundtrip to retrieve HTML (Takada, n.d.). It works as follows: the browser makes a request to a remote server, which responds with an HTML document that can contain only a JavaScript resource reference, or it can contain some basic initial HTML too. This JavaScript reference points to the whole Web application, which is initialized and executed in the frontend by the browser. If the user interacts with the application, new data may be required from the server, but instead of reloading the entire page, an Ajax request is done in the background to get the data, and the DOM is updated with the new content (Petersson, 2012).

In terms of data manipulation in an SPA context, what a browser typically access in a remote server when a request is made is known as an *Application Programming Interface* (*API*). It is an intermediary that allows a client to retrieve data in a predefined format (for example, *JavaScript Object Notatio*n or *JSON*) which is understood by both client and server. Figure 1.3 shows the typical communication flow of an SPA.

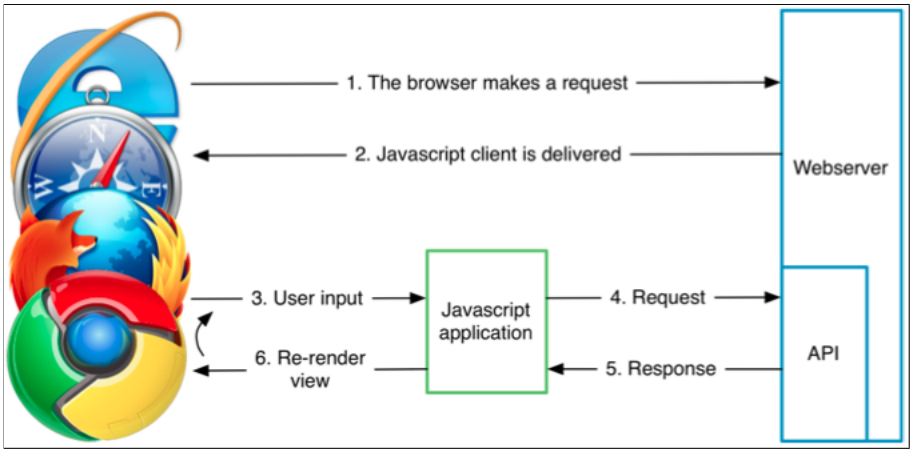

FIGURE 1.3. SPA and server communication. Source: Petersson (2012)

1.6.1. Issues related to SPAs

The SPA approach is recent and represents a growing trend (Podila, 2013). The main reason to develop an application this way is that it allows us to offer an experience to the user closer to what it would be in a native application (Takada, n.d.). An SPA is an application full of JavaScript code (the whole application is executed in the browser). This implies that the JavaScript code of a modern Web application is growing fast, and jQuery is (still) the most used library to achieve this goal.

The jQuery library provides lots of features to ease the SPA development, however, it does not provide a way to structure and organize code (Osmani, 2013). It is not hard to end up with a mix of code that has different responsibilites, but only for being all together generates confusion in the developer itself. This is a known problem and it is called *spaghetti code* (SourceMaking, n.d.). As a result, the code of the application tends to be very difficult to maintain, even for the developers that first wrote it.

1.7. JavaScript MV* frameworks

The Web community addressed the spaghetti code issue and, like has always happened in the past, solutions have been created. The main goal is to build "something" that provides structure to an SPA (from an architectural perspective), and a way to organize its code. If this goal is accomplished, the result is a maintainable Web application.

The solutions that have been developed accomplish this goal, but also add the ability to implement SPA common features/tasks, like DOM manipulation, data management and syncing (using Ajax), a way to generate HTML easier (templating) and URL routing, among others. All of those solutions use a design pattern or paradigm known as *Model-View-Controller* (*MVC*), so first we are going to go deeper into this concept.

1.7.1. MVC design pattern

MVC is an architectural design pattern that provides a separation of concerns inside an application. Three concerns are particularly identified and isolated: data management

(Model), user interface generation (View), and application or business logic (usually triggered by user interaction) to tie up the Model and the View (some people even call it the "glue" between Model and View).

The MVC pattern was first introduced as part of the Smalltalk-80 language, but it has been modified since then. Since we are focused on the modern MVC applied to the Web, we will not discuss the first MVC release. In a server-side Web application, MVC has to deal with the HTTP protocol, which is stateless: there is no open communication between client and server (a new connection has to be established with each client request). Thus it works as follows: the application receives an HTTP request (through a Web server), and it has to interpret the resource request based on the URL. When the resource is identified (by a Routing component), the Controller takes control: it may access the data source through the Model (if needed), it generates HTML using the View component, and finally delivers the content to the client through an HTTP response. Figure 1.4 shows this approach.

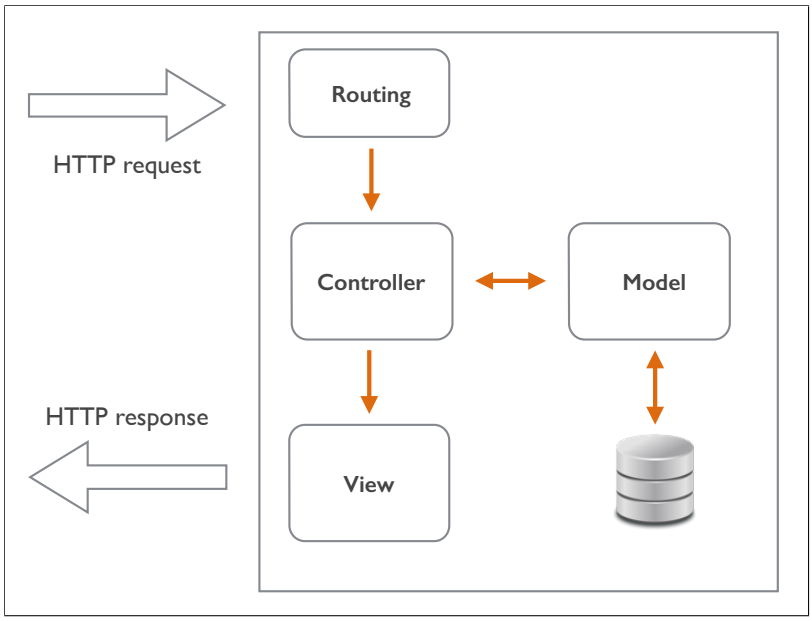

FIGURE 1.4. MVC server-side interaction. Application logic is in the server.

1.7.2. MV* concept

The JavaScript solutions to spaghetti code take the MVC design pattern (used at server-side applications) and move it to the client or frontend. This means that the server loses its original focus, and is now only used as an API (like it was explained above). The client, on the other hand, is now responsible for all MVC interaction. However, these solutions do not take the server-side MVC concept in the same way; they do some variations depending on what each solution believes it is a better approach. This means that the MVC design pattern is implemented with variations (MVP, MVVM, MVC-ish), so the design pattern they use is commonly refered to by the community as *MV** (Model-View-Anything) (Osmani, 2012, 2013). These concrete solutions come to life as JavaScript MV* frameworks, as they provide a complete and complex platform to build Web applications executed at the client-side.

The MV* flow is a bit different from server-side MVC: after the initial load of the application, there is a component (the "'*" part of MV*) listening to user interaction (through what is known in JavaScript as event bindings). When the user performs some action, this component changes the model if necessary (which may need to persist data to a remote server). The View component is observing Model changes, so if a Model is changed, the View changes the DOM with the corresponding updates. Figure 1.5 illustrates how this approach works.

There are many JS MV* frameworks developed by the Web community. Some people say that every week a new framework is introduced, but only a few are popular and widely used. Among those frameworks, there is Backbone.js, AngularJS, Ember.js and KnockoutJS (Synodinos, 2013). Each one of these frameworks has a different way to solve the mentioned issues, despite having one point in comon: produce an SPA as a result. The main features of each one are:

- Backbone.js: one of the first and most popular MV* solutions. It is considered more like a library (set of modules) than a framework. It implements MVC in a Controller-less way, in which the View takes some of this responsibility,

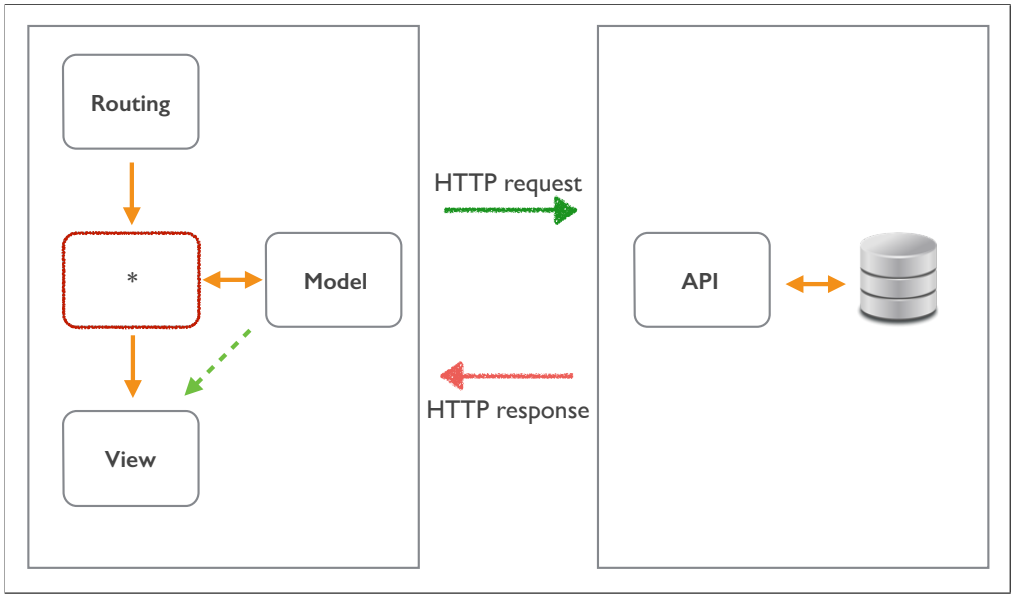

FIGURE 1.5. JavaScript MV* interaction. Server is used as an API and the application logic is in the client.

and the rest is up to the developer. This provides great flexibility, but when the application grows in scale and features, Backbone does not provide support for all possible scenarios (leading to additional programming). It is a low-level framework, which means that the developers need to write some tasks that could be more automated.

- AngularJS: one of the newest and most promising MV* frameworks. It is powered by Google, and its main characteristic is that it handles lots of the behavior of the application built directly in the HTML, as custom HTML attributes (it is extremely tied to this approach). This allows the developer to write less JavaScript code, making the development quicker than the other frameworks. It implements complex data handling (called data bindings), which allows the user interface to update automatically when the Model changes.
- Ember.js: as opposed to Backbone, Ember is a high-level MV* framework that simplifies a lot of work to the developer in order to ease code writing. It provides the possibility to implement complex tasks in a very easy way. However, this

leaves few space for the developers to implement their own solution. It provides data bindings, and a well-structured routing system.

- KnockoutJS: this MV* framework is similar to AngularJS in that both use HTML attributes to allow data bindings. The difference is that Knockout uses a Model-View-ViewModel (MVVM) approach, meaning there is a component called ViewModel that communicates with the View, informing of Model changes, but the View is responsible for updating the DOM.

As we can see, the complexity of these frameworks is given by how much the Model is connected to the View: some require the developer to update the DOM manually, but others do this task automatically through data bindings. It sounds like the latter case is better and more time-saving, but it comes with a cost, reducing the flexibility for the developers to write their own solution (Podila, 2013).

SPAs implemented with JS MV* frameworks reduce the gap between Web applications and desktop/native applications. This is even more important if the application is meant to be accessed through a mobile device (Hales, 2012), so the existence of these frameworks is relevant to the present and future of Web development.

1.8. Hypothesis

Until now, the discussion regarding the strengths and weaknesses of these new MV* frameworks is being marked by how much these solutions ease the code writing to the developer, how much more maintainable the resultant application is, the completeness of documentation and how much the community is involved in the development of the framework (Graziotin & Abrahamsson, 2013). The learning curve is another big issue considered a lot by developers (they want something easy to get into). Much discussion in forums and Q&A sites exists regarding these aforementioned issues, and the result is that developers tend to use the “fashionable” framework.

However, the concern of making an application run as fast as possible has not been addressed. Performance is not an issue inside all the existing discussion, perhaps because there are some well accepted myths about each framework. There is no detailed performance study that indicates the real impact of introducing all the application logic that the MV* frameworks do in the client, and this is not a minor issue. Only one study (Gizas, Christodoulou, & Papatheodorou, 2012) has been conducted regarding JavaScript frameworks performance, but it only considers non MV* solutions (like jQuery, Prototype, YUI).

In this research, there are two particular questions that we want to answer, and that can be crucial in choosing one or another MV* framework:

(i) Is there a cost in performance when using an MV* framework, compared to the case of not using any at all (this means using only jQuery)?

(ii) What is the real difference between a simple MV* framework and a more complex one? Is it possible to establish an outline so as to compare the different JS MV* frameworks?

1.9. Objectives

1.9.1. Overall objective

The overall objective of this research is to provide a study of the performance of Web applications built using the most popular JavaScript MV* frameworks. By providing this, the Web community would benefit with one more tool when choosing a JS MV* framework.

1.9.2. Specific objectives

(i) Provide quantitative (and empirical) data of the differences between the most popular JS MV* frameworks.

(ii) Evaluate performance of different components of each framework, using the same use case scenario for each one.

(iii) Provide a starting point from which more research can be done regarding JS MV* frameworks, especially focusing on performance.

2. CHAPTER 2: JAVASCRIPT MV* FRAMEWORKS FROM A PERFORMANCE POINT OF VIEW

2.1. Preamble

In the last decades the Web has been in constant evolution. The first Web applications of the 1990's were quite different from the typical modern application. For years the focus was based upon the fact that the majority of the content was processed at the backend, with static visuals which could only be modified by means of an explicit action of the user, which would then generate a request to the server. When Ajax appeared in 2005 (Garrett, 2005), a paradigm shift occurs towards a greater delegation of responsibility to the client or frontend of the application. The jQuery framework started to become popular, not only for its support of asynchronous requests (Ajax), but also for facilitating the handling of the DOM. The Web applications began to look more like desktop applications, due to the fact that the user does not see the change of page when a click is made on a link, except that he sees that the content changes without him noticing that a request was made to the server. The last and current stage of this evolution of Web applications is represented by what is known as *Single page application* (*SPA*) (Takada, n.d.; Mesbah & van Deursen, 2007), a Web application made up of "logical pages" that are included in a single real page, that is loaded initially when the application starts up. The application makes requests for the resources it needs to the server, independently and unrelated to the explicit actions of the user. This architecture is especially popular in applications geared to be used mainly from mobile devices (smartphones, tablets).

However, as more and more code is transferred towards the client, the JavaScript component of the application becomes more significant (it is the only language that can be run by any standard browser). It is in this manner that the JavaScript code of a modern Web application has increased both in size as well as in complexity. In the first stage the jQuery framework supplied temporary relief to the developers, by facilitating not only the handling of the DOM but also the massive incorporation of Ajax. However, even with

jQuery, and as the application grows, the code becomes difficult to maintain and could easily take a shape of spaghetti code (SourceMaking, n.d.).

As a means of confronting this problem, a number of ideas have been put forth so as to better structure the code with the client and facilitate the maintainability task. Many of these initiatives involve taking the MVC paradigm (with certain variations) from the server to the client, which has generated the emergence of a series of JavaScript frameworks known as *MV** frameworks (Osmani, 2013, 2012). These frameworks implement the SPA concept, reinforcing the idea that the only interaction with an eventual server should be to obtain data, but not to handle routes, or to handle elements or neither the DOM, nor any type of application logic. These tasks would now be handled by the code in the client.

Although there are many MV* frameworks, only a few of them are frequently used. Among those we find *Backbone.js*, *AngularJS*, *Ember.js* and *KnockoutJS* (Synodinos, 2013). Each one has a different way to solve the above mentioned problems. Backbone, on the one hand, is rather a set of classes that facilitate things, and is very flexible for the developer to implement a solution on his own. Angular, Ember and Knockout, on the other hand, are more complex and complete frameworks, leaving less margin for the developer to implement an individual solution of his own, but considerably more robust in terms of the result that can be obtained (Podila, 2013).

It is clear that there is a tendency to provide a user experience similar to what is available with native applications. This is far more important if the application is used on a mobile device (Hales, 2012), due to the fact that the Web application comes into direct competition with similar native versions and also because, in this case, one can trust to a lesser degree on having a permanent connection with the server. To this effect and considering the growing use of the Web from mobile devices, the importance of these frameworks for developing applications for these scenarios is very significant.

Up until now, the discussion regarding the strengths and weaknesses of these MV* frameworks has been focused mainly on to what extent these environments foster and facilitate the development of an application, or if the application generated can be easily

maintained over time. Another aspect also normally considered is the learning curve associated with each one of them. In this manner, the developers have followed to some degree the "fashionable" framework. This is quite understandable, if we think that these are created as a result of the problem of maintainability of the software.

However since the objective is to produce a piece of software, making it run as fast as possible is an inevitable concern in the long term (Webb, 2012). We are not aware of studies about the real impact on performance associated with the use of these MV* frameworks. The only study carried out regarding performance, is a comparative study of some JavaScript frameworks (Gizas et al., 2012), but these are not part of the MV* family.

There are two interesting questions which can define the use of one or another framework, and have yet to be addressed: first, is there a cost in performance when using an MV* framework, with respect to not using any at all (only jQuery)? And secondly, what is the real difference between a simple MV* framework compared to a more complex one? Is it possible to establish a framework so as to compare the different MV* frameworks? These are the questions which motivated this research.

The rest of this chapter is organized as follows: in section 2.2 we present the methodology and tools we used, in section 2.3 we provide a short review of related research. In section 2.4 we describe the actual tests performed and the associated results, and two final sections (2.5 and 2.6) present analysis, discussion and conclusion.

2.2. Methodology

The growing tendency in Web applications towards SPAs has not gone unnoticed, and the Web community has gone to great lengths to initiate and guide the developers in the use of these frameworks. There are many introductory tutorials to the different frameworks, and these frequently allude to a "To-do" application which permits adding, modifying, eliminating as well as filtering different tasks.

In an effort to help the developers to compare the different MV* frameworks, A. Osmani and S. Sorhus created the project-application *TodoMVC* (Osmani & Sorhus, 2014),

which consists in a To-do application with the distinctive feature of implementation for the majority of the most popular existing frameworks, with the same functionality in each one. This creates a great setting to measure the performance of each framework under equal conditions. The existence of TodoMVC permitted us to have at our disposal a standard application with simple functionalities. Better yet, as the project is public and known to many, these different implementations have already been tested and validated. In order to have a more complete idea regarding the possible impact on performance of the different frameworks, a series of tests were designed to include in the best possible way all of the functionalities of TodoMVC, under different scenarios: starting with the initial loading up, to the insertion or elimination of groups of tasks. These tests were carried out on the corresponding versions of the standard application.

The different versions of the TodoMVC application were subject to two categories of tests: a first group where the goal was to search for parameters relating to measuring the initial load of the application, and a second group related to gathering data relating to the performance of the application in actual operation.

In order to carry out the tests for the first category, an application or *testbed* was put together (that we named *MVC JS Performance*[1]). This application uses the API of the *Webpagetest* tool (Meenan, 2014) which permits the automation of the test process. Webpagetest delivers a breakdown of the different requests associated with the loading of a URL, including values linked to a set of predetermined metrics both in an aggregated and non-aggregated way, which makes it the ideal tool for performance studies. In addition, Webpagetest can carry out these tests on actual physical machines, distributed in different locations around the world, permitting us to select both the browser and the location that will be used to execute these tests.

The second group of measurements involves performance evaluations of different use cases of the application. For this purpose the *PhantomJS* (Hidayat, 2014) tool was used,

[1]Github repository of the project available at: `https://github.com/sivicencio/mvc-js-perf`

which provides a headless browser that allows navigation to a certain URL to begin interacting with the DOM (which is known as *Page Automation*). This way, the different use cases can be run, and the execution times of each one can be recorded. This second group of tests addresses the particularities of an SPA after the initial loading; what subsequently occurs is handled mainly by the JavaScript code of the client. Considering in addition that, in the case of the selected application (TodoMVC), the source of data is the local storage of the browser (`localStorage` object), there are no further requests to access a data source from a remote server.

The data collected in these different tests was processed and visualized with the help of the statistical tool *R*.

2.3. Related work

In their work, Gizas et al. (2012) understand the importance of choosing a framework that suits the developer's needs, and also provides high quality code and good performance. The research evaluates 6 different JavaScript frameworks (ExtJS, Dojo, jQuery, MooTools, Prototype and YUI), in terms of quality, validation and performance. However, none of them provides an MV* architecture, but only some help in DOM and Ajax manipulation. In addition, the tests were designed to evaluate the internals of each framework and not the behavior in a real Web application context.

Graziotin and Abrahamsson (2013) reaffirm in their work that there is little research to help practitioners to select the most suitable JavaScript framework. Their paper is a call for action, and proposes a research design towards a comparative analysis framework of the different JavaScript MV* frameworks. This design extends the work of Gizas et al. (2012), adding a layer related to practitioners interests, in addition to the existing research layer (quality, validation, performance). The authors interviewed some frontend developers and they were recommended to perform measurements on the same software project implemented using different JavaScript frameworks, instead of measuring them

alone. They propose to perform these measurements (including performance) using the TodoMVC project, which is suitable for that end.

A few more papers also use the TodoMVC project from a comparison perspective. Petersson (2012) designed and implemented a JavaScript MV* framework called MinimaJS, which is suitable for lightweight SPAs. He performed some evaluations of the loading time of his framework, compared to the TodoMVC Backbone.js and Ember.js implementations. He also considered some use cases of the application to test his framework: add, toggle, remove, clear all, etc, but instead of measuring how much time it takes to perform each use case, he measured the line coverage (how many lines of code were interpreted during execution). Runeberg (2013) performed a study of the differences between two JavaScript MV* frameworks: Backbone.js and AngularJS. One chapter of his thesis covers performance issues, using the TodoMVC project. He performed some tests using PhantomJS for page automation, in a very similar way to what we did in this research. The difference is that he considered an extended use case (create 1000 to-do entries, mark them as complete and delete each one of them). The results showed that Backbone.js completed the test in 22% of the time it took to AngularJS to complete it. The same test was performed with 50 to-do entries, in which case AngularJS was still slower, but the difference was not significant, and finally with only 1 to-do entry, AngularJS outperformed Backbone.js. The author suggests that AngularJS is not as efficient as Backbone.js when it has to deal with multiple DOM elements. Each element is associated with a model, and that is the reason behind the increasing difference in time when more to-do entries are added. In this research we try to go further in the TodoMVC features, and measure each use case in a separate way.

Some comparisons have been made outside the academic context. Gómez (2013) compared the different TodoMVC implementations in terms of their complexity. He used several metrics, including Source Lines of Code, Cyclomatic complexity, Halstead complexity and Maintainability Index, showing that AngularJS, Ember.js and KnockoutJS are clearly better than the rest. Of course these measurements do not include performance metrics. Nolen (2013), on the other hand, created a library named Om, which takes a

different approach when it comes to data handling. He implemented the same TodoMVC application using this library, and showed some benchmarks, comparing this implementation with the TodoMVC Backbone.js one. The test includes creating, toggling and deleting 200 to-do entries. The differences in the time it takes to each framework to do the task are significant, showing that Om outperforms Backbone.js.

None of the related work mentioned above performs a deep comparison between MV* frameworks, in terms of their performance, and also none of them use a base point like the TodoMVC jQuery implementation, which is considered to be the MV* *frameworkless* case. This research takes those points into account, as the next section describes.

2.4. Tests performed

The tests were divided into two large groups: initial loading and interaction tests with the application. In the initial loading tests, we are not only interested in the loading time of the application, but also when the response arrives and when something begins to appear in the screen from the user's standpoint, among other things. In the interaction tests on the other hand, the focus lies in the time it takes to carry out the different tasks (use cases) independently of the initial loading of the application (which already occurred).

To execute the tests, implementations of TodoMVC were taken using four of the most popular frameworks currently in use (instances of TodoMVC): Backbone.js, Ember.js, AngularJS and KnockoutJS. In addition, in order to have a comparative base line, the same tests were conducted with the jQuery TodoMVC application. The comparison against an implementation based solely on jQuery is reasonable because that library was the alternative of choice until the appearance of the MV* frameworks.

As we mentioned previously, the statistical tool R was used to process the data. However, as the information in each case was different, the strategy used in each case was also different.

For the initial loading tests, the results were stored in a *PostgreSQL* database. A script using R was written with a series of functions which carry out queries to the database using

the RPostgreSQL package. The results of each query contain the value of the different metrics of Webpagetest by tuple, where each tuple corresponds to a run. With this data the average and standard deviation were calculated taking into account all the runs for each test. This information is represented in graphic form showing the results for each one of the TodoMVC instances.

In the case of the interaction tests, the results were stored in plain text files, where a header is present first and it is followed by a line for each run carried out. The structure of each line is *run number - framework – time*, where each piece of information is followed by a blank space. An R script was written, which reads each text file and, as in the previous case, calculates the average and standard deviation for all the runs.

Regarding the graphs produced for both groups of tests, the X axis shows the different instances relating to the different frameworks, whereas the Y axis represents the execution time expressed in milliseconds (ms).

In order to facilitate a quick comparison with the base line represented by the instance associated to jQuery, a second Y axis was incorporated on the right side to each one of the graphs. The results obtained for the jQuery instance received a value of 1 and the results obtained for the rest of the instances received values like, for example 2, which means that for that instance the time required to complete the task was 2 times that of the jQuery instance. The actual average values can be seen above each bar of the graphs, followed by the relative values, which are surrounded by parentheses.

2.4.1. Initial Loading Tests

These tests involve the initial loading of the TodoMVC application in its different instances and were carried out using MVC JS Performance and Webpagetest. This tool delivers two sets of results. The first one, named *first view*, has to do with accessing the application as a first time visitor (cache and cookies are empty). The second, named *repeat view*, is the same but without emptying anything, that is considering the cache and cookies of the browser. This facilitates access to known resources as jQuery, Backbone, Ember

or Angular libraries, among others. It represents what a user would see if he returned to the page after his first visit. These two sets of results add more weight to the information obtained, due to the fact that each one represents a completely different scenario.

Webpagetest requires some configuration prior to running a test: a server (location) and a browser. In our tests, the Webpagetest server used was located in Dulles, VA, USA, and the browser used was Google Chrome. In addition, 30 runs were carried out by framework, generating a set of total results of 150 different runs. Each one of the runs generated the following metrics (in milliseconds) both for the first view, as for the repeat view:

- Load time: time from the beginning of the initial navigation, up to the window load event.
- Start render time: time from the beginning of the initial navigation up to the first non-blank contents rendered by the browser.
- Speed Index: value associated to the speed with which the user sees the entire page displayed completely. It utilizes the virtual progress of the page (based upon the video frames captured) in order to calculate a score (represented by time, where less is better).

The results obtained in the case of first view are shown in figures 2.1, and 2.2.

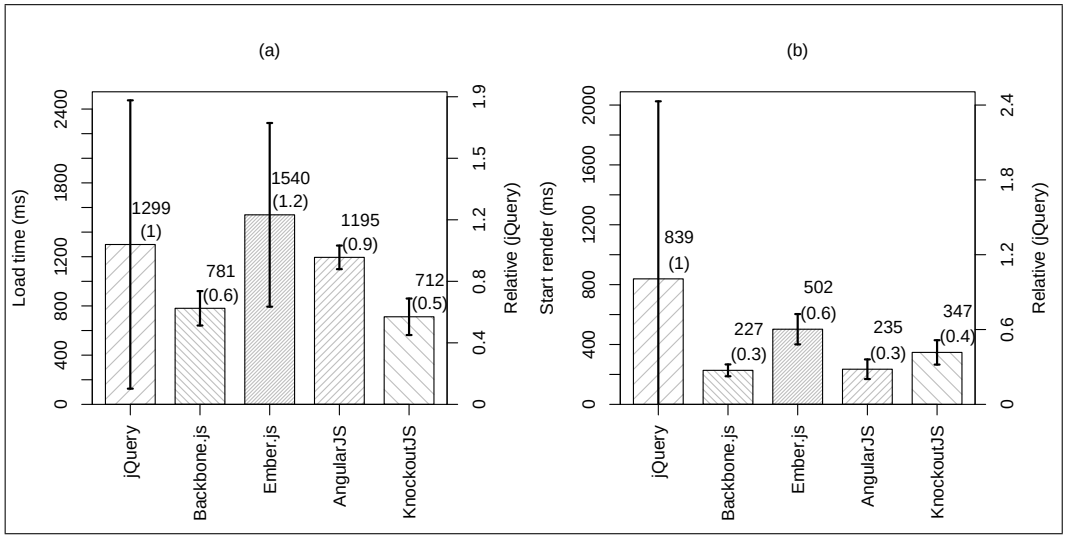

FIGURE 2.1. First view metrics: (a) Load time in ms and (b) start render time in ms.

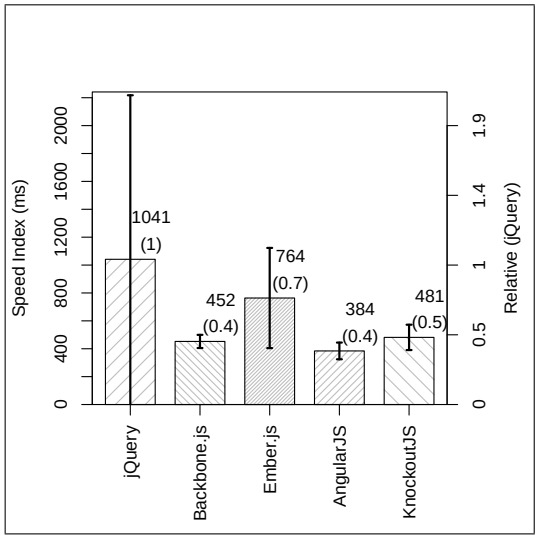

FIGURE 2.2. First view metric: Speed index in ms.

The results obtained in the case of the repeat view are shown in figures 2.3, and 2.4.

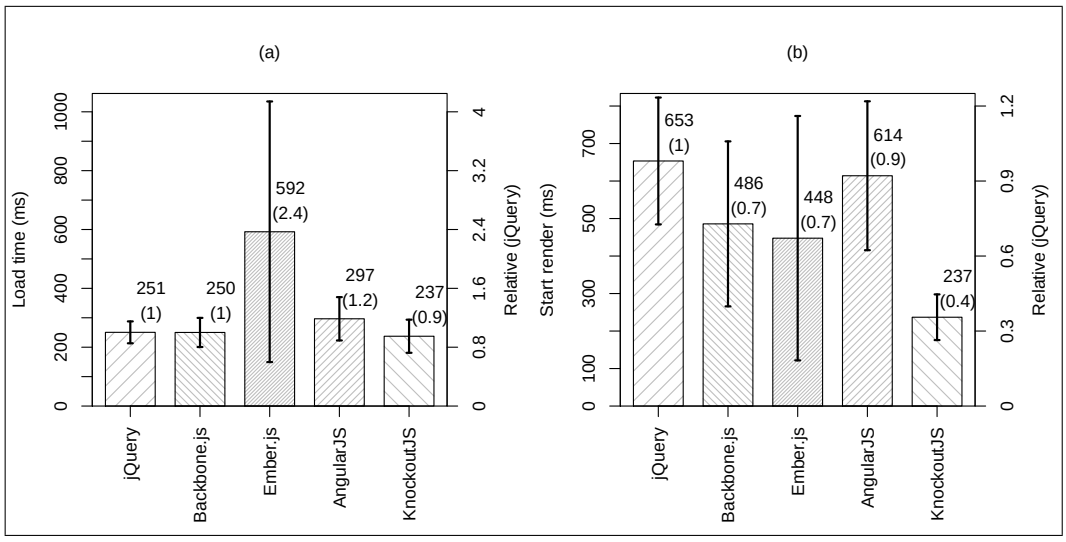

FIGURE 2.3. Repeat view metrics: (a) Load time in ms and (b) start render time in ms.

2.4.2. Interaction Tests with the Application

These tests consist in carrying out a series of actions within the application (associated with different use cases) immediately after the initial loading has taken place. Therefore, these tests do not consider the time that the application takes in loading up, but are only

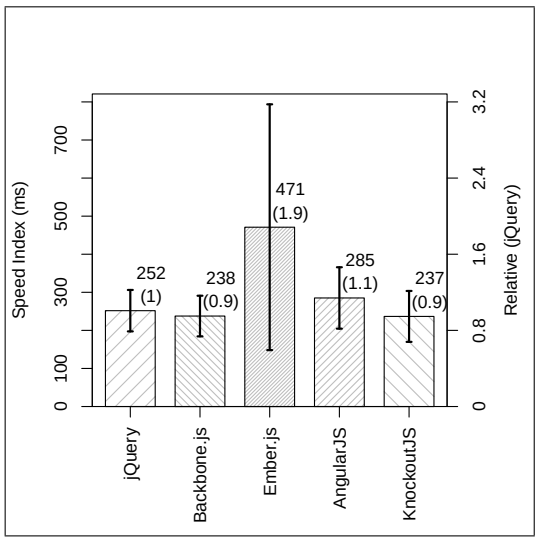

FIGURE 2.4. Repeat view metric: Speed index in ms.

concerned with what the application can do after the initial loading. As TodoMVC does not carry out requests to remote servers (it uses the `localStorage` object of the browser to persist data), it is possible to isolate from what is web traffic, without altering the behavior of the application. In other words, after the initial loading, the Web application becomes a bunch of JavaScript code that is executed according to the user actions, but not depending on external elements other than the browser being used.

In order to carry out actions within the application, a tool called *PhantomJS* was used, which is a headless browser (Webkit) capable of executing the same tasks a regular browser would, but without the graphic interface. We wrote a script that gets into the URL for each instance of TodoMVC, and carries out actions on the actual application. In order to accomplish this, jQuery was used with the purpose of being able to manipulate the DOM. For each use case considered, we recorded the time immediately before triggering the functionality in question, and the time immediately after the function was executed (checking that the result of the action was the one expected). The elapsed time (in milliseconds) was stored in a plain text file. As in the first set of tests, we carried out 30 runs for each use case for each framework. We present below each of the use cases and the results in graphic representation.

2.4.2.1. Adding a task

In the `#new-todo` input, the value "Example task" is set and then the event is triggered when the *ENTER* key is pressed. This causes the task to be added to the to-do list. We confirm that the task has effectively been added to the to-do list by checking the size of the list (which should be 1). Figure 2.5 shows the results obtained in this test.

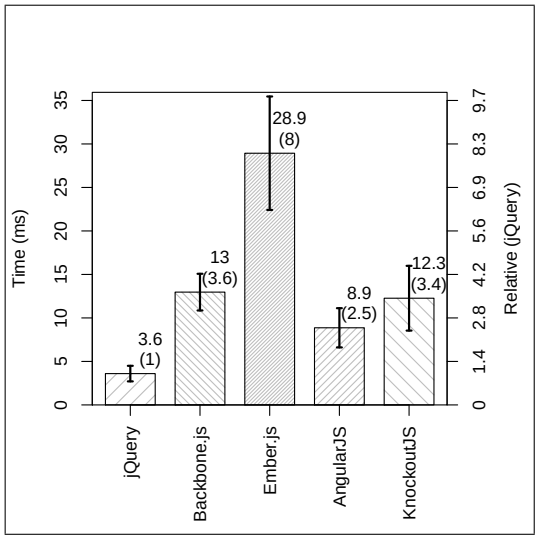

FIGURE 2.5. Adding one task.

2.4.2.2. Editing one previously added task

Assuming that a task was previously added, the event associated to perform a double click on the `#todo-list li label` element (which contains the text of the task) is triggered. This makes the `#todo-list li .edit` input visible, enabling the setting of the value "Example task edit" over this element. Then the blur event is triggered on it, which completes the editing of this task. We confirm the actual editing by verifying that the old text of the `#todo-list li label` element is different from the new one. The results can be seen in figure 2.6.

2.4.2.3. Deleting one previously added task

Considering that a task was previously added, the event associated to clicking on the `#todo-list li button` element (the cross key to eliminate a task) is triggered,

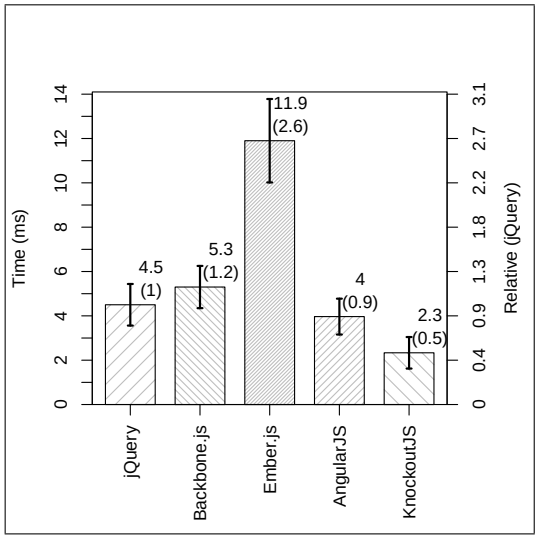

FIGURE 2.6. Editing one previously added task.

which in fact eliminates the existing task. In order to confirm that this task has been actually removed we check the size of the to-do list (which should be 0). See figure 2.7.

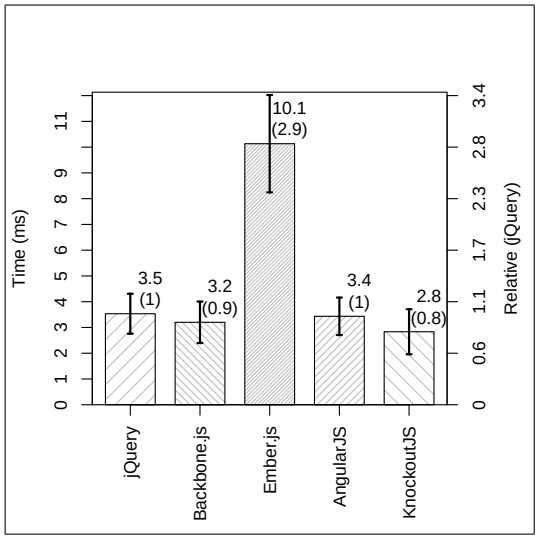

FIGURE 2.7. Deleting one previously added task.

2.4.2.4. Adding a group of tasks

This test is very similar to the test explained in 2.4.2.1, only that instead of adding one task, a group of tasks is added (set the value of the input and afterwards trigger the event by pressing the *ENTER* key). This was done first for a group of 10 tasks and later for 100

tasks. Confirmation that this was successful is obtained by checking the size of the to-do list (which should be 10 or 100 as the case may be). The corresponding results are shown in figure 2.8.

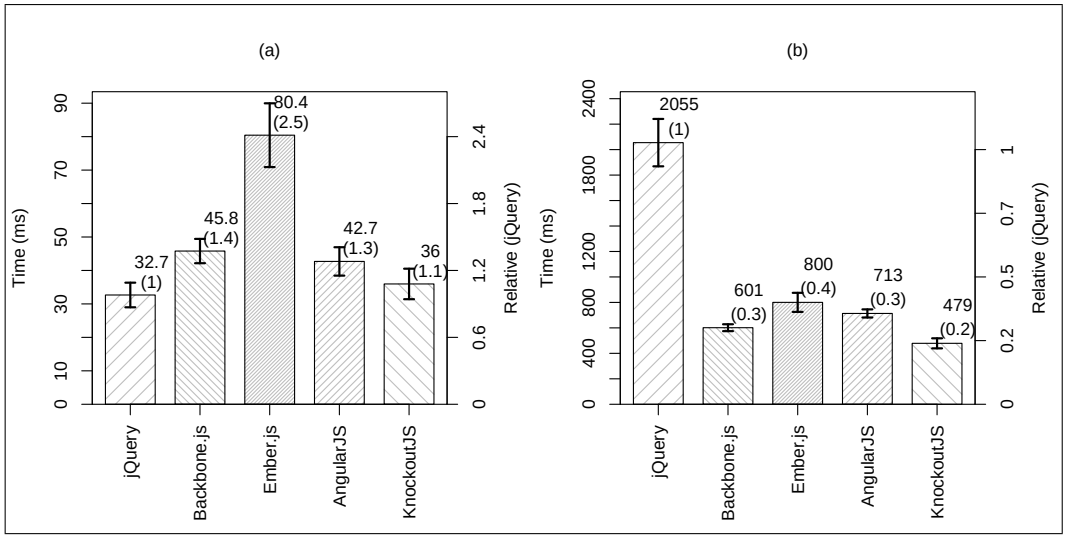

FIGURE 2.8. Adding multiple tasks, where (a) shows the case of adding 10 tasks, and (b) 100 tasks.

2.4.2.5. Deleting a group of tasks

Asuming that a group of tasks was previously added, we delete them in the following manner: we trigger the event associated to clicking over the `#toggle-all` element, which causes all of the tasks to be selected. Following this, we trigger the event associated to clicking over the `#clear-completed` element, which sets off the mass elimination of the selected tasks. We confirm that these tasks were actually removed by checking the size of the to-do list (which should be 0). This test was carried out twice, once with 10 tasks, and later with 100 tasks. These results are outlined in figure 2.9.

2.5. Analysis and Discussion

The results obtained in the initial loading tests show that, as we had suspected, Ember appears to be slightly "heavier" in comparison with the rest. However, in light of the results shown in the tests we do not observe an important or significant penalization with

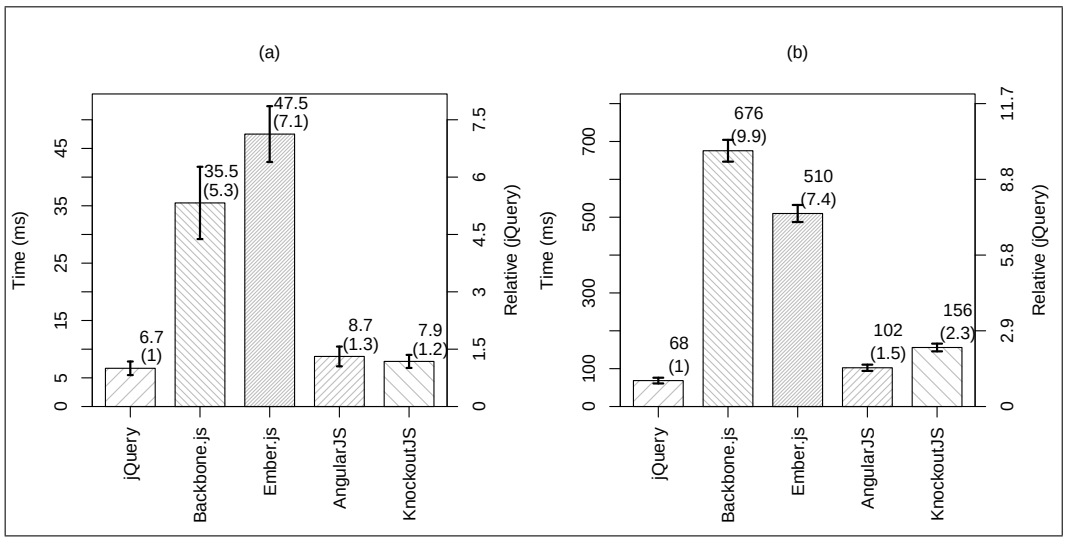

FIGURE 2.9. Deleting multiple tasks, where (a) shows the case of deleting 10 tasks, and (b) 100 tasks.

regard to the use of any of these frameworks. In addition, if we consider the beginning of the rendering process of the page, the instances associated with the different frameworks improve with regard to the jQuery base line. Finally regarding the speed index in the use of an MV* framework, which in our opinion is more relevant because it relates directly to user perception, not only does it not worsen, but it rather produces an application which is perceived as faster (with the only exception of Ember relating to the repeat view set of results).

The interaction tests reinforce the idea that there is no significant cost in performance when programming the application based upon one of these frameworks, as compared to doing it simply using the jQuery library. In this case, however, it is necessary to consider some interesting aspects. First, the action of adding one task resulted in it being much slower in all of the instances associated with the frameworks, in some cases up to 8 times slower with Ember than with jQuery. Once again, the trend repeats itself in that Ember shows performance somewhat slower than the rest. But more significant than this is to observe what occurs if we repeat the action 10 or 100 times; these numbers now end up being much closer to those of the base instance. In effect, for the base instance the additional time associated to adding 10 tasks is close to 10 times the one we had with

one task, while in almost all the other instances it only increased 3 times or less. These results suggest that the variations may be product of the strategy used by the programmer who generated the corresponding application, than to an intrinsic factor associated with the actual framework.

Finally, in comparative terms, the results obtained confirm to some degree our initial suspicions with regards to Backbone and Ember, in that these frameworks should be on the opposite performance spectrums. This is because Backbone is a low-level framework where the programmer has greater control over the code. Ember is a framework that operates at a higher abstraction level which facilitates programming as well as its maintainability, but which generally carries with it a higher cost in terms of performance. The Angular and Knockout frameworks showed fairly similar results in our performance tests and it is interesting to note that in many cases they are better than those of Backbone, in spite of both being frameworks which operate at a much higher abstraction level.

2.6. Conclusion and future work

This new architecture of Web applications where the client takes on the main role thereby reducing the interaction with the server to mainly services of synchronization of data, has arrived and is here to stay. The ample popularity and growing use of the Web from mobile devices only accentuates this tendency and demands that the applications be designed as SPAs.

Some years ago the MVC architecture was established as a paradigm which allowed the arrangement of code of a Web application in a manner in which it could be maintained over time. Since then, several frameworks have emerged which became popular because they greatly facilitated development work and forced this type of architecture. What is occurring today is completely analogous: the surge and popularization of frameworks which facilitate development and force architecture of better maintainability. The difference now is that the greatest part of the code lies in the client and must be expressed in the JavaScript language.

Few developers who have been involved in SPA type applications would doubt the virtues of these frameworks with regard to accelerating its development, facilitating its maintainability, and handling the complexity, etc. However, not many could answer questions related to how much cost in terms of performance this entails. The objective of this study had been to contribute to answering some questions in this area and in the process dispel some myths as well.

The more relevant findings, in our opinion, relate to demystifying affirmations of the type "if you need it to run fast don't use any framework". The results show that the use of these components in general does not generate a significant performance cost when compared to a library such as jQuery. Another "urban myth" is that if you are forced to use a framework, the one with the lowest level of abstraction is the one that would deliver the highest performance. Once again our findings show that this is not necessarily true, as we were able to verify that the Angular and Knockout frameworks produce better results than Backbone.

The availability of several instances of the same application (TodoMVC) allowed us to isolate variables that, in general, make it difficult to evaluate the performance of tools or software products. However, the application is too simple, meaning that it is possible that it does not totally represent the family of applications that concern us.

In our opinion there are two interesting lines of work in the future. First, it would be interesting to carry out a similar study with a more complex, or "real" application. This has the difficulty of writing several high quality applications so as to eliminate other factors. It would also be interesting to consider not only a more complex application, but also different categories of applications, since two applications may be similar in terms of complexity, but serve completely different purposes (which means using different framework features). Secondly, we believe that, given the fact we have access to the source code of each one of these frameworks, we should design "white box" tests (altering frameworks source code) that could be used to highlight the weaknesses of each one of these frameworks and explain the reasons for their good or bad performance in specific situations.

3. CHAPTER 3: CONCLUSION AND FUTURE RESEARCH

3.1. Review of the Results and General Remarks

The main motivation of this research was to generate quantitative knowledge that allowed us to contribute with a study, from a performance perspective, on the decision of whether or not to incorporate a JavaScript MV* framework inside the client code of a modern Web application. The idea was to answer questions like:

- Is there a significant performance cost when using a MV* framework in an SPA (independently of the development and maintainability advantages)?
- Is there significant performance differences between the most popular MV* frameworks?
- If a developer uses a low-level framework (like Backbone.js), will this translate in performance advantages comparing it to using a high-level abstraction one (like Ember.js)?

We believe that this research allowed us to answer those questions satisfactorily. First, although in general terms the associated framework code translates in a slower first loading time, it does not appear to be a significant cost in performance related to include or not a framework in a Web application. This is especially evident when analyzing the results of the repetitive interaction tests carried out in this research. Secondly, we can say that there are indeed differences in performance between the different MV* frameworks, mainly because in the majority of the tests carried out, AngularJS and Knockout JS showed better numbers than the rest. However, these differences are not quite significant and it is possible that they depend greatly on what kind of application is been considered. Finally, the answer to the third question is clearly negative. In several of the tests carried out, the Backbone framework is not better that some of the high-level ones, and even appears as a disadvantage with respect to the others.

3.2. Value of the research

The contribution of this research can be divided into three different areas:

(i) Provide quantitative data that allows the resolution of doubts in terms of performance at the moment of choosing what MV* framework to use.

(ii) Emphasize the value of having a "canonical" application available, written in several languages/platforms/paradigms and validated by the Web community. This is not a common case in the software field, and that is why we were very lucky to count with the TodoMVC project. This project has a completely different goal, which is related to allow the comparison of the different frameworks in terms of development patterns, but it was fundamental when we needed to compare frameworks between each other in terms of performance.

(iii) Contribute with tools, methodologies ahd measurement processes for other research related to Web application performance.

3.3. Future Research Topics

This research was a starting point for analyzing performance of the different JavaScript MV* frameworks. Much work can be done from here. It would be a good idea to explore if the obtained results can be extrapolated to real Web applications (which used to be much bigger than TodoMVC). This represents a big challenge, because it is necessary not to just include an important programming effort (real application instances would need to be generated), but also isolate variables like the expertise degree and skills of the developer when using a certain framework. This work would require one to also consider different categories of applications, because two applications may seem similar from a complexity perspective, but serve two absolutely different purposes, implying the need to use different features of the framework.

There is also another type of work that can be done: design "white box" tests, where each framework can be subject to special scenarios (favorable or not), trying to force them

to limit situations (and measure all of them). This could be important to understand in a better and deeper way the detailed reasons of performance costs that each framework exhibits.

3.4. Future of JavaScript MV* frameworks

As has been previously mentioned, the more attractive scenarios for the use of SPAs involve Mobile Web. The penetration of smartphones and tablets is impressive even in the least developed countries and, more importantly, the use of mobile devices to access the Web is growing at a very high rate. Considering these factors, the need to incorporate a significant component of client code in a Web application is growing. There is also agreement within the community of software developers in that, in this kind of scenario, it is very hard to build the JavaScript component without the support/help of one of these MV* frameworks. Therefore, its use will continue to grow.

There is also research into new ways of doing things that MV* frameworks already include. For example, there is a recent framework named Om (Nolen, 2013), which handles data in an innovative way, showing much better performance than other known and popular MV* frameworks. It may be possible that future development go in this direction.

It is very possible that the dozens of frameworks available today will converge into a smaller number in the future, with these ones being used in the majority of Web applications.

References

Berners-Lee, T. (1989). *The original proposal of the www, htmlized.* Retrieved from http://www.w3.org/History/1989/proposal.html (Accessed 16-January-2014)

Borodescu, C. (2013). *Web Sites vs. Web Apps: What the experts think — VisionMobile.* Retrieved from http://www.visionmobile.com/blog/2013/07/web-sites-vs-web-apps-what-the-experts-think/ (Accessed 16-January-2014)

Docs, W. P. (2014). *the web standards model · concepts · WPD · WebPlatform.org.* Retrieved from http://docs.webplatform.org/wiki/concepts/internet_and_web/the_web_standards_model (Accessed 16-January-2014)

Garrett, J. J. (2005). *Ajax: A New Approach to Web Applications — Adaptive Path.* Retrieved from http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ (Accessed 16-January-2014)

Gelbmann, M. (2013). *Top 10 rising web technologies in 2012.* Retrieved from http://w3techs.com/blog/entry/top_10_rising_web_technologies_in_2012 (Accessed 17-January-2014)

Gizas, A. B., Christodoulou, S. P., & Papatheodorou, T. S. (2012, April). Comparative Evaluation of JavaScript Frameworks. In *Proceedings of the 21st international conference companion on world wide web* (p. 513—514). ACM.

Graziotin, D., & Abrahamsson, P. (2013). Making Sense Out of a Jungle of JavaScript Frameworks. In *Product-focused software process improvement* (pp. 334–337). Springer Berlin Heidelberg.

Gómez, R. (2013). *How Complex are TodoMVC Implementations.* Retrieved from http://blog.coderstats.net/todomvc-complexity/ (Accessed 16-January-2014)

Hales, W. (2012). *HTML5 and JavaScript Web Apps*. O'Reilly Media.

Hidayat, A. (2014). *Phantomjs.* Retrieved from http://phantomjs.org/ (Accessed 17-January-2014)

MDN. (2014). *XMLHttpRequest - Web API Interfaces — MDN.* Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest?redirectlocale=en-US&redirectslug=DOM%2FXMLHttpRequest (Accessed 16-January-2014)

Meenan, P. (2014). *WebPagetest - Website Performance and Optimization Test.* Retrieved from http://www.webpagetest.org/ (Accessed 17-January-2014)

Mesbah, A., & van Deursen, A. (2007, March). Migrating multi-page web applications to single-page Ajax interfaces. In *Software maintenance and reengineering, 2007. csmr '07. 11th european conference on* (pp. 181–190). IEEE.

MSDN. (2013). *About Native XMLHTTP (Internet Explorer).* Retrieved from http://msdn.microsoft.com/en-us/library/ms537505(vs.85).aspx (Accessed 16-January-2014)

Nolen, D. (2013). *The Future of JavaScript MVC Frameworks.* Retrieved from http://swannodette.github.io/2013/12/17/the-future-of-javascript-mvcs/ (Accessed 16-January-2014)

Osmani, A. (2012). *Review of JS Frameworks — Journey Through The JavaScript MVC Jungle — Smashing Coding.* Retrieved from http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/ (Accessed 17-January-2014)

Osmani, A. (2013). *Developing backbone.js applications*. O'Reilly Media.

Osmani, A., & Sorhus, S. (2014). *TodoMVC*. Retrieved from http://todomvc.com/ (Accessed 17-January-2014)

Owen, J. (2010). *Implicit Interfaces as a Dynamic Adaptation Strategy in Frameworks* (Master's thesis). Pontificia Universidad Católica de Chile.

Petersson, J. (2012). *Designing and implementing an architecture for single-page applications in JavaScript and HTML5* (Master's thesis). Department of Computer and Information Science, Software and Systems, Linköping University.

Podila, P. (2013). *Important Considerations When Building Single Page Web Apps — Nettuts+*. Retrieved from http://net.tutsplus.com/tutorials/javascript-ajax/important-considerations-when-building-single-page-web-apps/ (Accessed 17-January-2014)

Runeberg, J. (2013). *To-do with JavaScript MV* : A study into the differences between Backbone.js and AngularJS* (Degree thesis). Arcada University of Applied Sciences.

Shan, T. C., & Hua, W. W. (2006, October). Taxonomy of Java Web Application Frameworks. In *e-Business Engineering, 2006. ICEBE '06. IEEE International Conference on* (pp. 378–385). IEEE.

SourceMaking. (n.d.). *Spaghetti Code*. Retrieved from http://sourcemaking.com/antipatterns/spaghetti-code (Accessed 17-January-2014)

Synodinos, D. (2013). *Top JavaScript MVC Frameworks*. Retrieved from http://www.infoq.com/research/top-javascript-mvc-frameworks (Accessed 17-January-2014)

Takada, M. (n.d.). *Single page apps in depth*. Retrieved from http://singlepageappbook.com/index.html

W3C. (2012). *XMLHttpRequest.* Retrieved from http://www.w3.org/TR/XMLHttpRequest/ (Accessed 16-January-2014)

Webb, D. (2012). *Improving performance on twitter.com, The Twitter Engineering Blog*. Retrieved from https://blog.twitter.com/2012/improving-performance-twittercom (Accessed 13-January-2014)