A Test Environment for Web Single Page Applications (SPA)

Hans Findel and Jaime Navon
Department of Computer Science, Universidad Catolica de Chile, Vicuna Mackenna 4860, Santiago, Chile

Keywords: SPA, RIA, JavaScript Framework, Performance.

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 the 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 this additional code will produce. One possible reason is the lack of simple and flexible tools to test the application when it is actually running in a real browser. We developed a test framework and tools that allows the developer to easily put under test different implementation options. The tools are implemented as plugins for the most popular browsers so the application can run in its real environment. To validate the usefulness of the tools we performed extensive test to a 6 different implementations of a single web application. We found important performance differences across the tested frameworks. In particular, we found that the Backbone instance was faster and used fewer resources.

1 INTRODUCTION

Web technologies have been in constant evolution in an accelerated way. All browsers have embraced JavaScript (JS) as the de facto standard scripting language. This has given a crucial role in the development of web based applications to this programming language. JavaScript makes it possible to improve the user interaction and also allows masking the network latency (I. Grigorik, 2013) to produce a much better and fluid user experience, closer to the one of a native application (D. Webb, 2012).

The incorporation of asynchronous requests (S. Stefanov, 2012; S. Casteleyn, I. Garrigo, J.Mazón, 2014) through AJAX (J.J. Garret, 2005; A. Mesbah and A. Van Deursen, 2007; M. Takada, 2012) has also contributed to improve the user experience, and the popular jQuery library has simplified the use of AJAX, the manipulation of XML documents through the DOM standard and cross-browser compatibility; furthermore, it has also speeded up development (D. Graziotin and P. Abrahamsson, 2013).

The constant search for a better user experience and the rising of the mobile web has moved the code to the side of the client, giving birth first to rich internet applications (RIA) and later to the single page application (SPA). But this tendency to move most of the application code to the client demands for a better organization and structure of the JavaScript code fragments to manage complexity and also to make the code extensible.

In a classic web application the code is structured around the Model-View-Controller (MVC) pattern. This architecture has so many benefits that it 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 (A. Osmani, 2012).

In the same way that the use of a framework is the best way to force a MVC architecture at the server side, the use of a MVC framework on the client facilitates the organization and structure of the JavaScript code at 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, such as Angular, Backbone and Ember have become really popular (D. Synodinos, 2013). Lately React has emerged, bringing new ideas to the frameworks discussion.

Findel H. and Navon J..
A Test Environment for Web Single Page Applications (SPA).
DOI: 10.5220/0005428000470054
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 47-54
ISBN: 978-989-758-106-9

Each JavaScript framework takes a quite different approach to fulfill its goals. This has an impact on several relevant issues (R. Gómez, 2013) including the overall performance of the application. Nevertheless, this is seldom taken into account when deciding about the framework to 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 some visible effects of performance problems.

A recent study (S. Casteleyn, I. Garrigo, J.Mazón, 2014) 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 rapid increase in the use of mobile devices. The reason is that mobile devices are, in general, less powerful that a desktop or a laptop computer and, of course, the performance of the application depends on the machine where it runs.

We developed a test environment that facilitates the application of performance tests to any RIA or SPA and we 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 allowed us to validate our test environment in a real scenario, and on the other hand, to learn more about the performance behavior of most popular frameworks.

The rest of the paper is organized as follows. 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 our test tools to measure relative performance of the most popular JavaScript frameworks. Finally, in section 5 we provide a conclusion for this work.

2 RELATED WORK

In Gizas et al.(A.B. Gizas, S.P. Christodoulou and T.S. Papatheodorou, 2012), the relevance of careful choosing a JS framework is expressed. The research evaluates the quality, validation and performance of different JavaScript libraries/frameworks (ExtJS, Dojo, jQuery, MooTools, Prototype and YUI). The quality is expressed in terms of size, complexity and maintainability metrics. The performance tests corresponded to measurements of the execution time of the framework with SlickSpeed Selectors test framework. The tests are designed to evaluate the internals of the libraries themselves and do not mix with the application built upon. Additionally none of the evaluated libraries in Gizas work does provide an architectural context to develop an application, they only help access the DOM and to communicate through AJAX calls.

Graziotin et al. (D. Graziotin and P. Abrahamsson, 2013) extends Gizas work proposing a design towards a comparative analysis framework of JavaScript MV* frameworks to help practitioners select a suitable one (JavaScript framework). The authors interviewed some front-end developers in order to get first hand opinions on the relevant aspects to ease their work.

Vicencio et al. (S. Vicencio, J. Navon, 2014) carried out a more recent research work on the relative performance of client side frameworks. They focused on 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 several well-known frameworks (Backbone, Ember, Angular, Knockout) using the TodoMVC application as a basis. They did not build or implemented any test tools but they used existing tools Webpagetest (P. Meenan, 2014) and PhantomJS (A. Hidayat, 2014).

Petterson (J. Petersson, 2012) compares a tiny framework called MinimaJS to Backbone and Ember in a similar way as Vicencio (S. Vicencio, J. Navon, 2014). In his work, Runeberg (J. Runeberg, 2013) performs a comparative study between Backbone and Angular. One of the aspects revised in the study covers some performance test with PhantomJS for page automation.

There are few comparative studies on frameworks for the mobile web as well. Heitkötter (H. Heitkötter, T. A. Majchrzak, B. Ruland, T. Webber, 2013) elaborates a set of evaluation criteria for converting web applications into apps for the different mobile operative systems. This is a future step on the investigation of performance, since it may be a good way to reduce the code that is constantly downloaded from mobile devices.

Nolen (D. 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 (A. Osmani, 2013) one. The test includes creating, toggling and deleting 200 to-do entries. The differences in the time it takes to each framework are significant, showing that Om outperforms Backbone.js.

On the other hand, on this work we focused our attention to measuring the performance of MVC frameworks in SPAs, taking into account the resources consumed by the browser. In order to accomplish this goal, we designed and developed tools specialized for two of the major browser (Google Chrome and Mozilla Firefox). These tools differ from the existing ones, because they take into account measurements for more indicators such as the memory and CPU usage. Additionally, these tools will be open source so they can be edited to measure more specific values in the testing process.

3 THE TEST ENVIRONMENT

3.1 General Aspects

We mentioned on section 2 previous work on performance of JavaScript frameworks carried on with the help of two existing tools: PageSpeed and PhantomJS. The first measures the loading time of the application whereas PhantomJS allows the user to program an interaction with the application (page automation test). Although these tools can do the job and they are relatively easy to use, they have some important drawbacks. One problem of these tools is that they use a headless webkit-based browser to perform the tests and not the real browsers where the application will run. Another limitation is that these tools cannot be extended so we cannot modify or add new metrics to them. Finally we needed tools that facilitate the creation of new tests.

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, measuring the applications performance. This interface provides us the same input and output formats making it easy to compare test results from 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 it can also perform dynamic analysis by capturing metrics such as time required to load (S. Stefanov, 2012) and the time to accomplish each task of the given set. It can also measure the following resources consumed by the browser: CPU usage, RAM usage and Network usage (downloads and uploads during the test).

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.

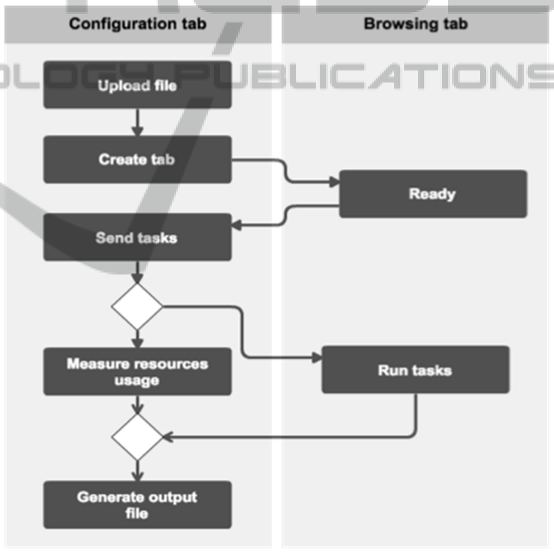

Figure 3.1: Chrome plugin usage flow.

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 that already contains the scripts to communicate with the main tab and to execute some actions on the website. The main tab starts monitoring the resources used by the created tab and then sends to it 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 page loading process and the characteristics of the DOM structure. Then the tab proceeds to send all this information back to the configuration tab, which then aggregates all the resources consumption and downloads the data in a standard JSON format in the computer (Figure 3.1)

3.3 Mozilla Firefox Implementation

The structure of the Mozilla Firefox plugin is more straightforward. Since in this browser the interaction across tabs is less flexible, we decided instead to extend the browser internal API. To achieve this, we leveraged on an existing low-level plugin that provides the tabs the option to subscribe to a feed of the consumption of a certain resource. For example, with a simple JavaScript command, the tab can handle, in real-time stream of data, the RAM usage of the browser through a callback function, which we call resource-handler.

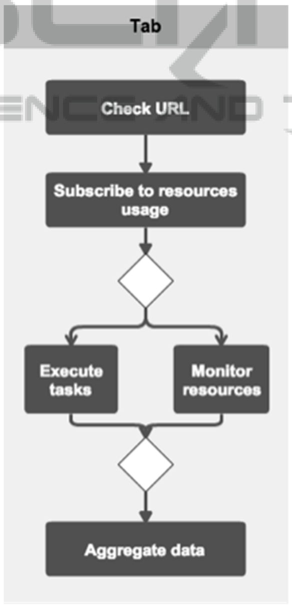

Figure 3.2: Mozilla Firefox plugin usage flow.

This plugin adds a little HTML fragment above the code of the visited websites. Within the plugins HTML fragment is a file input form where the JSON automation test can be uploaded. Then the browser verifies if it matches the current URL and parses the instructions. Using the low-level plugin, this extension starts monitoring the resources consumed by the browser. After subscribing to the data-feed, the tab executes the instructions given in the JSON file while the resource-handlers keep track of the resources usage. Once the tab is ready executing the instructions, it collects the information about the loading process and aggregates with the time required to complete each task and the resources usage collected by the handlers. Then the tab proceeds to download the aggregated data into the users computer (as shown in Figure 3.2).

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 DOM analysis.

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 (S. Souders, 2014). This shared script leverages on this and stores some relevant values from it, but could be edited to consider other timing metrics. 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).

3.5 Limitations

These tools have one major limitation. Our tools can test features that do not require the tab to refresh itself. This limitation does no harm as long as we test SPAs, but discard the usage of our tools out of the single page application spectrum.

Another detail to take into consideration is that the results provided by each browser tool are not directly comparable with each other. Since the implementations differ, each extension impacts the performance differently. But on the other hand, the impact within each browser is comparable among the different tests and frameworks, because its performance cost is a function on the number of tasks to perform in the automation test.

4 USING THE TOOLS

4.1 The Application

We wanted to compare the most popular 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 frameworks. But if we had done that we could have introduced a bias related to our relative expertise on the use of a particular framework.

Fortunately, A. Osmani and S. Sorhus created an open source project called TodoMVC (A. Osmani and S. Sorhus, 2014) where you can find the same application (a task manager) 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 so it is fair to assume that it is close to the best possible implementation in each case.

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 backing them. Marionette is a framework that operates over Backbone and it was included to match the amount of features provided by the first three, in an attempt to make a comparison in even terms. React, backed by Facebook, has been raising a lot of attention lately and brought a few new ideas to the development environment. 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.

4.3 The Tests

4.3.1 Add 1000 Tasks to the to Do List

We carried on this test by creating an automation test that, starting from a clean state, submits one thousand times a new task to be inserted into the task list. The idea behind this test is to measure the behaviour of web applications using these frameworks when exposed to different amounts of data. The results obtained in the experiments show how the frameworks behaviour changes under these conditions. These results are represented in the figures presented in this section.

We performed this experiment for each of the 6 instances of the application and for the two browsers using their respective plugins. The results are shown in Figure 4.1 and 4.2, for Chrome and Firefox, respectively.

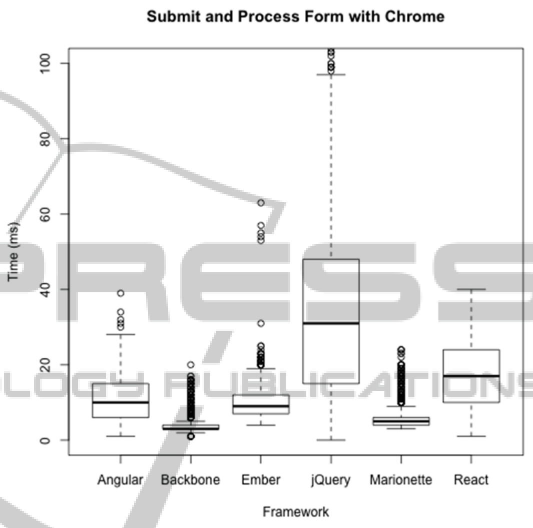

Figure 4.1: Time required to insert a task (Chrome).

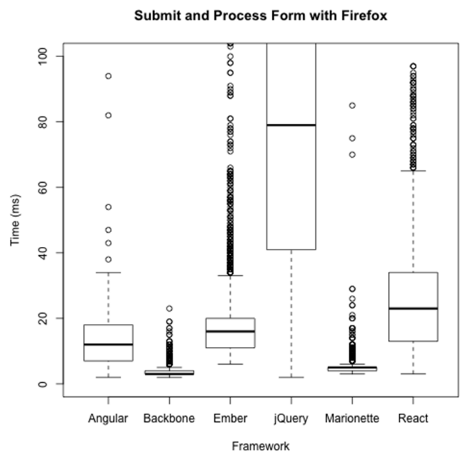

Figure 4.2: Time required to insert a task (Firefox).

The previous figures show the time required to perform the tasks. But it depends heavily on the machine running the tests, so our tools also allow us to measure the resources usage (relevant for SPA in the mobile devices context). In order to complement the tests and to use more of our tools features, we compared the frameworks under their RAM usage. The results for Chrome are shown in Figure 4.3.

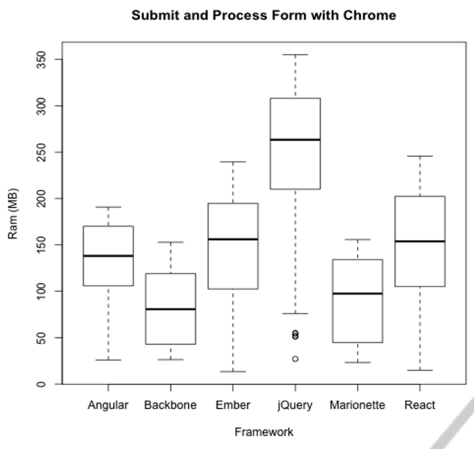

Figure 4.3: RAM usage.

The bad results for jQuery are explained by the fact that the used implementation requires an event handler for each task, creating many instances of the same function but with distinct targets. This massive creation of event handlers builds up, leading to a slower execution of the given tasks. The memory usage depends on the framework itself that differ between them; additionally the memory usage also depends on the representation of the model instances within each framework and the event listeners on the DOM. The chart also reveals that the Backbone instances use about half the memory of the other frameworks.

4.3.2 Delete All Tasks One by One

The test simulates a click on the button with the class "destroy" (HTML class, selectable by ".destroy") within every task element. The results corresponding to Chrome are presented in Figure 4.4 (Firefox produced very similar results).

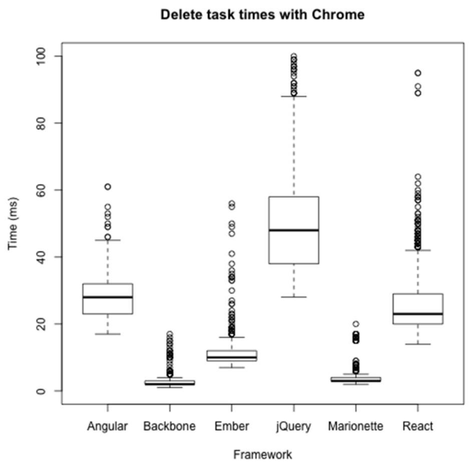

Figure 4.4: Time to delete a task in Chrome.

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 thousand of items. We wanted to test how each of the application instances respond to an incremental load of the to-do list from 0 to 5000. Figures 4.5 (Angular) and 4.6 (Backbone) reveal how the required time to insert a new task depends on the previously inserted ones.

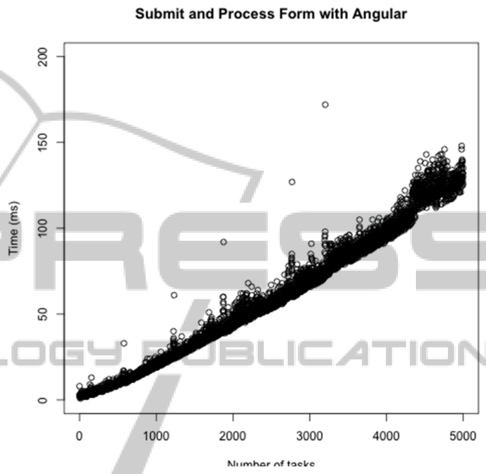

Figure 4.5: Progressive times to add a task.

(Angular instance)

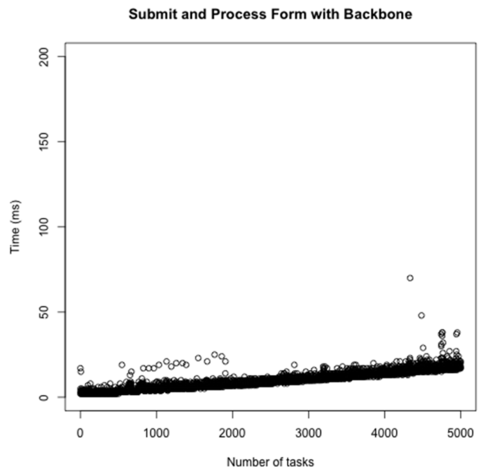

Figure 4.6: Progressive times to add a task (Backbone instance).

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 corresponds to tasks added when the list included many tasks.

From the charts, it is clear that Backbone does not degrade its performance when the data-load is incremented. On the other side, we can see that in Angular the application grows slower on each task insertion.

5 CONCLUSIONS

Performance is always an important issue for web applications. Both, the surge in access to the web through mobile devices and strong user experience requirements, have produced a huge increase in the quantity and complexity of the JavaScript code on the client side. JavaScript frameworks such as Angular or Backbone have come to the rescue but each of these pieces of code may have an impact on the performance of the application. Simple and flexible tools that allow testing the performance behaviour of the application running in the real browser can help us to take the right decisions in terms of the performance requirements.

Our tools were put to a test by examining the behaviour of a single page application that had been implemented with the help of the most popular frameworks. We found interesting differences in performance (and use of RAM) that were no obvious beforehand. Backbone based versions outperformed the other frameworks implementations. Particularly in the progressive test, Angular requires significantly more time to execute the same tasks as the Backbone implementation when handling larger amounts of data.

The tools themselves demonstrated to be not only useful, but quite flexible and easy to use. Given its simplicity, it should be considered to compare future frameworks or versions of them.

ACKNOWLEDGEMENTS

This work has been partially supported by the National Research Center for Integrated Natural Disaster Management CONICYT/FONDAP/15110017.

COMMENTS

At the time of this publication, the tools for both browsers have not been open-sourced.

REFERENCES

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

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

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

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.

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

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

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

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

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.

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

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).

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).

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

A. Hidayat (2014), PhantomJS, Available at: http://phantomjs.org/ (Accessed 12 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).
R. Gómez (2013), How Complex are TodoMVC Implementations, CodeStats Blog, Available at: http://blog.coderstats.net/todomvc-complexity/ (Accessed 12 December 2014).
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).
S. Souders (2014), Resouce timing, Available at: http://www.stevesouders.com/blog/2014/11/25/serious-confusion-with-resource-timing/ (Accessed: 18 December 2014).

APPENDIX

Example automation test file:

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