PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

PERFORMANCE OF JAVASCRIPT FRAMEWORKS ON WEB SINGLE PAGE APPLICATIONS (SPA)

HANS FINDEL DAVILA

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 NAVON C.

Santiago de Chile, (January, 2015)

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

PERFORMANCE OF JAVASCRIPT FRAMEWORKS ON WEB SINGLE PAGE APPLICATIONS (SPA)

HANS FINDEL DAVILA

Members of the Committee:

JAIME NAVON C.

YADRAN ETEROVIC S.

LIUBOV DOMBROVSKAIA

IGNACIO CASAS

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, 2015)

To my family

ACKNOWLEDGEMENTS

I wish to acknowledge the valuable contributions of the numerous collaborators of the TodoMVC project for making this research possible. I would like to thank my advisor, Jaime Navon, for the constant dialogs that shaped the ideas behind this work. This work has been partially supported by the National Research Center for Integrated Natural Disaster Management CONICYT/FONDAP/15110017.

INDICE GENERAL

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ABSTRACT

The architecture of Web applications has evolved in the last few years. The need to provide a native-like quality user experience has forced developers to move code to the client side (JavaScript). The dramatic increase in size of the JavaScript code was addressed first with the help of powerful libraries (jQuery) and more recently with the help of JavaScript frameworks. But although nowadays most Web applications use these powerful JavaScript frameworks, there is not much information about the impact on performance that the inclusion of the additional code will produce. One possible reason is the lack of simple, flexible tools to test the application when it is actually running in a real browser. We developed a test framework and tools implemented as plugins for the most popular browsers that allow the developer to put different implementation options under test. We used our test environment to perform extensive test for 6 different implementations of a single page Web application and we found important performance differences across the tested frameworks. These differences vary between the last print of the application when starting and its performance while already loaded.

Keywords: SPA, RIA, JavaScript Framework, Performance

RESUMEN

La arquitectura de las aplicationes web ha evolucionado en los últimos años. La necesidad de proveer una mejor experiencia de usuario ha forzado a los desarrolladores a agregar más código en el lado del cliente (JavaScript). Este dramático aumento en el tamaño de los scripts se ha debido en gran parte a la inclusión de librerías que facilitan el trabajo (jQuery) y más recientemente por los frameworks JavaScript. A pesar de que un gran número de aplicaciones usan estas librerías y frameworks, no hay suficientes estudios sobre el impacto (causado por la inclusión de estos) en el performance de las aplicaciones. Una posible causa es la falta de herramientas sencillas y flexibles para realizar estas pruebas en navegadores (browsers) reales. Desarrollamos un framework de testeo y herramientas para dos navegadores populares que facilitan a los desarrolladores a probar distintas alternativas de implementación. Usamos nuestro ambiente de pruebas para realizar una comparación de 6 implementaciones de una misma aplicación Web, encontrando importantes diferencias entre los frameworks utilizados. Estas diferencias se distinguen entre el tiempo de carga de la aplicación y la velocidad con la que ejecutan las acciones una vez cargada la misma.

Keywords: SPA, RIA, JavaScript Framework, Performance

1. INTRODUCTION

1.1 History

The Web technologies have evolved rapidly during its short history. At first, Web applications were mostly read-only. With the so called Web 2.0 applications became more interactive and users could not only read but also add new information like posts in a blog or comments about a movie. The changes also involved more effects, visualization enhancements and new ways to interact with these applications thanks to JavaScipt and CSS.

For more than a decade every change and actualizations of the information required to refresh the browser and to load a new version of the page from the server (backend). This simple model changed in 2005 with AJAX that allows the application to send or to ask for information to the server asynchronously without reloading the browser. Then Javascript code in the client side quietly and asynchronously performs the required DOM modifications.

Higher demands on Javascript code and increasing demands of AJAX is responded by powerful Javascript libraries to facilitate the development of Web applications. In a short time most developers began to favor jQuery over all those libraries and it became inmensely popular. JQuery allows programmers easy access to AJAX operations and DOM modifications with a unique and browser-independent application programming interface (API).

The increasing demand for a better experience is responsible for the increasing amount of code handled in the client side by the browser. This enrichment of the Web applications, known as “Rich Internet Applications” (RIA), pushes the

develpment of many specialized libraries, for example libraries focused on client-side rendering (update the visible part of the Web applications) to make it easier to write an RIA.

Finally, we witnessed of the birth of the "Single Page Applications" (SPA), applications that can handle all the user operations (including navigation) without any server help to reload the application. This kind of applications is becoming very popular with the growth of the so called mobile Web, due to a better use of the network and faster feedback to the user.

1.2 Frameworks and browsers

SPAs have been favored by an increase of internet accesses through mobile devices, most of them Smartphones. When the application is intended to be used mostly by Smartphones it is even more important to reduce the interaction with the server. This is because the network usage tends to become a bottleneck (slow and/or expensive). Although more code is needed to start running the application, we do not require to reload after every interaction so the traffic can be greatly reduced in the short term. On the other hand, the application needs to load more code (libraries and frameworks) and then to parse it before being available to the user.

In the last years developers have been using and enjoying the advantages of application frameworks for development. Automatic compatibility, the ability to write a feature once and being available on every browser are only some of the gains involved in using a framework. Having a new layer of abstraction also accelerates the development process and the code tends to be more structured.

This makes it easier to find the code related to a certain feature favouring the maintainability and extensibility of the code. Additionally, around every (popular) framework a community grows that continues improving the source code and generates a knowledge base that help new programmers to use the framework.

In the design of each framework a distinct set of characteristics is emphasized or prioritized. The main decisions are related to ease of coding, task automatization, resources usage and performance. For the end user who has to use and interact with the application, performance is the most important aspect. The end user will favor an application that loads fast and does not feel slugish or janky, independently of the device or browser that is been used.

In the last years, the browsers have raced against each other in terms of performance. As we mentioned before, the increasing amount of code in the client side has made JavaScript engines a crucial aspect in the browsers development.

Modern browsers have evolved both in the way to manage and execute the code sent by the server as well as in the tools provided to Web applications developers. This translates into big advances not only with JavaScript engines up to 100 times faster than its predecesor and other multiple optimization strategies. The browsers have also allowed developers to extend their own features with a simple plug-in strategy, that has been used for many purposes.

Although browsers have increased their performance, an application with a lot of code to execute or simply a bad code on the client side, can be perceived as

slow by the end user. Most JavaScript frameworks, used to create SPAs, add a significant amount of code into the application so it is reasonable to wonder how this code impacts the performance of the Web application. This is a question the developers, mostly the code architects and designers, should ask themselves before choosing the framework for the project. But, in spite of it, many times the performance factor is of little or no relevance when making the decision.

1.3 This work

Considering that making decision without having information on the performance impact on the applications is today a common practice. This work attempts to provide a better support for the creation of performance oriented web applications. To this end the main contributions are:

- Provide tools that allow performance testing of SPAs through plug-ins for certain browsers.
- A comprehensive study on the relative performance of a simple application across some of the currently popular frameworks on the Web.

We designed and developed extensions (plug-ins) to measure the performance of SPAs for the Google Chrome and Mozilla Firefox browsers. These plug-ins allow the developers to make automation tests on Web applications while measuring the time required to perform each task and the resources used to achieve them. In order to validate this work, we perform and analyze a set of

tests over a defined application developed on different popular frameworks. With these tests we observe interesting results that were not evident beforehand.

2. PERFORMANCE OF JAVASCRIPT FRAMEWORKS ON WEB SINGLE PAGE APPLICATIONS (SPA)

The content of the following chapter corresponds to a paper, submitted for publication to the Journal of Web Engineering. This chapter is organized as follows. The first section gives an introduction to the investigation topic. In the second section we put our research in perspective by reviewing the relevant related work. Section 3 describes our test environment and tools used in the research. Section 4 describes the results obtained after using the test tools to measure relative performance of the most popular JavaScript frameworks. Finally, section 5 we provide a conclusion for this work.

2.1 Introduction

Web technologies have been evolving in an accelerated way. All browsers have embraced JavaScript as a unique scripting language, giving to this programming language a crucial role in the development of Web applications. JavaScript makes it possible to improve the user interaction and also allows masking the network latency [25] to provide a much better and fluid user experience, closer to the one of a native application [17].

The incorporation of asynchronous requests [38,40] through AJAX [3,27,32] has also contributed to improve the user experience. The popular jQuery library has simplified the use of AJAX, the manipulation of the DOM [23] and cross-browser compatibility; furthermore, it has also speeded up development [14,37].

The constant search for a better user experience and the rising of the mobile Web have increased the amount of code in the side of the client [21], giving birth first to rich Internet applications (RIA) and later to the single page applications (SPA) [19]. But this tendency to move most of the application code to the client, demands a better organization and structure of the JavaScript pieces to manage complexity and also to make the code more extensible.

In a classic Web application the code is structured around the MVC pattern. This architecture has many benefits and has been implemented in most development frameworks from Struts to Rails. It is no surprise then that the same MVC pattern could be useful to organize the JavaScript code at the client side as well [6].

In the same way that the use of a framework is most approppiate to force the MVC architecture on the server side. The use of a MVC framework on the client facilitates the organization and structure of the JavaScript code on the client side. Many of these frameworks have been developed and most of them follow some variation of the MVC pattern (MV*). Natural selection within the community has made many of them irrelevant but a few. For instance, Angular, Backbone and Ember have become really popular [16]. Lately, React has emerged, bringing new ideas to the frameworks discussion.

Each of these JavaScript frameworks uses a different approach to fulfill its goals. This has an impact on several relevant issues [34] including the overall performance [8] of the application. Nevertheless, this is seldom taken into account when deciding which frameworks should be used. Many studies have

shown that performance has a profound impact on end users. An application that takes a long time to load or a browser that apparently freezes are visible examples of performance issues.

A recent study [35] showed the need for more research in several relevant aspects of RIA, such as security, offline functionality, and performance. This last aspect is becoming more and more important with the increased use of mobile devices. The reason is that mobile devices are, in general, less powerful than a desktop or a laptop computer and, of course, the performance of the application depends on the machine where it runs.

In our work, we have developed a test environment that facilitates the application of performance tests on differret RIA or SPA. We have used it to conduct a wide range of tests to different implementations of the same single page application. Each implementation corresponded to a version of the application that was built using a different JavaScript framework. This strategy has allowed us to validate our test environment in a real scenario, and on the other hand, to learn more about the performance behaviour [28] of the most popular frameworks.

2.2 **Related work**

Gizas et al. [1], exposed the relevance of careful choosing a JS framework. The research evaluated the quality, validation and performance of different JavaScript libraries/frameworks (ExtJS, Dojo, jQuery, MooTools, Prototype and YUI). The quality was expressed in terms of size, complexity and maintainability. The performance tests corresponded to measurements of the

execution time of the framework with the SlickSpeed Selectors test framework. These tests however, are designed to evaluate the internals of the libraries themselves, and do not mix with the application built upon. Additionally, none of these libraries provide an architectural context to develop an application, they only help in accessing the DOM and in communicating through AJAX calls.

Graziotin et al. [14] extends Gizas work by proposing a comparative analysis framework of JavaScript MV* frameworks that can help practitioners to select the most suitable one. To this end, the authors interviewed front-end developers to get first hand opinions on the relevant aspects.

A recent research work by Vicencio et al. [41] focused on the relative performance of client side frameworks. They measured the time it takes the application to load and to render the page in the browser, and the time it takes the application to execute a given action on the user interface. The results compared some well known frameworks using the TodoMVC application as a basis. They used existing test tools such as PhantomJS [33] and Webpagetest [2]. Because they use these tools, the performance measures could not be taken with the application running in a real browser.

In the same line of Vicencio, Petterson [29] compares a tiny framework called MinimaJS to Backbone and Ember and Runeberg [30] performs a comparative study between Backbone and Angular.

There are few comparative studies on frameworks for the mobile Web. Heitkötter [24] elaborates a set of evaluation criteria for converting Web

applications into apps for the different mobile operative systems. This work could be used to reduce the constant code downloading from mobile devices.

Nolen [15] was interested in relative performance of Backbone.js so he created a new library named Om, which takes a different approach when it comes to data handling. He implemented the TodoMVC application using this library, and compared this implementation with the Backbone.js one [5]. The test included creating, toggling and deleting 200 to-do entries and they showed that Om clearly outperformed Backbone.js.

Further comparisons among frameworks or even optimizations in code development techniques with a given framework are performed in various environments and can be found on the Web [11,12,13,18,20,22,36,42]. In particular, the filament group [26] published a consistent research comparing the major frameworks (Angular, Backbone and Ember) over three distinct environments (devices) including the download time for each application. They only researched the time required by the application to be ready. The results are consistent with ours.

We are not aware of any flexible test environment that could be used to measure a SPA running in a regular modern browser. Perhaps for the same reason there are not extensive performance studies of performance of a SPA running under the dominant JavaScript frameworks.

2.3 Test environment

2.3.1 General aspects

On the previous section we have mentioned the work carried on with the help of PageSpeed and PhantomJS. PageSpeed is used to measure the loading time of the application whereas PhantomJS, allows the user to program an interaction with the application (page automation test). These tools, however, have some important drawbacks. The main problem is that they rely on an artificial headless webkit-based browser to perform the tests and not the real browsers where the application will run at the end. Another limitation is that they are non-extensible, so it is very hard to modify or add new metrics.

We designed and implemented a browser-specific test environment for the Google Chrome and Mozilla Firefox browsers. The idea behind these tools is to create a simple interface to run automation tests over defined SPAs, to measure the application performance. Because there is a common input/output [appendix 1 and 2] format it is easy to compare across different implementations.

The tools can measure different aspects of the application. There is a static analysis for the DOM in terms of the dependencies and number of elements, but the tools can also perform dynamic analysis by capturing metrics such as time required to load, time to accomplish each task of the given set. They can also measure resources consumed by the browser: CPU usage, RAM usage and Network usage (downloads and uploads during the test).

2.3.2 Google Chrome implementation

In Chrome, a plugin can distribute its code into different structures of the browser. In this case, we used a main tab for the plugin itself, with its own HTML and JavaScript, but the extension also adds scripts on the tabs used by the user to add or modify features of the websites he visits. The main tab, that we call configuration tab, and the tabs used by the user to browse the Web can communicate with each other through their scripts.

Through the main tab we have access to special browser resources such as CPU, RAM and network used by every tab. So we placed here a script to monitor the resources consumed by the tabs and a file input form to upload the automation test in JSON format. On the navigation tabs we placed a listener and the scripts required to execute tasks and analyse the DOM.

The usage-flow of the plugin is as follows. Once the file is uploaded in the configuration tab, it opens a new tab in the given URL. The main tab starts monitoring the resources usage of the created tab and then sends the list of tasks to perform. The tab executes each task measuring the time required to complete it while the configuration tab measures the resources consumption of that tab. When all the tasks are completed, the browsing tab (the one executing the tasks) records the relevant aspects from the loading process and the characteristics of the DOM structure. Then the tab proceeds to send all this information back to the configuration tab. The configuration tab aggregates the resources consumption and downloads the data in a standard JSON format in the computer (see Figure 2.1)

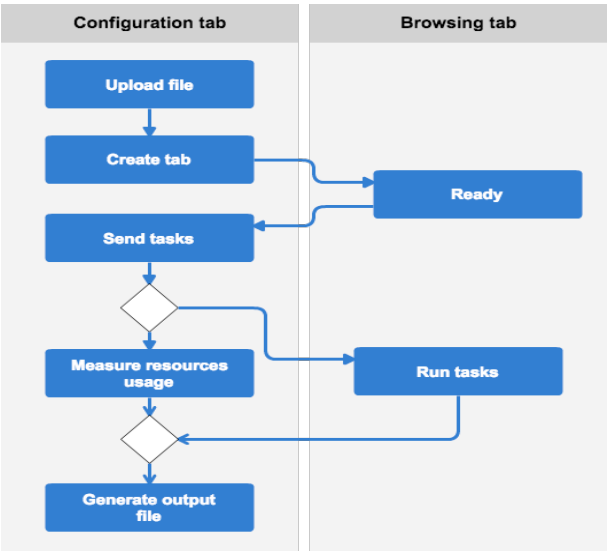

Figure 2.1: Chrome plugin flow

2.3.3 Mozilla Firefox Implementation

The Mozilla Firefox plugin is straightforward (Figure 2.2). Since in this browser the interaction across tabs is less flexible, we decided instead to extend the browser internal API. We leveraged on an existing low-level extension to provide us with resources consumption.

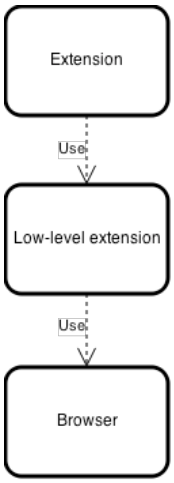

Figure 2.2: Mozilla Firefox plugin structure

This plugin adds a little HTML fragment before the code of the visited pages. Within the plugins HTML fragment is a file input where the JSON document can be uploaded. Then the browser verifies it matches the current URL and parses the instructions. It starts monitoring the resources consumed by the browser and executes the instructions given in the JSON file. Once it is ready with the instructions it collects the information about the loading process and then proceeds to download the aggregated data into the users computer.

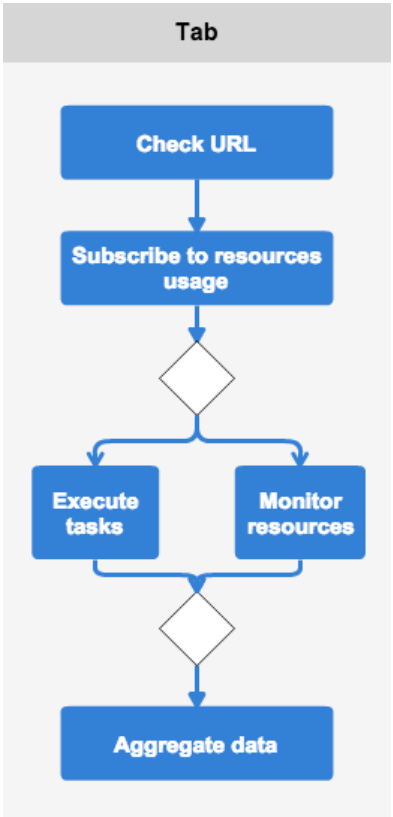

Figure 2.3: Firefox plugin structure usage

2.3.4 Shared code

The shared scripts among the plugins correspond to the execution of the automation test itself, the timing of the loading process and the basic analysis of the DOM.

The time measurement of the tasks is performed as follows. The tab receives an array of tasks to execute as JSON objects containing an action name, target and optionally other parameters. Then the algorithm traverses this array and for each element the algorithm translates the JSON object into an executable action. Before each instruction the tab stores the timestamp, registering the start time of the instruction. After completing the task the tab registers another timestamp marking the end of the instruction execution. The difference between these two values is the required time to execute the instruction. The tabs stores these values in an array with the same order of the instruction set, so the data can be directly inferred and processed when processing the information.

The loading times are saved by default by the browser and can be accessed through the 'window.performance.timing' object [10, 39]. This shared script leverages on this and stores some relevant values from it [9] , but could be edited to consider other timing metrics [31]. The DOM static analysis is performed by scanning the structure of the DOM and trying to classify the source of the external resources and the count of elements in it (the DOM) [23].

2.4 Methods

2.4.1 The application

We wanted to compare the dominant JavaScript frameworks in terms of performance; but we were interested not in the isolated or intrinsic performance of these pieces of software, but rather on how fast an application built with them would run. To this end we would have to build the same application several times using in each instance a different framework. But if we had done that we could have introduced a bias related to our relative expertise on a particular framework.

Fortunately, A. Osmani and S. Sorhus created an open source project called TodoMVC [4] where you can find the same task manager application implemented in almost every existent framework. Since this is an open-source project the experts in each of those communities produce and update the respective code version of the application, it is sound to assume that it is close to the best possible implementation in each case.

2.4.2 The Frameworks

We selected the following 5 frameworks: Angular, Backbone, Ember, Marionette and React. The first three frameworks are among the most popular ones, with strong development communities. Marionette is a framework that operates over Backbone and it was included to match the amount of features provided by the first three. React is backed by Facebook has been raising a lot of attention lately and brings a few new ideas to the development environment [7]. Finally, we also included a version of the application built just with jQuery (no

framework) to use it as a baseline. This is reasonable since in most cases the alternative to the use of a JavaScript framework is not using nothing, but using just the ubiquitous jQuery library.

The table 2. 1 presents a summary of the tested frameworks. The word “base” in the rendering engine column indicates that the framework uses its own self-made engine. In the case of Angular there is no rendering engine and uses an approach of extending the HTML (as a DSL). The persistent storage column indicates the structure used to manage the local storage (in browser database), which is either an array or a hash. This may have important performance implications.

	Dependencies	Render engine	Persistent storage
Angular	-	-	Array
Backbone	jQuery, Underscore	“base”	Hash
Ember	jQuery	Handlebars	Array
jQuery	-	Handlebars	Array
Marionette	jQuery, Underscore, Backbone	“base”	Hash
React	-	“base”	Array

Table 2.1: JavaScript Frameworks tested

2.4.3 The Tests

In order to inspect different aspects on the application’ implementations, we performed a set of tests that basically inserted and deleted tasks into or from the list. The previously described automation test suite executed these tests automatically.

We executed each of these tests in each of the 6 instances of the application and for both browsers (Chrome and Firefox). Each automation test was performed

five times in order to validate the results of them. All of this results in a large list of experiment results. The results and discussion of these tests are shown in the fifth section.

2.4.3.1 Add 1000 tasks to the To Do list

We carried on this test by creating a test that, starting from a clean state, submits one thousand times a new task through the tested application implementation. In short, the test consists in a large set of instructions that the test suite executes sequentially. For each word (task text value) in a one thousand long list, the instructions are to write the value in the task forms input and then to submit the form. After submitting the form, the application performs its own process updating its internal data and the DOM. This process translates into the application inserting the given task values in the same order in the task list, moving from an empty list to a one thousand task list.

2.4.3.2 Delete all the tasks one by one

This test simulates the user clicking on the button with the class “destroy” (HTML class, selectable by “.destroy”) within every task element. This results in the application deleting each task one at the time, starting from the first task until there are no more elements on the list.

The full cycle of the test starts with the automation tool clicking the delete button. This click triggers the tested application to identify the task and remove the task from its internal memory representation (including the in-browser database) and notify the DOM to do the same. After the application has

performed this triggered reaction, the control returns to the automation tool, which moves to the next instruction. All the instructions in this automation test were to delete the first task in the list, so it deletes every task when it is on top of the list.

2.4.3.3 Incremental behaviour to add 5000 tasks to the list

In general, adding a new task to the list takes more time on a list that already has thousands of items. We wanted to test how each of the application' implementation instances responded to an incremental load of the to do list from 0 to 5000.

In order to do this test, we copied five times the instruction set of the first test. This means that the applications inserted every element on the list and started again, repeating the task values every one thousand elements.

2.5 Results

2.5.1 Compairing runs within each framework

The first result analysis consisted on five consecutive runs of creating one thousand (1.000) records with a task-cleaning step before each run. This means that each time the browser started with no previous data and automatically wrote and submitted words as tasks in the applications form. The results for the Angular instance of the application and the Chrome browser are presented in Figure 2.4. The values include both the submitting of the form with the new item and the process to insert it into the task list.

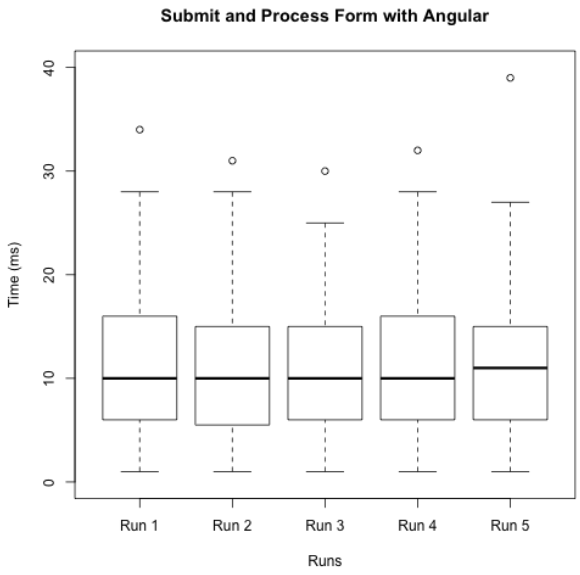

Figure 2.4: Insert new item (submit and process)

As Figure 2.4 shows, we found that although the time needed to process each item may vary, the distribution in each of the 5 runs is very similar with average around the 10 milliseconds. The same was true for every framework and browser.

A similar analysis, but with the focus on RAM usage was also made and we again observed numbers that were quite similar among the different runs. Figure 2.5 shows the results for Angular and Chrome.

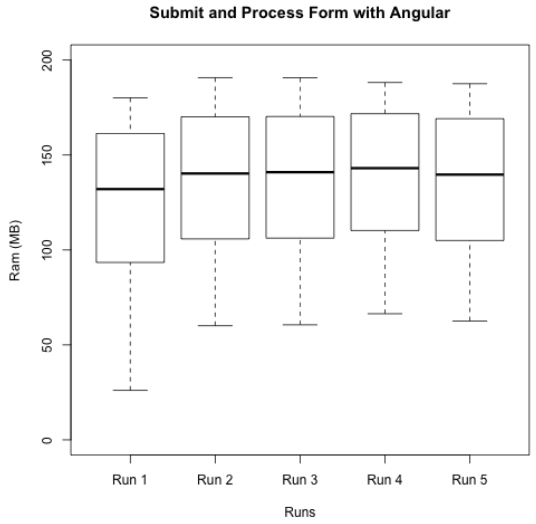

Figure 2.5: RAM usage (submit and process)

2.5.2 Compairing runs within each framework

The second series of results allows us to compare the relative performance of the different implementations of the application running in each of the two browsers. For this purpose we aggregated the results of the runs obtaining bigger datasets.

We compare first the time for each application instance to complete the list of assigned tasks described in the automation test. Figure 2.6 shows the results for the two browsers.

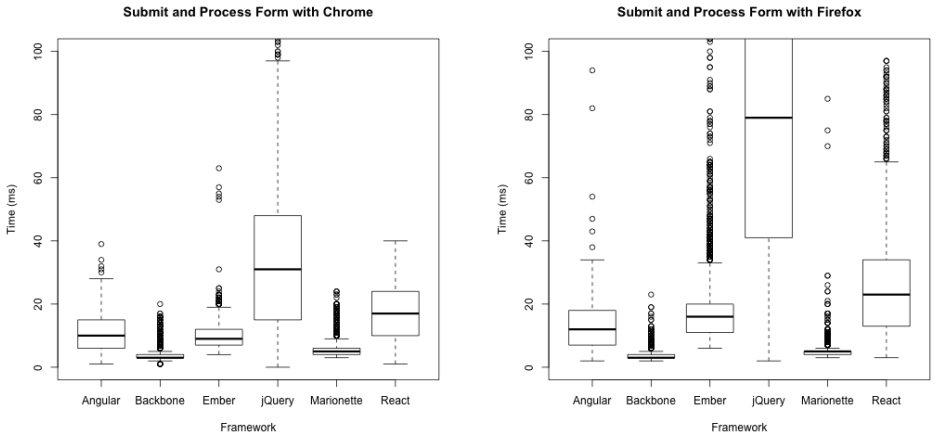

Figure 2.6: Time required to process a form submission by framework in Chrome and Firefox

Absolute comparison across browser is not completely fair since the mechanisms to obtain them are a little bit different. Times recorded for each framework in the same browser are comparable. In general, the application instances that were faster in Chrome were also faster in Firefox except for Ember y Angular whose relative order is swapped. As Figure 2.7 reveals, with respect to RAM usage the differences were important.

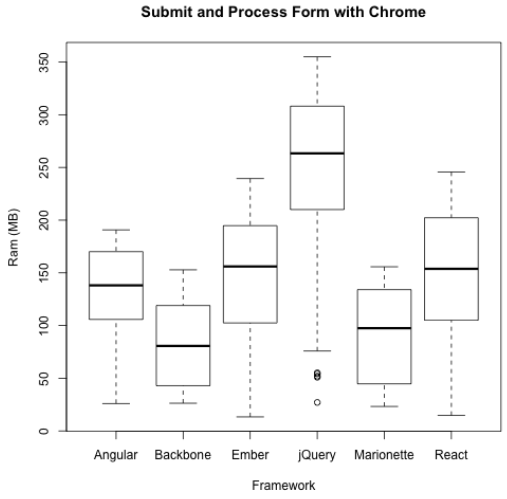

Figure 2.7: RAM usage required to submit and process form

The poor performance of jQuery may be related to the fact that it requires an event handler for each task. Memory usage depends mostly on the representations of the model instances within each framework and the event listeners on the DOM. The chart also reveals that the Backbone and Marionette instances use about half the memory of the other frameworks to work and the others are quite similar in terms of memory usage.

We repeated the same process with the second automation test. The one that deletes tasks from the list one by one by simulating a click on the button with the destroy class (HTML class, selectable by ".destroy") within every task element. The results are presented in Figure 2.8.

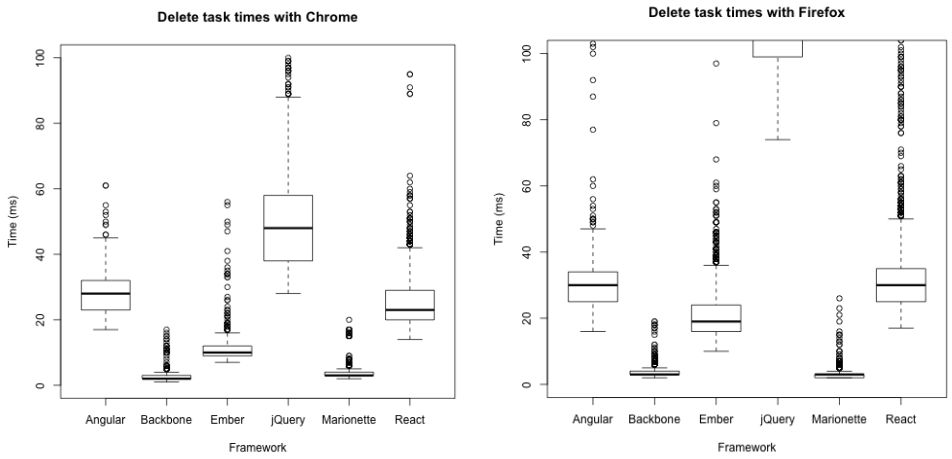

Figure 2.8: Time required to delete a task for Chrome and Firefox

In this case the relative order is also preserved across browsers except for a swap between Angular and React but the difference is very small.

The memory usage for this test is presented in Figure 2.9. The Backbone instance proved again to be the one with the lowest memory consumption but in this case the jQuery instance exhibits better results than in the previous test

whereas React performs worse. The React version may be suffering due to its virtual DOM optimizations to reduce the updates to the actual DOM.

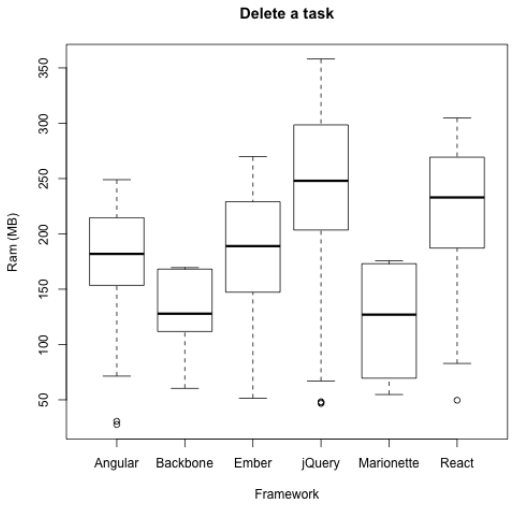

Figure 2.9: RAM usage required to delete a task

2.5.3 Comparing frameworks scalability

Our final tests involve increasing the amount of data to see how the different instances respond. Figure 2.10 presents the time needed to add a new task starting with an empty list and up to 5000 tasks in the list.

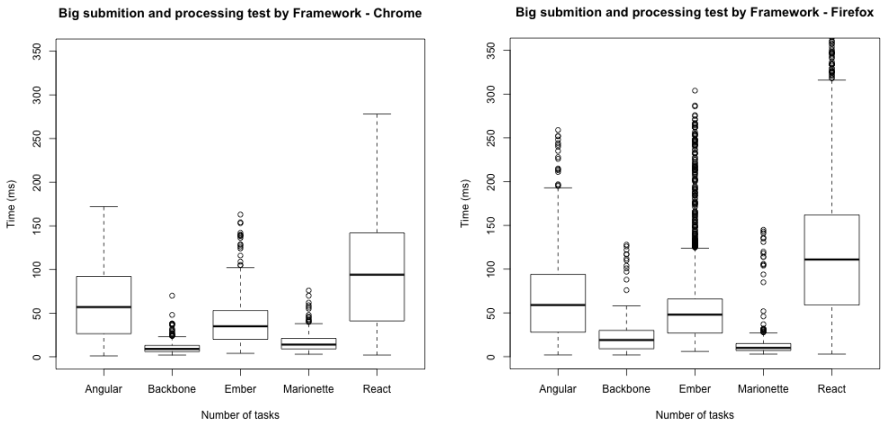

Figure 2.10: Time required completing new task addition starting from no previous data to 5.000 records (Chrome and Firefox)

As we can see, some frameworks perform vastly better and more consistently than others. In these charts we show the aggregated data so in general the lower ends of the boxes is associated to the time required to add the firsts tasks, while the upper ones should correspond to the last insertions.

Figure 2.11 reveals allows us to visualize how the required time to insert a new task depends on the previously inserted ones. In these charts, the X-axis corresponds to the number of registered tasks in the application whereas the Y-axis is used for time required to add a new one. So the points near the left part of the chart are the tasks inserted when there were still few records on the list and the points to the right correspond to tasks added when the list included many tasks.

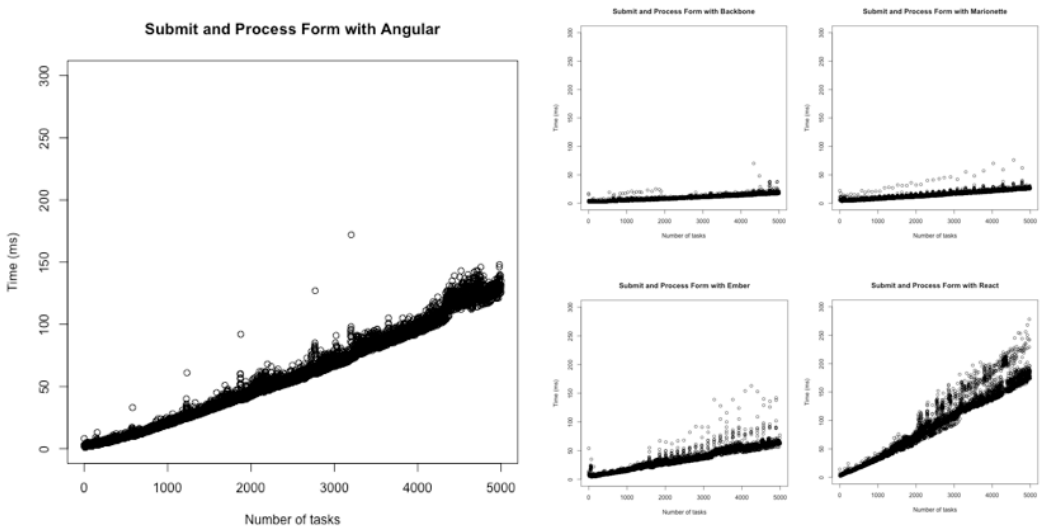

Figure 2.11: Progressive times required to add a task (Chrome)

As expected, the test revealed an almost linear behaviour. The starting point and the slope of these curves gives us more details about how do these frameworks behave when big amounts of data being inserted on the application. The two Backbone based frameworks (the two on the top right) show very flat curves, demonstrating that they can cope very well with the insertion of new information compared with the other frameworks.

Another interesting result related to the behaviour with and without data has to do with the load time. Figure 2.12 presents the range of load times for each instance of the application in a timeline chart. To the left is faster, so the bars start from the left when no previous data was on the application and finish on the right where one thousand (1.000) records are stored. The chart has vertical marks every one second. As usual we did this for Chrome and Firefox.

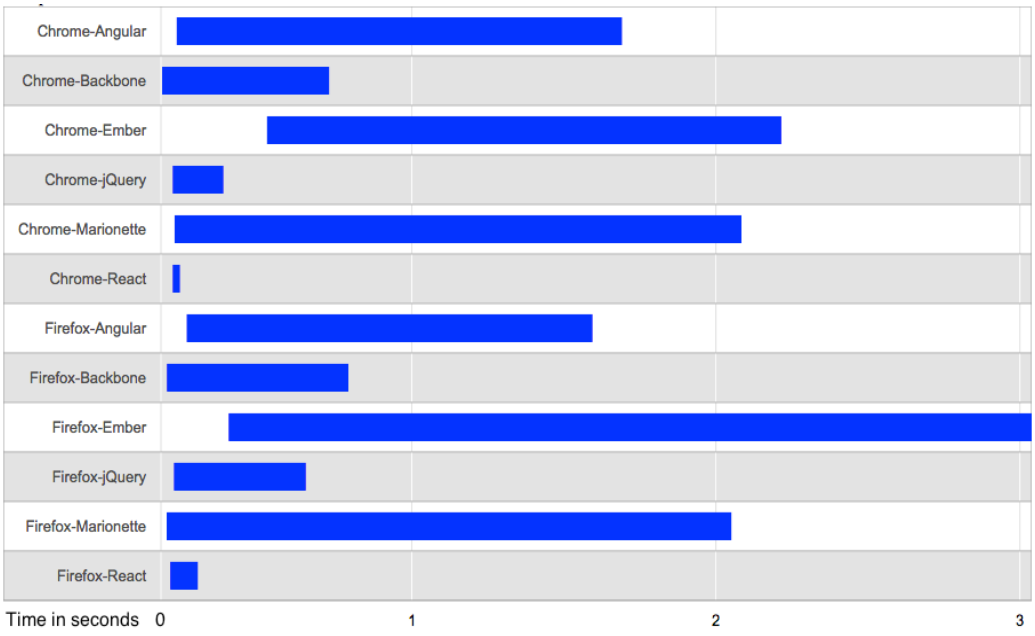

Figure 2.12: Time range required rendering and being ready for use with data loads from 0 to 1000

This figures does not consider the time required by the browser to download the framework, because this depends on the net, so this results would be like having the framework in the browsers cache. Both browsers behave almost the same and if the framework should be downloaded it would affect the results depending on their total size.

The charts show that the React instance of the application clearly outperforms the other frameworks. jQuery produced surprisingly good results whereas Ember was the worst in both scenarios but especially on the one with data.

2.6 Summary

When accessing the Web, users prefer applications that provide a good experience and performance. JavaScript helps making better transitions and animations, it also helps masking the latency and reducing the network traffic. In few words, JavaScript can be used to provide a better experience to the users and more performant. SPA started appearing in the Web, and with the increase of features on the client side the average weight of the applications has been rising at an accelerated pace. This increase on the complexity of the applications code results in a rising difficulty for some devices to execute them. Many frameworks, such as Angular or Backbone, surged to help structuring the growing amounts of generated code. But there are no studies that compare the performance among these client side frameworks.

The performance of the application should be considered when designing the software. We designed and developed simple and flexible tools that allow performance testing on real browsers in order to help making the right decisions. While developing and testing the tools, we found interesting differences in performance and the resources usage among the popular frameworks. Backbone based frameworks showed a tendency of requiring less resources and being faster than the competition. And React impressed us by being faster to load the application than the more mature frameworks. On the other hand, Ember showed an overall good performance on every test, while providing a robust framework that enables developers to write extensible applications.

The tools we developed demonstrated to be not only useful, but very flexible and easy to use. It was easy to extend the tests and mutate them in order to check a new aspect on the applications. The measurement of the time required executing a certain task plus the resources consumption of the browser are relevant factor when designing an application and should be considered in the decision-process.

It is important however to have in mind that there are also some limitations in our study. First, the amount of available memory might affect the performance

of the application. If the application requires more RAM than the amount available, the OS starts doing memory swapping that surely will affect the observed results. A second consideration has to do with the data-structures we used to store the data of the application. There are two places were the data might be located, the first one is the in-memory store, which is inherent to the framework and cannot be changed, and the second is in the localStorage. As discussed previously, each application makes use of a specific framework, which tends to be associated with a certain local persistence-library that changes that aspect. After reviewing the implementations, we noticed that some of these frameworks use an array to store the records while others make use of unique identifiers in the localStorage for faster individual access.

3. CONCLUSIONS AND FUTURE WORK

3.1 Review of the tools and general remarks

End users favor applications that deliver a good experience. Among the relevant aspects of this user experience are the load speed and how smooth it behaves when used. JavaScript plays an important role on this second aspect, helping better the transitions and animations, hiding the latency and reducing the network traffic. The increasing complexity of the Web applications has lead to larger amounts of code being executed on the client side. This has motivated experienced programmers to develop frameworks to ease their work and to the rise of single page applications (SPA).

All of this code has to be executed by the client itself. The problem surges when the device running the application has lower capabilities than expected, for example large applications on smartphones. This is a clear sign that the developers should consider where the application will be executed and design it accordingly. To help in this task we designed and developed an automation test environment for two browsers that allow programmers to measure the time requried for every task and the resources consumption for these tasks.

The differences among the extended browsers make this work more interesting. There are pieces of shared code among them, the available features and general behaviour of each browser differ from each other. To be able to use a given format for both browsers and receive a standard format as its output helped making the analysis easier to perform. One of the biggest successes of these tools is that their usage may be more extense than presented in this work, since it

provides the ability to test other changes such as JavaScript libraries, differences on the loadtime based on the CSS. However, the design has some limitations, for example the tools can only test applications that do not reload themselves. This limits the application to the asynchronous features of RIAs and SPAs.

3.2 Value of the research

The protype achieved its objectives, delivering interesting results in the JavaScript frameworks comparisson. The time and the resources consumption differences are evident when reviewing the results of the tests. From the graphics is easy to conclude that Backbone and Marionette have better overall results in terms of the required time and resources consumption in both browsers. While jQuery has the worst ones due to implementation that handles each task element individually instead of encapsulating the behaviour.

The comparisson among the other three frameworks is less evident. In both operations, task creation and task deletion, Ember and Angular use similar amounts of resources while React requires a little more. In the creation of tasks, the time required to complete the test in each framework was like its memory usage, but for the elimination task React behaves better than Angular while Ember continues consistently good.

When increasing the amount of data handled by the application, the load starts affecting the performance on the frameworks. The graphics show clearly how do these frameworks compare to each other. Backbone and Marionette get a little overhead while handling this amounts of data, resulting in a small slope on the curve. While the other tested frameworks present a steep slope in the required

time to handle a task on an increasing amount of tasks. On this test, jQuery does a better job than the other competitors and React behaves the worst.

In absolute terms, the time difference is not as significant as it may appear. We are talking about a really big number of actions and the worst framework took less than a fifth of a second to process the form or to delete a task in the computer where the tests were executed. But where this test show a big difference is in their resources consumption. While the Backbone-based versions required around 50MB to run the application, others took up to 300MB to do the same. Given that other applications and other tabs on the browser require their own resources to the system and that the application could be more complex than a simple task manager, devices may run out of resources causing the application to feel slugish.

There is another interesting result. When measuring the load time in the applications on different amounts of data, React does behave better than the other frameworks. All of the tested frameworks load fast when they do not have any data, but when heavily loaded differences appear. From all the tested frameworks, Ember is the slowest one in this test, taking nearly 3 seconds to be ready to use, while React achived that in less than half a second. This may be due to the innovation of the React framework, known as the Virtual DOM. This allows the framework to handle the changes in the DOM instead of the browser, pre-computing the changes and updating them all at once. The reason behind the slow reaction of the Ember framework may be due to the heavy implementation and more complex structre and bigger in-memory representations. On this test, jQuery does also behave very well.

3.3 Future work

The tools we developed for this work demosntrated to be useful, flexible and easy to use. Future work may consider extending them to be able to compare more aspects of applications or to automate some analysis based on the results of different applications. One interesting work over this data would be to port the code to work in smartphones and tablets, making it possible to explicitly test some features on mobile devices.

We expect that Web applications developers will start using our tools and others to better justify their decisions with hard facts. This extends also to the communities involved in designing new SPAs frameworks.

BIBLIOGRAFIA

A.B. Gizas, S.P. Christodoulou and T.S. Papatheodorou (2012), Comparative evaluation of javascript frameworks, In Proceedings of the 21st International Conference Companion on World Wide Web, pp. 513–514

A. Hidayat (2014), PhantomJS, Available at: http://phantomjs.org/ (Accessed 12 December 2014)

A. Mesbah and A. Van Deursen (2007), Migrating Multi-page Web Applications to Single-page AJAX Interfaces. In Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference, pp. 181-190.

A. Osmani and S. Sorhus (2014), TodoMVC, Available at: http://todomvc.com/ (Accessed 12 December 2014)

A. Osmani (2013), Developing Backbone.js Applications, O'Reilly

A. Osmani (2012), Journey Through The JavaScript MVC Jungle, Smashing Magazine, Available at: http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/ (Accessed: 12 December 2014)

A. Osmani (2012), JavaScript Application Architecture On The Road To 2015, Available at: https://medium.com/@addyosmani/javascript-application-architecture-on-the-road-to-2015-d8125811101b (Accessed: 18 December 2014)

A. Podelko (2014), Different Angles of Web Performance. Available at: http://calendar.perfplanet.com/2014/different-angles-of-web-performance/ (Accessed 5 January 2015)

A. Reitbauer (2014), W3C APIs in the Wild – Measurements beyond plain load times. Available at:http://calendar.perfplanet.com/2014/w3c-apis-in-the-wild-measurements-beyond-plain-load-times/ (Accessed 5 January 2015)

B. Bermes (2014), Fast-Forward Performance – The Future Looks Bright. Available at: http://calendar.perfplanet.com/2014/fast-forward-performance-the-future-looks-bright/ (Accessed 5 January 2015)

B. Frain (2014), CSS performance revisited: selectors, bloat, and expensive styles, Available at: http://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/ (Accessed: 18 December 2014)

B. McCormick (2014) The case for marionette js, Available at: http://benmccormick.org/2014/12/02/the-case-for-marionette-js/ (Accessed: 18 December 2014)

D. Espeset (2014) - Unpacking the Black Box: Benchmarking JS Parsing and Execution on Mobile Devices. Available at: https://speakerdeck.com/desp/unpacking-the-black-box-benchmarking-js-parsing-and-execution-on-mobile-devices (Accessed 5 January 2015)

D. Graziotin and P. Abrahamsson (2013), Making Sense Out of a Jungle of JavaScript Frame- works: Towards a Practitioner-Friendly Comparative Analysis, Proceedings of the 14th Interna- tional Conference, PROFES 2013, Paphos, Cyprus, June 12-14, 2013, pp. 334-337

D. Nolen (2013), The Future of JavaScript MVC Frameworks, Available at: http://swannodette.github.io/2013/12/17/the-future-of-javascript-mvcs/ (Accessed 12 December 2014)

D. Synodinos (2013), Top JavaScript MVC Frameworks, InfoQ, Available at: http://www.infoq.com/research/top-javascript-mvc-frameworks (Accessed: 12 December 2014)

D. Webb (2012), Improving performance on twitter.com, The Twitter Engineering Blog, Avail- able at: https://blog.twitter.com/2012/improving-performance-twittercom (Accessed: 12 December 2014)

G. Bahmutov, Improving Angular web app performance example. Available at: http://bahmutov.calepin.co/improving-angular-web-app-performance-example.html (Accessed: 18 December 2014)

G. Rauch (2014), 7 Principles of Rich Web Applications. Available at: http://rauchg.com/2014/7-principles-of-rich-web-applications/

G. Vargas (2014), Relieving Backbone Pain with Flux & React, Available at: http://dev.hubspot.com/blog/moving-backbone-to-flux-react (Accessed: 18 December 2014)

http://httparchive.org/ (Accessed: 5 January 2014)

http://jsforallof.us/2014/10/17/backbone-series-handling-asynchronous-data/ (Accessed: 18 December 2014)

http://www.w3.org/TR/DOM-Level-2-Core/introduction.html (Accessed: 12 December 2014)

H. Heitkötter, T. A. Majchrzak, B. Ruland, T. Webber (2013), Evaluating Frameworks for Creating Mobile Web Apps. Web Information Systems and Technologies 2013

I. Grigorik (2013): Browser Network, O'Reilly 2013

J. Bender, T. Parker, S. Jehl (2014), Research: Performance Impact of Popular JavaScript MVC Frameworks, Available at: http://www.filamentgroup.com/lab/mv-initial-load-times.html (Accessed: 18 December 2014)

J. J. Garret (2005), Ajax: A New Approach to Web Applications, Available at: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ (Accessed: 12 December 2014)

J. McCutchan, L. Lee (2013), Effectively Managing Memory at Gmail scale, Available at: http://www.html5rocks.com/en/tutorials/memory/effectivemanagement/ (Accessed: 18 December 2014)

J. Petersson (2012). Designing and implementing an architecture for single-page applications in Javascript and HTML5 (Master's thesis, Linköping University)

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

M. Schoeffler (2014), Simplify speed with the HALT number. Available at: http://calendar.perfplanet.com/2014/simplify-speed-with-the-halt-number/ (Accessed 5 January 2015)

M. Takada (2012), Single page apps in depth, Available at: http://singlepageappbook.com/ (Accessed: 12 December 2014)

P. Meenan (2014), WebPagetest - Website Performance and Optimization Test. Available at: http://www.webpagetest.org/ (Accessed 12 December 2014)

R. Gómez (2013), How Complex are TodoMVC Implementations, CodeStats Blog, Available at: http://blog.coderstats.net/todomvc-complexity/ (Accessed 12 December 2014)

S. Casteleyn, I. Garrigo, J.Mazón (2014), Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond. ACM Transactions on the Web, Vol. 8, No. 3, Article 18.

T. Kadlec (2014), JS Parse and Execution Time. Available at: http://timkadlec.com/2014/09/js-parse-and-execution-time/ (Accessed 18 December 2014)

S. Padmanabhan (2014), The Power of Perceived Performance. Available at: http://calendar.perfplanet.com/2014/the-power-of-perceived-performance/ (Accessed 5 January 2015)

S. Souders (2009), Even Faster Web Sites, O'Reilly 2009

S. Souders (2014), Resouce timing, Available at: http://www.stevesouders.com/blog/2014/11/25/serious-confusion-with-resource-timing/ (Accessed: 18 December 2014)

S. Stefanov and others (2012): Web Performance Daybook vol. 2, O'Reilly 2012

S. Vicencio, J. Navon (2014), JavaScript MV* Frameworks from a Performance Point of View. Journal of Web Engineering 2014.

U. Shaked (2014), AngularJS vs. Backbone.js vs. Ember.js, Available at: https://www.airpair.com/js/javascript-framework-comparison/ (Accessed: 18 December 2014)

A N E X O S

APENDINX A : AUTOMATION TEST EXAMPLE - JSON FORMAT

```
{
    "id":1414437765192,
    "tasks": [
                {
                        "action":"write",
                        "content":"hola1",
                        "selector":"#new-todo"
                },{
                        "action":"submit",
                        "content":"",
                        "selector":"#new-todo"
                }
    ],
    "type":"task",
    "urls": [
                "http://localhost:3000/static/emberjs"
    ]
}
```

APENDINX B : AUTOMATION TEST OUTPUT EXAMPLE – JSON FORMAT


```
{
    "type":"results",
    "data": {
        "startTime":1415030526112,
        "endTime":1415030547734,
        "timeElapsed":21622,
        "timeSeries":[
                {"start":1415030526113,"end":1415030526160,"elapsed":47},
                {"start":1415030526160,"end":1415030526200,"elapsed":40}
        ],
        "cpuSeries":[{"id":1415030526110,"value":0.10091220068415051}],
        "ramSeries":[{"id":1415030526111,"value":{"used":"7087546.24","total":"8388608.00"}}],
        "netSeries":[{"id":1415030526111,"value":{"up":342,"down":0}}],
        "staticStatistics":{
                "loadTime":{"id":"loadTime","value":273,"unit":"ms","label":"Load Time"},
                "latency":{"id":"latency","value":27,"unit":"ms","label":"Latency"},
                "frontEnd":{"id":"frontEnd","value":246,"unit":"ms","label":"Front End","limit":218},
                 "backEnd":{"id":"backEnd","value":27,"unit":"ms","label":"Back End","limit":55},

                "responseDuration":{"id":"responseDuration","value":0,"unit":"ms","label":"Response
Duration"},
                "requestDuration":{"id":"requestDuration","value":26,"unit":"ms","label":"Request
Duration"},
                "redirectCount":{"id":"redirectCount","value":0,"label":"Redirects"},
                "loadEventStart":{"id":"loadEventTime","value":10,"unit":"ms","label":"Load Event
duration"},
                "domContentLoaded":{"id":"domContentLoaded","value":0,"unit":"ms","label":"DOM
Content loaded"},
                "processing":{"id":"processing","value":154,"unit":"ms","label":"Processing
Duration"},
                "numOfEl":{"id":"numOfEl","value":50,"label":"DOM elements"},
                "cssCount":{"id":"cssCount","value":3,"label":"CSS"},
                "jsCount":{"id":"jsCount","value":20,"label":"JavaScript"},
                "imgCount":{"id":"imgCount","value":0,"label":"Images"},
                "dataURIImagesCount":{"id":"dataURIImagesCount","value":0,"label":"Data URI
images"},
                "inlineCSSCount":{"id":"inlineCSSCount","value":0,"label":"Inline CSS"},
                "inlineJSCount":{"id":"inlineJSCount","value":10,"label":"Inline JavaScript"},
                "thirdCSS":{"id":"thirdJS","value":0,"label":"3rd Party JavaScript"}
        },
        "url":"http://localhost:3000/static/react",
        "browser-vendor": "mozilla",
        "test_id":1414944743638
    }
}
```